Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo
Preface

Recent years have witnessed the appearance of new paradigms for designing distributed applications where the application components can be relocated dynamically across the hosts of the network. This form of code mobility lays the foundation for a new generation of technologies, architectures, models, and applications in which the location at which the code is executed comes under the control of the designer, rather than simply being a configuration accident.

Among the various flavors of mobile code, the mobile agent paradigm has become particularly popular. Mobile agents are programs able to determine autonomously their own migration to a different host, and still retain their code and state (or at least a portion thereof). Thus, distributed computations do not necessarily unfold as a sequence of requests and replies between clients and remote servers, rather they encompass one or more visits of one or more mobile agents to the nodes involved.

Mobile code and mobile agents hold the potential to shape the next generation of technologies and models for distributed computation. The first steps of this process are already evident today: Web applets provide a case for the least sophisticated form of mobile code, Java-based distributed middleware makes increasing use of mobile code, and the first commercial applications using mobile agents are starting to appear.

This volume contains the proceedings of the Fifth International Conference on Mobile Agents (MA 2001). MA 2001 took place in Atlanta, Georgia, USA, at the Georgia Center for Advanced Telecommunications Technology (GCATT), on December 2–4, 2001. The ambitious goal of MA 2001 was to gather researchers and practitioners from all over the world and shed some light on the open issues related to the exciting research topic of code mobility.

The first conference in this series was held in 1997 in Berlin, and since then it has been, by number of attendees and by quality and breadth of the research disseminated, among the top events for the community of researchers and practitioners interested in mobile code and mobile agents. The previous two conferences were held together with the International Symposium on Agent Systems and Applications (ASA) as joint ASA/MA events that aimed at gathering researchers interested in all the flavors of agent systems, e.g., including also intelligent and non-mobile agents. Although these joint events were very successful, MA 2001 was presented as a stand-alone event, entirely focused on the original target of mobile code and mobile agents. Our goal with this and future events is to strengthen the MA conference as the international venue at which the best and latest results in the topics of mobile code and mobile agents are disseminated and discussed.

The conference received 75 submissions from authors all over the world. The CyberChair system [www.cyberchair.org] greatly simplified the submission and review process. The Program Committee, composed of 20 of the most distinguished researchers in code mobility, reviewed all of the papers carefully. Each paper was assigned to at least three reviewers – four in the case of papers authored by Program Committee members. Reviewers were asked to declare in
advance potential conflicts of interest, to allow a proper assignment of papers and ensure fair reviews. Moreover, this information was used at the Program Committee meeting, that took place in Milan at the end of May, where reviewers with a conflict of interest on a paper were asked to leave the room during the related discussion. After a full-day meeting, the Program Committee selected the 18 papers included in the technical program.

In addition to these papers, we were honored that two distinguished experts accepted our invitation to give keynote presentations. Fred Schneider (Cornell University, USA) shared his views about the past, present, and future of mobile agent research, while Aleta Ricciardi (Valaran Corporation, USA) reported on her first-hand experience in applying code mobility within a real-world industrial context. The program was completed by a “Posters and Research Demos” session, and by four tutorials by leading experts in the field.

Conferences are the result of the concerted efforts of several people. First of all, I would like to express, personally and on behalf of the rest of the Organizing Committee, my appreciation to the authors of the submitted papers, and sincerely thank the members of the Program Committee and the external reviewers for their fundamental contribution to ensuring the quality of this conference. I would also like to thank the General Chair of MA 2001, David Kotz, and the rest of the Organizing Committee for their work in making this event a success. Finally, I would like to acknowledge and thank the IEEE Technical Committee on the Internet and the IEEE Computer Society for sponsoring the event, and Nokia and Georgia Tech College of Engineering for supporting it.

September 2001

Gian Pietro Picco
Organization

The Fifth International Conference on Mobile Agents (MA 2001) took place at the Georgia Center for Advanced Telecommunications Technology (GCATT) in Atlanta, Georgia, USA, on December 2–4, 2001.

Executive Committee

General Chair: David Kotz (Dartmouth College, USA)
Program Chair: Gian Pietro Picco (Politecnico di Milano, Italy)
Tutorials: David Wong (Mitsubishi Electric Research Labs, USA)
Advertising: Marco Cremonini (Dartmouth College, USA)
Registration: Lori A. Terino (Dartmouth College, USA)
Local Arrangements: Ashraf Saad (Georgia Institute of Technology, USA)
Treasurer: Robert S. Gray (Dartmouth College, USA)
Webmaster: Guanling Chen (Dartmouth College, USA)

Program Committee

Israel Ben-Shaul Israel Institute of Technology, and Versedge Technologies, Israel
Lubomir F. Bic University of California Irvine, USA
Luca Cardelli Microsoft Research, UK
Rocco De Nicola Università di Firenze, Italy
Andrzej Duda LSR-IMAG, France
Robert S. Gray Dartmouth College, USA
Shinichi Honiden National Institute of Informatics, Japan
Günter Karjoth IBM Zurich Research Laboratory, Switzerland
Dag Johansen University of Tromso, Norway
Danny B. Lange Vocomo Software, USA
Thomas Magedanz IKV++, Germany
Keith Marzullo University of California San Diego, USA
José Meseguer SRI International, USA
Amy L. Murphy University of Rochester, USA
Kurt Rothermel Universität Stuttgart, Germany
Niranjan Suri University of West Florida, USA
Anand Tripathi University of Minnesota, USA
Christian Tschudin Uppsala University, Sweden
Giovanni Vigna University of California Santa Barbara, USA
Franco Zambonelli Università di Modena e Reggio Emilia, Italy
VIII Organization

Steering Committee

Robert S. Gray 
David Kotz 
Danny B. Lange 
Friedemann Mattern 
Gian Pietro Picco 
Kurt Rothermel 

Dartmouth College, USA 
Dartmouth College, USA 
Vocoro Software, USA 
ETH Zurich, Switzerland 
Politecnico di Milano, Italy 
Universität Stuttgart, Germany

Sponsoring Organizations

MA 2001 was sponsored by the IEEE Technical Committee on the Internet and the IEEE Computer Society, and supported by Nokia and the Georgia Tech College of Engineering.
Referees

Miki Abu
Yariv Aridor
Christian Becker
Lorenzo Bettini
Bozhena Bidyuk
Michele Boreale
Cora Burger
Michael Dillencourt
Jesus Favela
Eugene Gentleman
Yonad Gidron
Daniel Hagimont
Nguyen Hoa Binh
Fritz Hohl
Kjetil Jacobsen
Neeran Karnik
Hairong Kuang
Michele Lorti
Koji Noguchi
Lei Pan
Stefan Pleisch
Rosario Pugliese
Franck Rousseau
Ant Rowstron
Markus Straßer
Wolfgang Thöllmann
Yaron Weinsberg
Contents

Security

On the Robustness of Some Cryptographic Protocols for Mobile Agent Protection ........................................ 1
Volker Roth
(Paulinoer Institute for Graphics Data Processing, Germany)

Trust Relationships in a Mobile Agent System ................. 15
Hock Kim Tan and Luc Moreau (University of Southampton, UK)

Evaluating the Security of Three Java-Based Mobile Agent Systems ....... 31
Sebastian Fischmeister, Giovanni Vigna, and Richard A. Kemmerer
(University of California Santa Barbara, USA)

Models and Architectures

Formal Specification and Verification of Mobile Agent Data Integrity Properties: A Case Study ...................................... 42
Xavier Hannotin, Paolo Maggi, and Riccardo Sisto
(Politecnico di Torino, Italy)

Lime Revisited (Reverse Engineering an Agent Communication Model) .... 54
Bohdan Carabunar, Marco Tulio Valente, and Jan Vitek
(Purdue University, USA)

Dynamic Adaptation of Mobile Agents in Heterogenous Environments ..... 70
Raimund Brandt (skyguide, Switzerland) and
Helmut Reiser (University of Munich, Germany)

Applications

Fast File Access for Fast Agents ........................................ 88
Eugene Genesman, Lubomir F. Bic, and Michael B. Dillencourt
(University of California Irvine, USA)

Flying Emulator: Rapid Building and Testing of Networked Applications for Mobile Computers ....................... 103
Ichiro Satoh (National Institute of Informatics, Japan)

Crawlets: Agents for High Performance Web Search Engines ............ 119
Prasannaa Thathi, Po-Hao Chang, and Gul Agha
(University of Illinois at Urbana-Champaign, USA)
Communication

An Efficient Mailbox-Based Algorithm for Message Delivery in Mobile Agent Systems ........................................... 135
  Xingyu Peng (Nanjing University, China),
  Jiannong Cao (Hong Kong Polytechnic University),
  Jian Li (Nanjing University, China), and
  Henry Chan (Hong Kong Polytechnic University)

Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent Environment .......................... 152
  Klaus Haller and Heiko Schuld
  (Swiss Federal Institute of Technology, Switzerland)

A Scalable and Secure Global Tracking Service for Mobile Agents .......... 169
  Volker Roth and Jan Peters
  (Fraunhofer Institut für Graphische Datenverarbeitung, Germany)

Run-Time Support

Translating Strong Mobility into Weak Mobility ................................. 182
  Lorenzo Bettini and Rocco De Nicola (Università di Firenze, Italy)

Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture ........................................... 198
  Torsten Illmann, Timm Krüger, Frank Kargl, and Michael Weber
  (University of Ulm, Germany)

Portable Resource Reification in Java-Based Mobile Agent Systems ......... 213
  Alex Villazón (University of Geneva, Switzerland) and
  Walter Binder (CoCo Software Engineering, Austria)

Quantitative Evaluation and Benchmarking

Mobile-Agent versus Client/Server Performance:
  Scalability in an Information-Retrieval Task ............................... 229
  Robert S. Gray, David Kotz, Ronald A. Peterson,
  (Dartmouth College, USA),
  Joyce Barton, Daria Chacón, Peter Gerken, Martin Hofmann
  (Lockheed-Martin Advanced Technology Laboratory, USA),
  Jeffery Bradshaw, Maggie Brody, Renia Jeffers, and Niranjan Suri
  (University of West Florida, USA)
Performance Evaluation of Mobile-Agent Middleware:
A Hierarchical Approach .................................................. 244

Marios Dikaiakos, Melinos Kyriakou, and George Samaras
(University of Cyprus)

Scheduling Multi-task Agents ................................................. 260

Rong Xie, Daniela Rus (Dartmouth College, USA), and
Cliff Stein (Columbia University, USA)

Author Index ................................................................. 277
On the Robustness of Some Cryptographic Protocols for Mobile Agent Protection

Volker Roth

Fraunhofer Institut für Graphische Datenverarbeitung
Rundeturmstraße 6, 64283 Darmstadt, Germany
vroth@igd.fhg.de

Abstract. Mobile agent security is still a young discipline and most naturally, the focus up to the time of writing was on inventing new cryptographic protocols for securing various aspects of mobile agents. However, past experience shows that protocols can be flawed, and flaws in protocols can remain unnoticed for a long period of time. The game of breaking and fixing protocols is a necessary evolutionary process that leads to a better understanding of the underlying problems and ultimately to more robust and secure systems. Although, to the best of our knowledge, little work has been published on breaking protocols for mobile agents, it is inconceivable that the multitude of protocols proposed so far are all flawless. As it turns out, the opposite is true. We identify flaws in protocols proposed by Corradi et al., Karjoth et al., and Karnik et al., including protocols based on secure co-processors.

Keywords: mobile agent security, cryptanalysis, breaking security protocols.

1 Introduction

Analyzing cryptographic protocols for mobile agent protection means meeting old friends and foes. In [1,2], Abadi, Needham, and Anderson summarized some rules and principles of good and bad practice for designing cryptographic protocols. We show in this paper that their advice was not followed thoroughly in the design of some cryptographic protocols meant to protect mobile agents against certain attacks by malicious hosts. We first summarize the typical objectives of the protocols we analyze:

Objective 1 (Confidentiality) Mobile agents shall reveal cleartext only while being on trusted hosts.

Objective 2 (Integrity) The agents shall be protected such that they can acquire new data on each host they visit, but any tampering with pre-existing data must be detected by the agent’s owner (and possibly by other hosts on the agent’s itinerary).

The general objective here is to protect certain features of a mobile agent against malicious hosts. By assumption, the host of the agent’s owner is always trusted. Some of the protocols address both objectives simultaneously, others address just one. All
protocols are targeted at protecting free-roaming mobile agents. In other words, mobile
agents that are free to choose their respective next hop dynamically based on data they
acquired in the course of their execution.

Unfortunately, these protocols expose hosts in a way that allows an attacker to abuse
them as oracles for generating protocol data. This enables attacks on cryptographic
protocols devised in [3,4,5,6]. In some cases this leads to a complete compromise of the
protocol’s security objectives. In other cases the adversary is able to forge and replace
subsets of the protocol data in a way that makes it impossible for an agent’s owner to
detect the tampering. The important observation here is not that protocol data acquired
by agents can be truncated (some authors already acknowledge this possibility) but that
the attacker can exercise control over the data returned by an agent.

The attacks we mount on the analyzed protocols can best be described as interleaving
attack [7, §10.5], which is “an impersonation or other deception involving selective com-
bination of information from one or more previous or simultaneously ongoing protocol
executions (parallel sessions), including possible origination of one or more protocol
executions by an adversary itself. Figure 1 illustrates the general scheme of attack: the
adversary receives an agent, and copies protocol data back and forth between this agent
and agents she sent herself.

2 Some Protocol Failures

We will write encryption of some plaintext into a ciphertext symbolically as $c = \{m\}^K$,
where $K$ is the key being used. A digital signature will be written as an encryption with
a private signing key $S^{-1}$. We will write $S^{-1}(m)$ when we refer to the bare signature
rather than the union of the signature and the signed data. We assume that the identity
of the signer can be extracted from her signature. A cryptographic hash of some input
will be written $h(m)$. Unless noted otherwise, we assume that $h$ is preimage resistant
and collision resistant [7, §9.2.2], which implies that $h$ must also be 2nd-preimage
resistant [7, §9.2.5]. When \( A \) sends some message \( m \) to \( B \) we will write \( A \rightarrow B : m \). We will write \( A \rightarrow B : \{m\}_{K_{A,B}} \) when \( m \) is sent over a confidential channel. Concatenation of \( m_1 \) and \( m_2 \) is written as \( m_1 || m_2 \). For ease of reading, we refer to some entities by their nicknames, e.g., Alice, Bob, and Eve. In general, Eve will play the role of the adversary, Alice will play the role of the victim agent’s owner, Bob and Dave will play the role of additional entities taking part in the protocols. The itinerary of Alice’s agent is written as \( i_k, \ldots, i_1 \), where \( i_k = \text{Alice} \) and \( i_1 \) is the host currently visited by the agent.

2.1 Decrypacting the Targeted State

In [3], Karnik and Tripathi propose a targeted state as a means to protect the confidentiality of data carried by an agent. The idea is to make this data available to the agent only when it is on a host that is trusted with respect to keeping this data confidential from other agents and hosts. In order to achieve this, the plaintext is encrypted with the public key of the trusted host. The targeted state looks like this:

\[
\{ \{m_1\}_{K_{i_1}}, \ldots, \{m_n\}_{K_{i_n}} \}_{S_{i_1}^{-1}}
\]

The targeted state is signed by Alice, who is the originator of the agent owning the targeted state. Having received an agent, each host inspects the targeted state for ciphertexts it can decrypt. If so, the host decrypts it using its own private decryption key, and makes the cleartext available to the agent.

Below, we illustrate the attack on this protocol. Without loss of generality, we assume that the agent’s targeted state contains a single ciphertext, which is encrypted with the public key of Bob. Alice first sends the agent to Eve from whom it hops to Bob and then returns to Alice. The protocol starts as follows (for simplicity, we assume here that an agent initially consists only of its targeted state and its program \( P_A \)):

\[
A \rightarrow E : P_A , \{ \{m\}_{K_B}\}_{S_{i_1}^{-1}}
\]

The attack is straightforward. Eve strips off Alice’s signature, copies \( \{m\}_{K_B} \) into the targeted state of an agent of her own, signs this targeted state, and sends her agent to Bob:

\[
E \rightarrow B : P_E , \{ \{m\}_{K_B}\}_{S_{E}^{-1}}
\]

\[
B : P_E , \{ \{m\}_{K_B}\}_{S_{E}^{-1}}, \{ \{m\}_{K_B}\}_{K_B^{-1}} = m
\]

Bob innocently decrypts the targeted state using his own private key and makes the resulting plaintext available to the agent. The agent then migrates back to Eve carrying the plaintext.

\[
B \rightarrow E : P_E , \{ \{m\}_{K_B}\}_{S_{E}^{-1}}, m
\]

Eve now is in possession of the plaintext which should be available only to Bob; Alice never detects the attack. The problem with this protocol is that, due to a lack of redundancy in the ciphertext, Bob can be abused as an oracle. Alice needs to include
an unforgeable identifier of her agent in the ciphertext, e.g., \(b(\Pi_A, A)\) (see [8] for an alternative approach). Even then, the agent’s program must be unique for each agent\(^1\) and designed carefully such that it can not be abused in the way illustrated above by means of malicious state changes.

2.2 Forging the Append Only Container

In addition to the targeted state, Karnik and Tripathi also propose an append only container. The idea is to protect a container of objects in an agent such that new objects can be added to it but any subsequent modification of an object contained therein can be detected by the agent’s owner. The protocol relies on an encrypted checksum, whose initial value \(C_0 = \{r\}_{K_A}\) is computed by Alice (the agent’s owner) based on a random nonce \(r\). The nonce must be kept secret by Alice, and is used in the verification protocol upon the agent’s return. The append only container is defined as follows:

\[
\{\{m_1\}_{S_{m_1}}, \ldots, \{m_n\}_{S_{m_n}}, C_n\}
\]

Whenever a new object is appended to the append only container, the checksum is updated\(^2\) as given below:

\[
C_{n+1} = \{C_n \parallel S_{m_{n+1}}^{-1}(m_{n+1})\}_{K_A}
\]

The signer of the appended object is the host on which the append operation takes place. Upon the agent’s return, Alice successively decrypts the checksums, extracts the signature, and verifies the signature against the corresponding object in the container. The last checksum must equal the initial nonce.

We now assume that Eve received Alice’s agent and she knows \(C_j\) for some \(1 \leq j \leq n\). Eve always knows \(C_m\), because it is embedded in the container. She might collude with other servers which the agent visited before, or she might be part of a loop in the agent’s itinerary. In these cases, Eve might discover a checksum \(C_j\) with \(j < n\).

At this stage, Eve has multiple choices. She can either truncate the container up to the \(j\)th object and grow a fake stem by releasing the agent. Or she can remove, add or replace arbitrary objects \(m_l\) with \(l > j\) in the name of other hosts. In order to do this, Eve creates an agent with the object that she wants to add at \(j + 1\), and an append only container of her own, with checksum \(C_j\) as its initial value. Eve now sends her agent to Bob. There, Eve’s agent inserts \(m_{j+1}\) in its own targeted state and migrates back:

\[
E \rightarrow B : \Pi_E, \ m_{j+1}, \{C_j\}
\]

\[
B \rightarrow E : \Pi_E, \ \{\{m_{j+1}\}_{S_{m_{j+1}}}, \{C_j \parallel S_{m_{j+1}}^{-1}(m_{j+1})\}_{K_E}\}
\]

\(^1\) Otherwise Eve can still cut & paste targeted states back and forth between agents that are owned by Alice and which share the same program.

\(^2\) In the original protocol description, the signature and identity of the server are appended. On the other hand, we assume that the signer’s identity can be extracted from the signature and appending it is, therefore, redundant.
Upon the agent’s return, Eve decrypts the checksum using her own private key, and re-encrypts it using the public key of Alice:

\[ C_{j+1} = \{ \{ C_j \| S_B^{-1}(m_{j+1}) \} K_E \} K_A^{-1} \]

\[ = \{ C_j \| S_B^{-1}(m_{j+1}) \} K_A \]

Then, she constructs a new container:

\[ \{ m_1 \}_{S_{j_1}^{-1}}, \ldots, \{ m_j \}_{S_{j_j}^{-1}}, \{ m_{j+1} \}_{S_{j_{j+1}}^{-1}}, C_{j+1} \]

which replaces the previous container of Alice’s agent. This process is repeated with the new checksum until Eve is pleased with the result, and releases Alice’s agent. Bob is not able to detect the attack, because \( C_j \) is not publicly verifiable (it is encrypted with Alice’s public key). All Bob can see is the length of \( C_j \), from which he can estimate the number of objects that must be in the append only container. So if Eve wants to make sure that Bob has no reason to get suspicious then she adds \( j \) signed objects to her agent’s container before she sends it to Bob. As long as these objects are properly signed it does not matter who signed them and where she got them.

Once again, a lack of redundancy allows Eve to abuse other hosts as oracles, this time for the purpose of signing and checksum computation rather than decryption.

2.3 Forging the Multi-hops Protocol

In [4], Corradi, Montanari, and Stefanelli propose a protocol they call multi-hops, which has the same purpose as the append only container presented by Karmik and Tripathi. It falls prey to the same type of attack. However, this time, the faked agent needs to do one more hop to complete its attack. For reference, we summarize the multi-hops protocol below.

Let \((\Pi, M, P)\) be an agent where \(\Pi\) is static (immutable) code and initialization data, \(M\) is (mutable) application data, and \(P\) is protocol data (meta information required by the protocol). Alice initializes her agent with \((\Pi_A, \epsilon, \epsilon)\). The protocol additionally requires a nonce \(\gamma\) and a message authentication code \(\mu\). The initial values are \(\gamma_0 = h(r)\) and \(\mu_0 = \epsilon\), where \(r\) is chosen randomly. On each hop, the agent can add some data \(m\) to its application data, which is then protected by the host using the multi-hops protocol. The protocol is defined as given below:

\[ \gamma_n = h(\gamma_{n-1}) \]
\[ \mu_n = h(m_n, \gamma_{n-1}, \mu_{n-1}, i_{n+1}) \]
\[ P_n = P_{n-1} \| S_{\gamma_n}^{-1}(\mu_n) \]
\[ M_n = M_{n-1} \| m_n \| i_n \]

\[ i_n \rightarrow i_{n+1} : (\Pi_A, M_n, P_n) \rightarrow \{ \gamma_n \} K_{i_{n+1}}, \mu_n \]
The message authentication code $\gamma_n$ serves as a chaining relation that binds results previously obtained by the agent to the ones obtained at the current host and to the identity of the agent’s next hop.

Due to this chaining relation, the attack cannot be executed in the same way as it is done for the append only container. The resulting star shaped itinerary with Eve in the center would be too obvious in the protocol data. What Eve has to do here is to plan ahead one step.

Again, we assume that Eve is $i_n$, and she knows some $\gamma_{j-1}$ for $1 \leq j \leq n$. She received the agent so she always knows $\gamma_{n-1}$ and $\mu_{n-1}$. She can obtain $\gamma_{j-1}$ with $j < n$ by colluding with other hosts or as a result of loops in the agent’s itinerary. Due to the chaining relation — remember that $\mu_{j-1}$ is computed on $i_j$ — Eve does not have free choice of her first target, although she does have free choice for subsequent targets. In particular, if $j = n$ then she has to append an offer herself. Eve now chooses $i_{j+1}$ and does the following:

$$E \rightarrow i_j : (\Pi_{E_2, \mathcal{M}_{j-1}, \mathcal{P}_{j-1}}; \{\gamma_{j-1}\}_{K_{ij}}, \mu_{j-1})$$

$$i_j \rightarrow i_{j+1} : (\Pi_{E_2, \mathcal{M}_{j-1}} \parallel m_j \parallel i_j, \mathcal{P}_{j-1} \parallel S_{ij}^{-1}(\mu_j)),$$

$$\{\gamma_j\}_{K_{ij+1}}, \mu_j$$

$$i_{j+1} \rightarrow E : (\Pi_{E_2, \mathcal{M}_{j-1}} \parallel m_j \parallel i_j \parallel m^* \parallel i_{j+1},$$

$$\mathcal{P}_{j-1} \parallel S_{ij}^{-1}(\mu_j) \parallel S_{ij+1}^{-1}(\mu_{j+1}) ,$$

$$\{\gamma_{j+1}\}_{K_{ij}}, \mu_{j+1}$$

Eve sends her agent first to $i_j$ where it inserts some $m_j$ chosen by her. Then it hops to $i_{j+1}$ (chosen by Eve), inserts some random data $m^*$ (which is discarded later on), and returns to Eve. Eve now updates Alice’s agent as shown below, using the data acquired by her own agent:

$$\langle \Pi_{A, \mathcal{M}_{j-1}} \parallel m_j \parallel i_j, \mathcal{P}_{j-1} \parallel S_{ij}^{-1}(\mu_j) = (\Pi_{A, \mathcal{M}_j, \mathcal{P}_j} \rangle$$

This completes the round. Eve now plans her next move (Eve chooses $i_{j+2}$, she already fixed $i_{j+1}$ in the previous round). In order to send the agent to $i_{j+1}$ she needs to know $\gamma_j$ and $\mu_j$, but she doesn’t — yet. However, Eve knows $\gamma_{j-1}$, $\mu_{j-1}$, and $m_j$. This is sufficient to compute

$$\gamma_j = h(\gamma_{j-1})$$

$$\mu_j = h(m_j; \gamma_{j-1}, \mu_{j-1}, i_{j+1})$$

At this stage, Eve either continues the attack, or she releases Alice’s agent and sends it to $i_{j+1}$, where it resumes normal execution.

$$E \rightarrow i_{j+1} : (\Pi_{A, \mathcal{M}_j, \mathcal{P}_j}; \{\gamma_j\}_{K_{ij+1}}, \mu_j)$$

The underlying weakness of the multihop protocol is the same as in the previously described protocols, namely, the abuse of servers as oracles. The digital signature gives no assurance about the intended recipient of the signed data.
3 The KAG Family of Protocols

Karjoth, Asokan, and Gülcü [5] published a family of protocols which are directed at preserving the integrity and confidentiality of data acquired by free-roaming agents. The general scenario is that of a comparison shopping agent that visits a number of shops, and collects offers from them. The idea behind these protocols is to preserve the integrity of collected offers. Some protocols also address confidentiality of offers.

3.1 Publicly Verifiable Chained Digital Signatures

The Publicly verifiable chained digital signature protocol (P1) is defined as given below:

\[ M_n = \{ (m_n, r_n)_{K_A}, C_n \}_{n=1} \]

\[ C_n = h(M_{n-1}, r_{n+1}) \]

\[ M_0 = \{ (m_0, r_0)_{K_A}, C_0 \}_{n=1} \]

\[ C_0 = h(r_1, i_1) \]

\[ i_n \rightarrow i_{n+1} : \Pi, \{ M_0, \ldots, M_n \} \]

where \( r_{n+1} \) is a dummy offer, \( r_n \) is random salt that makes it harder to attack the encryption. \( C_n \) is called the chaining relation at \( n \). By assumption, it shall be possible to deduce the identity of the signer from a signature [5, pp. 198]. The signer of \( M_0 \) is deemed to be the owner of the agent (unfortunately, the authors of [5] do not explicitly mention from what they conclude who the owner of a given agent is, so we have to do a little guesswork here).

The security of the protocol relies on the assumption that an attacker does not change the last element \( M_n \) in the chain. However, there is no reason why an attacker would be so obliging. On the contrary, if the attacker is willing to build a complete chain for the agent then he can even remove chain elements before his own entry (this contrasts with e.g., the honest prefix property introduced by Yee [9, pp. 267]). The important observation here is that the input to all previous chaining relations is known.

We assume that Eve received an agent owned by Alice. Let Eve be \( i_n, n > 1 \). She picks \( j \) with \( 0 < j < n \) and a new \( i_{j+1} \) of her choice. Please note that there is no free choice of \( i_j \) once \( j \) is fixed, only of \( i_{j+1} \). Eve has to collect an offer from the original shop \( i_j \) for her chosen \( j \) in order to maintain the chaining relation’s validity at \( j - 1 \). Then Eve does the following:

\[ E \rightarrow i_j : \Pi_{E}, \{ M_0, \ldots, M_{j-1} \} \]

\[ i_j \rightarrow i_{j+1} : \Pi_{E}, \{ M_0, \ldots, M_j \} \]

\[ i_{j+1} \rightarrow E : \Pi_{E}, \{ M_0, \ldots, M_{j+1} \} \]

Upon the agent’s return, Eve throws away \( M_{j+1} \), increments \( j \), and picks a new \( i_{j+1} \). The chaining relation and encapsulated offers are build as if Alice’s agent had requested the offer (instead of Eve’s agent with Eve’s program) because \( M_0 \) bears Alice’s signature. Eve repeats the process at her discretion. When she is finally satisfied with the collection
of encapsulated offers she assembled, she pastes them into Alice’s agent, and sends that
agent to $i_{j+1}$. If Alice can be fooled into forwarding agents whose $\mathcal{M}_i$, she signed herself
then Eve’s charade can carry on until the very last (faked) hop. Otherwise, Eve has to
stop her attack before the next to last hop.

It must be stressed here that the problem is not that Eve can truncate offers and grow a
fake stem (this possibility is acknowledged by the authors, so this fact is not surprising).
The problem is that shops can be abused as oracles for generating offers to the terms of
Eve rather than Alice (this remark also holds for sections 3.2 and 3.3). In other words,
Alice might look for blue or red shirts with a preference on blue ones; she might find
out that Eve is the only shop that offers her blue shirts, though. This is possible because
Eve’s agent looks only for red shirts, and the offers made to this agent are returned to
Alice.

3.2 Chained Digital Signatures with Forward Privacy

The second protocol proposed in [5] is the chained digital signature protocol with for-
ward privacy (P2). It is a variation of the protocol discussed in §3.1, with the order of
encryption and signature computation being swapped. The goal of this arrangement is
to hide the identity of shops that provided offers while keeping the integrity assurances.
The protocol is defined as given below:

$$\mathcal{M}_n = \{m_{1-n}, r_n\}_{K_A}, C_n$$
$$C_n = h(\mathcal{M}_{n-1}, r_{n}, i_{n+1})$$
$$i_n \rightarrow i_{n+1} : \Pi; \{\mathcal{M}_0, \ldots, \mathcal{M}_n\}$$

A problem we couldn’t resolve is how a shop knows who the owner of an agent is,
and hence for whom the offers must be encrypted. The shop cannot extract the identity
of Alice from $\mathcal{M}_0$, because the signature of the dummy offer $m_0$ is hidden by the
encipherment. The authors leave that to speculation. The protocol’s description is far
from being sufficiently detailed at this point. Whereas a signer’s identity can be verified
easily against her signature using a public key and corresponding certificate (where the
identity binding is assured by a certification authority), anybody could have used
somebody else’s public key to encrypt data.

Again, we assume that Eve received Alice’s agent, and Eve is $i_n$, as in the previous
attacks. Let $j$ be the smallest number for which Eve knows $i_j$. Eve probably knows $i_{n-1}$
because this is most certainly the host that sent her the agent. In any case she knows $i_n$
(her own identity).

Eve collects arbitrary signed offers using agents of her own, including an offer from
$i_j$. Then, she cuts off the chain at $j$, and appends the offers, starting with the fresh one
collected from $i_j$ and the remaining ones in arbitrary order. In doing so, she generates
random nonces as required, and builds the chaining relations consecutively from known
data. The last chaining relation is computed with the identity of the entity to whom Eve
wants to hand off Alice’s agent.

Upon the agent’s return, Alice cannot decide whether her agent remained unattacked,
or carries offers of shops it has never seen actually. It is worth noting that the integrity
assurance of the protocol relies on the secrecy of the association of $M_j$ with the identity of the shop who signed offer $r_j$. This means that privacy of offers is not only a feature of the protocol, but is also a requirement. In particular, secrecy of the agent’s itinerary is a requirement.

Once again, not the truncation of protocol data is the important point, but Eve’s ability to set the terms for (authentic) offers returned to Alice.

### 3.3 Publicly Verifiable Chained Signatures

Another protocol that is proposed in [5] is the publicly verifiable chained signatures (P4) protocol. The key aspect of the protocol is that each shop generates a temporary asymmetric key pair (either on the fly or by means of pre-computation) to be used by the successor. The public key is certified by the shop that generated the key pair. Each shop uses the private key that it received from its predecessor for signing its partial result, the chaining relation, and the public key to be used by its successor. The private key is destroyed subsequently. Let $(\chi_A, \chi_A^{-1})$ be a temporary key pair generated by $A$. The protocol is as follows:

Oracle

\[ M_0 = \{ (m_0, r_0) \}_K_A, C_0, \chi_{m_0} \}_{\chi_{m_0}^{-1}} \]

\[ C_n = h(M_{n-1}, s_n+1) \]

\[ z_n \rightarrow z_{n+1} : \Pi, \{ M_0, \ldots, M_n \}, \chi_{m_n}^{-1} \]

The protocol is initialized by Alice with:

\[ M_0 = \{ (m_0, r_0) \}_K_A, C_0, \chi_A \}_{S_A^{-1}} \]

\[ C_0 = h(r_0, s_1) \]

It is easy to see that Eve can collect valid certified temporary key pairs from Bob, simply by dispatching and agent of her own to Bob, which promptly returns to Eve. On the agent’s transport to Eve, Bob sends a temporary private key $\chi_B^{-1}$ and corresponding certified public key $\chi_B$ (contained in $M$).

We assume that Eve is $z_n$ and she received Alice’s agent. Let $j$ be the smallest number for which Eve knows $\chi_{z_j}^{-1}$. She received $\chi_{m_j}^{-1}$ with the agent, so at least one such $j$ exists and $j < n$. Eve then cuts off all encapsulated offers following $M_j$, and collects key pairs from all the shops in whose names she wants to fake offers, including shop $i_{j+1}$. Starting with $i_{j+1}$, she appends arbitrary offers, building the protocol data consecutively. The identity that Eve uses in the final chaining relation is the one of the entity to whom she wants to hand off Alice’s agent (for instance Alice herself).

### 4 Protocols Using Secure Co-processors

In [6], Karoth proposes use of trusted secure co-processors as a means to protect mobile agents in a distributed marketplace. The setting equals that described in §3, with the
exception that each shop $i_n$ has a trusted tamper-proof hardware $T_n$ (in brief, its device), which is issued and certified by a central market authority $\mathcal{H}$. The market authority acts as a trusted third party for merchants and customers. By assumption, the channel between a shop and its device is secure against active attacks. Each device has its own asymmetric key pair, and is capable of computing suitable asymmetric ciphers, symmetric ciphers, and message digests. Furthermore, each device has the public key of the market authority, and uses it to authenticate the public keys of other devices.

At the beginning of the protocol, Alice chooses a random $\mathcal{K}$ and sets $C_1 = h(\mathcal{K})$. The protocol continues as follows:

$$i_{n-1} \rightarrow i_n : \Pi A_5\{K_n, C_n\}, \{M_1, \ldots, M_{n-1}\},$$
$$\{C_1, \ldots, C_{n-1}\}$$

$$i_n \rightarrow T_n : \{K_n, C_n\}, \{m_n\}_\mathcal{S}\overline{S}_n, \{K_{T_{n+1}}\}_\mathcal{S}\overline{S}_n,$$
$$T_n : \mathcal{M}_n = \{\{m_n\}_\mathcal{S}\overline{S}_n\}_\mathcal{C}, C_{n+1} = h(M_n, C_n)$$

$$T_n \rightarrow i_{n+1} : \{K_n, C_{n+1}\}, \{C_{n+1}\}_\mathcal{C}, C_{n+1}, \mathcal{M}_n$$

$$i_{n+1} \rightarrow i_n : \Pi A_5\{K_n, C_{n+1}\}, \{M_1, \ldots, M_n\},$$
$$\{C_1, \ldots, C_n\}$$

In the final protocol step, the last shop sends Alice the agent and the final checksum, which is encrypted with $\mathcal{K}$:

$$i_n \rightarrow i_0 : \Pi A_5\{M_1, \ldots, M_n\}, \{C_1, \ldots, C_n\}, \{C_{n+1}\}_\mathcal{C}.$$ 

Alice knows $\mathcal{K}$, so she decrypts $\{C_{n+1}\}_\mathcal{C}$, verifies the checksums consecutively from $C_1$ to $C_{n+1}$, decrypts $M_1, \ldots, M_n$, and finally she verifies the signatures.

We assume that Eve runs a shop in the electronic marketplace, which implies that she has a device certified by the market authority. Consider that Eve received an agent owned by Alice, so Eve is $i_n$. Eve now has a number of encrypted offers, an equal number of checksums, and $\{K_n, C_n\}_\mathcal{K}T_{n}$, which can be decrypted only by her device.

From the protocol, we know that $C_{n+1} = h(M_n, C_n)$. There is nothing secret about $h$, so in fact Eve can take $j$ of the $n-1$ encrypted offers, shuffle them, and re-compute the appropriate checksums herself, beginning with the initial checksum $C_1$ (without ever going through her device). However, Alice expects to receive a matching $\{C_{j+1}\}_\mathcal{K}$ with her agent. Eve cannot encrypt her final checksum with $\mathcal{K}$ because she does not know it – but her device can do it for her! All Eve has to do is passing $C_{j+1}$ in the place where her device expects to receive Eve’s signed offer:

$$E \rightarrow T_n : \{K_n, C_n\}, \{C_{j+1}\}, \{K_{T_{n+1}}\}_\mathcal{S}\overline{S}_n$$

substituted for Eve’s offer

The device first extracts Alice’s secret key $\mathcal{K}$ from $\{K_n, C_n\}_\mathcal{K}T_{n}$, which is encrypted with the device’s public key. Then the device uses $\mathcal{K}$ to encrypt what it thinks is Eve’s signed
offer. Only that it is not the signed offer but the checksum that must be passed back to Alice with her agent.

\[ T_n : \mathcal{M}_n = \{C_{j+1}\}_{j=0}^{n}, C' = h(\{C_{j+1}\}_{j=0}^{n}, C_{j+1}) \]

oracle computation

Eve also passed her own device’s public key rather than that of another shop’s device. What Eve gets back from her device is:

\[ T_n \rightarrow E : \{\mathcal{K}, C_{n+1}\} K_{T_n}, \{C'\}_{j=0}^{n}, C_{n+1}, \{C_{j+1}\}_{j=0}^{n} \]

leaked result

In other words, given a set of signed offers \( \mathcal{M}_1, \ldots, \mathcal{M}_j \) (which are encrypted with Alice’s secret key \( \mathcal{K} \)), Eve can construct a valid representation of Alice’s agent, and return it to Alice in a way that is indistinguishable from an ordinary run of the agent.

Eve can also collect signed offers herself (at her own terms) using agents of her own. For instance, let \( \{m_B\}_{s_B^{-1}} \) be such an offer, collected from Bob. Eve sends this offer to her device, rather than one of her own offers:

\[ E \rightarrow T_n : \{\mathcal{K}, C_{n+1}\} K_{T_n}, \{m_B\}_{s_B^{-1}}, \{K_{T_n}\}_{s_{T_n}}^{-1} \]

Bob’s offer

\[ T_n \rightarrow i_n : \{\mathcal{K}, C_{n+1}\} K_{i_n}, \{C_{n+1}\}_{j=0}^{n}, C_{n+1}, \mathcal{M}_B \]

Bob’s offer encrypted with \( \mathcal{K} \)

The device returns the offer encrypted with \( \mathcal{K} \). Offers prepared in this way can also be used by Eve in her attack on the checksum.

If Eve just wants to append offers that she collected to Alice’s agent (following \( \mathcal{M}_{n-1} \)), then the attack is even simpler. All Eve has to do is passing her own device’s public key to her device rather than that of another shop’s device until she wants to hand off Alice’s agent. In that case she either passes the public key of the next shop’s device, or returns the agent to Alice herself.

In summary, Eve can delete and rearrange any offers brought by the agent, and insert forged offers collected by her, at any position in the chain of results. This means in particular that the protocol does not achieve forward integrity as is claimed by its author. The surprising fact is that although secure co-processors are used, the protocol fails where some software only approaches succeed (for instance the chained MAC protocol [5]). The lesson that is to be learned is that tamper-proof hardware is no guarantee for improved security.

In order to prevent the attack on the final encrypted checksum, the device has to verify that the data input as the signed offer is “well-formed”, in other words, actually constitutes a signature rather than random data. Providing typed driver APIs is not sufficient since the device software itself can be tampered with (which exposes the device’s raw hardware interface).

\[ ^3 \text{In general, Eve knows only } \{\mathcal{K}, C_{n}\}_{K_{i_n}}, \text{ so if she touches any encrypted offers before } n \text{ then she has to hand off the agent herself to Alice, and cannot let another shop do this. However, she can pass on the agent if she knows that it will return to her before it hops back to Alice.} \]
5 Authentication to the Rescue?

It might be argued that mutual authentication of hosts in the course of agent hand-off may inhibit some of the attacks we described. Upon closer inspection, it turns out that actually only one protocol of the ones we discussed may profit from this (although that protocol still remains vulnerable to some extent).

The target state §2.1 does not profit for obvious reasons. The append-only container (§2.2) defines the crucial checksum $C_n$ in a way that makes it impossible for a hop to verify intermediate targeted states. Consequently, Eve can arrange a targeted state in her attack at will, and there is no point for hop $i + 1$ to verify e.g., that the sender of the agent actually inserted element $j$. Neither does the multi-hops protocol (§2.3) benefit from authentication. Eve may always sign $μ_{j−1}^{-1}$ (the last element of $P$) herself, replace the last element of $M$ with her own identity, and complete her attack without raising suspicion.

Protocols §3.2 and §4 obscure or encrypt all protocol data that is passed from one hop to the next. Again, there does not seem to be a hook to improve the protocol’s security by verifying protocol data against authentication results. In protocol P4 (§3.3), hosts are exploited as key-generating oracles. Authentication results can hardly be connected with anything useful either, unless the protocol itself is modified.4

This leaves protocol P1 (§3.1). This protocol has two important properties. First, the data that is added by each host is randomized, and thus cannot be reliably reproduced by means of an oracle exploit. Second, the protocol builds a strong backward chain including the signature of the agent’s previous host. Each host can verify this chain back to $M_1$, starting with the last element in $M$ whose signer must be the authenticated previous hop of the agent.5 This makes it impossible for Eve to hide her traces completely, although she can still launch her attack in one sweep rather than multiple rounds. But her attack must start at her own position in the existing chain, and she must appear as well at the end of her faked sub-chain, because she needs to hand off Alice’s agent and pass the combined authentication and signature check as well.

6 Conclusions

One problem repeatedly occurred in the protocols we analyzed: a legitimate host could be abused by malicious hosts as an oracle that decrypts, signs, or otherwise computes protocol data on behalf of an adversary. These flaws could have been avoided, had the authors of the protocols taken the advice of Needham and Anderson [1] faithfully: “be careful, especially when signing or decrypting data, not to let yourself be used as an oracle by the opponent.”

Mobile agent systems are particularly vulnerable to this type of attack because they are meant to work autonomously, and no human intervention is expected to happen in order to validate and authorize the processing of agents by cryptographic protocols.

\footnote{Each host may certify its temporary key with an authenticated attribute that includes the identity of the agent’s previous hop. However, in that case Eve simply sends her key-collecting agent first to the hop whose identity shall be certified by her next target, then to her target, and back to her.}

\footnote{Due to an unfortunate choice of $C_0$, only Alice can fully verify the chain at $0$.}
Hence, agent servers and agent owners must have means to decide whether protocol data that an agent requests to process or returns, actually belongs to that agent. This brings us to another of Needham’s and Anderson’s rules of good practice: “where the identity of a principal is essential to the meaning of a message, it should be mentioned explicitly in that message.”

None of the protocols that involved signing as a means of authenticating protocol data actually signed a data type or recipient identity along with the data. Hence, protocol data that was collected by one entity appeared valid to other entities as well. Obviously, inclusion of a recipient’s identity is not even enough, because protocol data from one agent instance can be used again in an attack on other agent instances owned by the same entity. Since mobile agents may be under way for a period of time that is hard to anticipate in advance, it is difficult to have a notion of “freshness”. If this were not enough, the protocols also have to cope with multiple agents that run concurrently. Both, agent owners and legitimate hosts must therefore “be sure to distinguish different protocol runs from each other.”

Each agent instance certainly constitutes a different protocol run. On the other hand, digital signatures affixed to an agent’s code are not sufficient to distinguish one agent instance from another. This leads to the important conclusion that digitally signing a mobile agent’s code alone is not sufficient to assert agent ownership.

However, this approach is a favorable one among contemporary mobile agent systems. A signature on code can be copied just like the code itself. Code is written to be re-used, so the agent instance is what renders an agent (a protocol run) distinct. Seen in this light, it is even less desirable to sign credentials that contain a code base rather than the code itself (as described e.g., in [3]), because this gives an adversary potentially more valid agent programs to choose from. Each agent program that is available from a particular code base can be used in conjunction with credentials that refer to the code base.

Instead, the owner of some agent should sign a static kernel, which includes the agent’s code as well as enough redundancy to distinguish between two instances of the same agent. A cryptographic hash value of the kernel’s signature may serve as a unique “anchor” to which protocol data can be bound by means of a digital signature.

Agent developers must still be aware of the fact that “a migrating agent can become malicious by virtue of its state getting corrupted” [10]. We cannot assume that a mobile agent properly represents the intentions of its owner, because – subsequent to its first hop – an agent’s state is a function of its own program and state, and the state and program of the hosts that it visited.

Hence, any attempt to protect a free-roaming agent against interleaving attacks is probably futile unless the agent’s code is carefully designed, such that it does not leak confidential data, and does not enter negotiations based on parameters stored in its mutable state.

Acknowledgments. This paper is a revised and extended version of [11], with a discussion of defensive measures omitted. We thank Günter Karjoth for his comments on the initial manuscript, which helped to improve its precision and focus, as well as for the inspiring discussion that evolved from the commentary. We also thank the anonymous reviewers for their comments, which, again, led to several improvements and extensions.
References

Trust Relationships in a Mobile Agent System

Hock Kim Tan and Luc Moreau*

Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
hkt99r@ecs.soton.ac.uk, L.Moreau@ecs.soton.ac.uk

Abstract. The notion of trust is presented as an important component in a security infrastructure for mobile agents. A trust model that can be used in tackling the aspect of protecting mobile agents from hostile platforms is proposed. We define several trust relationships in our model, and present a trust derivation algorithm that can be used to infer new relationships from existing ones. An example of how such a model can be utilized in a practical system is provided.

1 Introduction

Mobile agent technology has been identified as a new paradigm that allows flexible structuring of distributed computation over wide-scale networks such as the Internet [11]. One of the main concerns currently impeding the wider acceptance and use of mobile agents, particularly in application areas such as e-commerce [6], is the issue of security. Farmer et. al. [4] provides an early discussion of the security problems and requirements unique to mobile agents, as well as the types of security goals that are achievable. A more recent overview of mobile agent security issues, along with a comparative discussion of the current techniques available to address them, can be found in [2], [13] and [8]. In general, we can divide mobile agent security into two broad areas: host security (protecting the host platform from a malicious agent) and code security (protecting the mobile agent from a malicious host platform).

In this paper, we discuss some of the techniques available for addressing the code security issue and suggest that the manner in which current techniques are implemented may not scale well for a security infrastructure that encompasses a large number of highly mobile agents. We identify trust as an important component of a security infrastructure, and develop a security framework for a mobile agent system which incorporates a simple trust model. Our model is motivated by the similarities between the manner in which distributed authentication is handled in a public key infrastructure, and the way code security could be handled in a security infrastructure for mobile agents. Existing work on trust relationships within the context of a public key infrastructure is used as a background to define trust relationships specific to a mobile agent system. We then show

*This research is funded by EPSRC and DERA project "Magnitude" reference GR/N35816.

© Springer-Verlag Berlin Heidelberg 2001
how new trust relationships may be derived from existing ones in our model and present an algorithm to formalize our approach.

The main contributions of this paper are:

- Identifying trust as an important component in a security infrastructure to handle code security;
- Proposing a security framework which incorporates the notion of trust through the delegation of a code security technique;
- Adapting an existing trust model for a public key certificate system to use in conjunction with the proposed security framework.

An overview of the paper is as follows. In Sect. 2, we discuss current code security techniques and suggest the need to incorporate trust. We develop our trust model and security framework through analogies of the use of trust within a public key infrastructure in Sect. 3. Trust relationships within this model is detailed in Sect. 4, while Sect. 5 describes the trust derivation algorithm that we use. Sect. 6 provides an intuitive discussion of how such a model can be deployed in a mobile agent system. Finally, Sect. 7 concludes the paper with a summary and identifies avenues for possible future work.

2 Mobile Code Security Techniques

Host security is a well researched area for which a number of viable techniques have already been developed. These include mechanisms such as sandbox security in the Java programming language [5], software fault isolation [19], proof carrying code [12] and type safe languages [18]. Code security is however more problematic, since this aspect has only come into prominence recently as a security problem unique to mobile code. Most solutions proposed so far have been conceptual, and it is likely that this area will be crucial in determining the future viability of mobile agent applications in scenarios such as distributed e-commerce.

Some of the more well known code security techniques include code obfuscation [7], encrypted functions [14], tamper-proof hardware [20] and execution tracing [17]. The reader is referred to [8,9] for a more thorough overview and classification of the code security techniques currently available. Execution tracing, the technique that we employ in the construction of our security framework, involves the detection of unauthorized modifications of an agent through the faithful recording of the agent’s behavior during its execution on each host platform. This technique requires each host platform involved to create and retain a log or trace of the operations performed by the agent while resident there. Upon return of the mobile agent, the agent owner may (if she suspects that the mobile agent was not correctly executed) request that the various host platforms submit their individual traces. These are then contrasted against a simulated execution of the original mobile agent (using the information contained in the traces) to detect possible deviations in execution of the agent.

All of these techniques attempt to safeguard the mobile agent with regards to one or more security aspects. For purposes of discussion, we provide a simple classification for the security aspects that these techniques might seek to protect.
Trust Relationships in a Mobile Agent System

- **Execution integrity**. This refers to the correct transformation of the current state of the mobile agent to a new state, in accordance with the semantics of the mobile agent code. To accomplish this, it is also necessary to ensure that the correct portion of the code is executed in order to affect the required transformation.

- **State/code integrity**. The state and code of the mobile agent need to be protected from invalid manipulation.

- **State/code visibility**. It may be necessary to permit only certain parts of the state and code to be made visible to the host platform since other parts may contain sensitive information.

- **Resource provision**. It is also important to guarantee the provision of necessary system resources (within the constraints of the resource and security policy of the host platform) to the mobile agent in order for it to execute successfully.

In general, code security techniques only address one or a few security aspects; those that attempt to address every single aspect conceivable are likely to be found deficient in certain aspects upon closer scrutiny. In view of this, it is likely that a future security infrastructure that addresses the code security issue comprehensively will need to incorporate a combination of techniques, rather than a ubiquitous “one-size-fits-all” solution. What is therefore required is a mechanism for selecting the appropriate technique or combination of techniques to use, depending on the execution environment and targeted application.

For example, in his discussion of the tamper-proof hardware approach, Yee [22] suggests the use of trust to negate the requirement for hardware to be installed on all execution environments. It could be permissible to run a mobile agent in a software-only environment (using other purely software based code security techniques), if the deployers of the agent have a certain amount of trust in the that environment. Tamper-proof hardware would only need to be installed in environments whose behaviour or reputation is unknown to the deployers.

Certain code security techniques, such as execution tracing, require active interception on behalf of the agent deployer (the platform deploying the mobile agent will need to verify the execution trace submitted back by the host platform executing the agent). If the deploying platform is the only entity capable of performing such a verification, it will quickly become overwhelmed when the number of mobile agents and corresponding verifications required increase. In order for such a system to scale, the deploying platform must be able to delegate some of its verification activities to other entities in the system. Again, some notion of trust between entities is required for the deploying platform to delegate its activities in this manner.

A more subtle point to be considered is the action of censuring a host platform that has been detected in the act of illegally manipulating some portion of the mobile agent's code or state (i.e. violating state integrity). In most literature describing code security techniques that detect such violations, the assumption is that uniform punitive action is taken towards all perpetrators. On reflection, we see that this inflexibility might not be desirable in every situation. For example, in e-commerce scenarios, it is possible to acquire additional economic interests
or benefits that result from constant interaction with a trusted platform, which may not be readily available from untrusted platforms. Thus when we discover that a trusted platform has violated an agent's integrity, we do not immediately bar our agents from visiting that platform (as might have been the case with an untrusted platform). Instead we could permit migration, but with possibly a more comprehensive code security technique applied. Trust thus provides us with the basis for deciding on a suitable course of action to be taken in dealing with the violation of the agent in the event that such flexibility is advantageous in a given situation.

A large number of agent frameworks, particularly those in e-commerce scenarios, involve the development and evolution of complex trust relationships between the various participating entities. The incorporation of the notion of trust in a security framework allows the development of trust models and metrics to express the nature and flow of trust resulting from the interactions of these entities. Such models and metrics also permit quantitative comparison between different frameworks that may provide useful guidelines on their future development.

We can now identify several points which we believe make a sound argument for the inclusion of the notion of trust as part of an overall security framework when addressing mobile agent code security

- it provides a basis for deciding on the particular code security technique or combination of techniques to be deployed in a particular environment or application;
- it permits the scalability of a system employing certain code security techniques through the delegation of specific security activities;
- it allows flexibility in deciding on the appropriate punitive action to undertake towards perpetrators;
- it allows development of trust models and metrics that express the trust dynamics in e-commerce agent frameworks

The benefits of using incorporating trust and using trust models in a distributed system in general [16] and a mobile agent system in particular [15] have been identified. Certain code security techniques, such as tamper-proof hardware, also incorporate the notion of trust, although in an implicit manner. However, to date, we are not aware of any work that develops a trust model explicitly in the context of a mobile agent system by defining the trust relationships possible in such a system. In the next section, we demonstrate how a simple security framework for a mobile agent system can be developed by adapting the trust model used in a distributed authentication system such as a public key infrastructure.

3 Framework for Code Security in a Mobile Agent System

A public key infrastructure (PKI) [1] is essentially a system that provides all the necessary maintenance activities associated with the complete life cycle of
Trust Relationships in a Mobile Agent System

certificates, which are one of the key elements of a distributed authentication service [10]. The main issue in such a service is ensuring that a public key is correctly associated with the identity of an entity that owns the corresponding private portion. PKI systems involve a trusted third party termed a certificate authority (CA) that is responsible for verifying name-key bindings through the issuance of certificates. On a large scale basis, a single CA would be incapable of handling the name-key binding activity for all users; thus the need for several CAs arises. In such a situation, an end-user may not be able to immediately identify a certificate received, and may require that the certificate be verified in turn by a CA that he or she is familiar with. This in effect creates a certification path through which CAs verify the certificates of other CAs all the way up to a root authority, for which a user would be acquainted with. The certification path thus reflects the propagation of trust between different CAs and users in the system.

We can now begin to develop a security framework for a mobile agent system based on the trust model just described. We do not claim that this model is a definitive one as far as mobile agent systems are concerned; rather it provides a guideline on how a more comprehensive model can be developed. Code security techniques have been classified in literature surveys into detection (execution tracing, state appraisal) and prevention mechanisms (code obfuscation, encrypted functions). Prevention mechanisms seek to prevent meaningful manipulation of agent code and hence are the most reliable, although they are usually very complicated and expensive in deployment. They assume a very simplistic trust model; no entity is trusted at all and maximal measures are undertaken to prevent any possible security breach. Detection mechanisms, on the other hand, are more easily deployable since they merely seek to detect possible violations in the agent. More importantly, when such violations are detected, the nature and severity of the violations allow us to determine the different levels of trustworthiness in the platforms concerned. Based on this consideration, we can select an appropriate combination of code security techniques (in addition to the detection mechanism) that needs to be applied on that particular host platform, in line with the original motivations for the use of trust in a code security framework.

With regards to this, we choose to employ execution tracing as our core code security technique in the framework that we are about to describe. This technique is well developed and to date its only criticisms are related to performance and scalability concerns. There are other detection mechanisms available such as forward integrity [22] and state appraisal [3]; execution tracing however offers the important advantage of being able to detect tampering of any part of the agent as opposed to only specific portions, as is the case with the former two mechanisms. To improve scalability, execution tracing requires the introduction of additional entities to undertake the verification process of execution traces on behalf of the deploying platform (as mentioned in the previous section). In such an instance, these entities assume the role of a trusted third party, not unsimilar to the role of the CA in a PKI. We refer to this trusted third party as a verification server.

In a PKI, the task of associating the public key correctly with an identity (a security requirement for any entity before it can commence utilizing the key),
has essentially been delegated from the entity to the CA. In our system, the verification server functions as an intermediary between an agent owner platform (the platform from which the mobile agent is initially launched from) and a host platform. The task of verifying the correct execution of the mobile agent on the host platform has now been delegated from the agent owner platform to the verification server. In addition, verification servers may delegate execution verification activities to other verification servers in the system, analogous to the manner in which CAs verify certificates of other CAs in a PKI.

Trust is generally established with regards to a specific activity, rather than as an unconstrained notion. For example, it is too general to simply state that an entity A trusts another entity B; it would be more accurate to say that A trusts B with respect to a certain activity. The context of trust used in a PKI is generally with respect to the name-key binding activity (other activities may include secure key pair generation, in the event the CA is responsible for key generation as well). In our system, we employ the classification of security aspects defined in Sect. 2 (execution integrity, state/code integrity, etc) as a context for establishing trust. Since we only utilize the execution trace technique, the two main activities would be execution integrity and state/code integrity.

4 A Trust Model for Mobile Agents

One of the seminal papers to discuss the idea of trust relationships in a distributed authentication system is [21]. There are two types of trust relationships introduced in this paper: direct trust and recommended trust. Direct trust is analogous to the trust obtained between a CA and an entity which generates a public key pair. In this instance, the CA can directly verify the identity of this entity and issue a certificate binding this identity to the entity's public key. Recommended trust is analogous to the trust obtained between an entity that has just received a certificate and the CA that issued it. In this instance, the entity has no way of determining directly that the public key in the certificate is bound correctly to the identity contained within, and trusts the CA to perform this activity for it. A trust derivation algorithm (also presented in [21]) can be used to generate new derived trust relationships from an existing set of direct trust and recommended trust relationships existing in the system.

To achieve a fine grained trust model, we introduce the idea of partitioning a complete mobile agent into several smaller, self-contained state and code components. We believe that in the future, mobile agents will be complex pieces of code composed by their deployers from reusable components that are distributed by third party code producers. In this instance, we can selectively apply different code security techniques to different state and code components, and thus establish trust relationships of different contexts with respect to different code and state components. Of course in our model, we only employ the execution trace technique, which we can still apply selectively to different code and state components.
4.1 Trust and Belief Relationships in a Mobile Agent System

Before defining the trust and belief relationships in our system, we first define a state space encompassing all the relevant entities:

\[
\begin{align*}
\mathbb{OP} &= \{ A, B, \ldots \} & \text{(Set of agent owner platforms)} \\
\mathbb{VSP} &= \{ V_{S_0}, V_{S_1}, \ldots \} & \text{(Set of verification servers)} \\
\mathbb{HP} &= \{ H_0, H_1, \ldots \} & \text{(Set of host platforms)} \\
\mathbb{SC} &= \{ s_0, s_1, \ldots \} & \text{(Set of code/state components)} \\
\mathbb{SO} &= \{ s_0, s_1, \ldots \} & \text{(Set of security objectives)} \\
\end{align*}
\]

We assume that all mobile agents in the system can be composed from a combination of predefined set of code and state components made available by third party code producers. Agent owner platforms are platforms where mobile agents are launched from. These agents will migrate through an itinerary of host platforms before terminating or returning to their respective agent owner platforms. Execution tracing of these mobile agents is performed by a set of verification servers distributed throughout the system. Trust and belief relationships between entities in the system are established with respect to a certain type or class of activities; this corresponds to the classification of security aspects mentioned earlier.

(1) \( V_{S_0} \ \text{trusts.exe} \ \ H_0 \ \text{with} \ (X,S) \) \hspace{1cm} \text{(Server-host trust)}
(2) \( A \ \text{trusts.exe} \ H_0 \ \text{with} \ (X,S) \) \hspace{1cm} \text{(Owner-host trust)}
(3) \( A \ \text{trusts.exe} \ V_{S_0} \ \text{with} \ (X,S) \) \hspace{1cm} \text{(Owner-server trust)}
(4) \( V_{S_0} \ \text{trusts.exe} \ V_{S_1} \ \text{with} \ (X,S) \) \hspace{1cm} \text{(Server-server trust)}
(5) \( A \ \text{believes.exe} \ H_0 \ \text{with} \ (X,S) \) \hspace{1cm} \text{(Owner-host belief)}
(6) \( A \ \text{believes.exe} \ V_{S_0} \ \text{with} \ (X,S) \) \hspace{1cm} \text{(Owner-server belief)}

where \( A \in \mathbb{OP}, X \in \mathbb{SO}, S \subseteq \mathbb{SC}, V_{S_0}, V_{S_1} \in \mathbb{VSP}, H_0 \in \mathbb{HP} \)

\textbf{Fig. 1. Basic trust and belief relationships}

We now describe the basic trust and belief relationships (Figure 1) that are possible between the three different types of entities in the system (host platforms, agent owner platforms and verification servers). The first relationship, server-host trust, represents the trust that a verification server, \( V_{S_0} \), has in a host platform, \( H_0 \), to undertake the transformation of the state components specified in \( S \) correctly with respect to a security objective \( X \). \( S \) and \( X \) are thus constraints on the context for which this relationship is applicable. We give this type of trust relationship the term execution trust. A server-host trust relationship is initially established when a verification server successfully validates the execution trace submitted by a host platform. This execution trace pertains to a mobile agent composed of the components \( S \), undertaken by the verification server to test the reliability of the host platform in question.

Owner-host relationship represents the trust an agent owner platform, \( A \), has in a host platform, \( H_0 \), to undertake the transformation of the code/state components specified in \( S \) correctly with respect to a security objective \( X \). Since
verification servers are the only entities in our system possessing the functionality
necessary to ascertain execution correctness, this type of relationship cannot be
established directly by an agent owner platform. It can only be derived from
other existing trust relationships, as will be demonstrated later.

Owner-server trust relationship refers to when an agent owner platform, $A$, trusts a verification server, $VS_0$, to undertake correct verification of the transfor-
mation of state components specified in $S$, with respect to a security objective $X$. This trust relationship is analogous to the idea of recommendation or rec-
commended trust as described in [21]. In our context, we shall give it the term
verification trust. Verification trust is established directly when an agent owner
platform decides to delegate the task of verifying correct execution of its mobile
agents to a verification server. Once such a relationship is in place, the agent
owner platform will supply the verification server with the necessary information
and resources (for example, a copy of all mobile agents launched by the agent
owner) to undertake the verification successfully.

Server-server trust relationship is interpreted to mean that a verification
server, $VS_0$, trusts another verification server, $VS_1$, to undertake verification of
the correct transformation of state components, specified in $S$, with respect to a
security objective $X$. This could result from a verification server delegating the
responsibility of verifying certain host platforms to other verification servers in
the system, and is thus a relationship that can be established directly.

All of these trust relationships, with the exception of owner-host trust, express
trust on the basis of explicit actions (or the results of these actions) undertaken by entities involved in the relationship. Sect. 6 elaborates further on
how these relationships are initially established and how they might evolve over
a period of time. We also require another type of relationship to express the as-
sumptions and/or beliefs that an entity has about another entity in the system.
For example, an agent owner platform, $A$, may have reason to believe (based on
knowledge acquired from an external source) that host platform $H_{a_3}$ is capable
of executing $S$ correctly with respect to $X$. $A$ is not capable of directly verifying
the accuracy of this belief (since only verification servers possess the function-
ality necessary to verify execution traces from host platforms). This belief is
thus expressed in the form of a owner-host belief relationship (statement 5). The
idea of a server-host and a server-server belief relationship is equally valid in this
context; they are not included in order to simplify the trust derivation algorithm
that we develop in the next section.

It is important to note that trust and belief relationships are not sym-
metric in our system (i.e. $VS_0 \text{ trusts } \text{ ver } VS_1$ does not necessarily imply that
$VS_1 \text{ trusts } \text{ ver } VS_0$). Also, we did not introduce the idea of trust originating
from a host platform (i.e. the host platform is the terminating point for an execu-
tion trust relationship). This could be useful if we wish to extend our model to
encompass the issue of host security (i.e. the host needs to be able to trust that
agent owner platforms do not dispatch malicious agents to its environment), but
we do not address this here.

In the next section, we explore how we can combine these basic trust and
belief relationships to produce new trust and belief relationships and present a
trust derivation algorithm to demonstrate our approach.
5 Deriving New Trust Relationships

The different ways in which new trust and belief relationships can be formed from existing ones is illustrated in Figure 2. For the server-host and server-server trust derivations, the intersection of the code/state component constraint sets $S_1$ and $S_2$ in the derived relationship, indicates that the derived relationship should be in the context of the code/state components common to both relationships that it is derived from. Owner-host and owner-server trust derivations will, in addition, involve owner-host and owner-server belief relationships as well. In this case, we only derive a new trust relationship if there already exists a belief relationship between the agent owner platform and the host platform or verification server concerned. Again, this new relationship will be in context of the code/state components common to both the two initial relationships as well as the belief relationship $(S_1 \cap S_2 \cap S_3)$.

Note that we do not include the idea of inferring new trust or belief relationships by using an existing belief relationship as a starting point. This is due to the fact that while trusting behaviour is transitive (resulting for example, from delegation of the verification activity in the case of server-server or server-host trust derivations), trusting belief is not. However the context of an existing belief relationship can be altered independently by an agent owner platform depending on the results of the new trust relationships derived. For example, if in deriving a new owner-host trust relationship, $S_1$ and $S_2$ are both supersets of $S_3$, then the agent owner platform could choose to expand the constraints of its current belief relationship with the host platform from $S_3$ to $S_1 \cup S_2$ instead. This ensures that the next time a trust relationship is derived from the existing owner-server and host-platform relationships, a wider constraint can be achieved.

5.1 Verification Path

It is important to note the exact sequence in which the existing relationships are combined to provide a context for the new trust relationships obtained. To achieve this, the idea of a verification path is introduced. A verification path refers to a sequence of entities (verification servers, agent owner platforms or host platforms) involved in the derivation of new trust relationships. Consider for example, the following trust relationship statements

(a) $V_{S_0}$ trusts over $V_{S_1}$ with...
(b) $V_{S_1}$ trusts over $V_{S_2}$ with ...
(c) $V_{S_2}$ trusts exe $H_0$ with ...

(a) and (b) can be combined to obtain a new trust relationship:
(d) $V_{S_2}$ trusts over $V_{S_2}$ with ...

The verification path at this stage involves the entities $V_{S_0}$, $V_{S_1}$ and $V_{S_2}$ in that given sequence. d) and c) can subsequently be combined to obtain a new trust relationship:

(e) $V_{S_0}$ trusts exe $H_0$ with ...

The verification path sequence now involves the entities $V_{S_0}$, $V_{S_1}$, $V_{S_2}$ and $H_0$. Further derivation of new trust relationships from (e) will involve expanding the verification path in a similar manner. The idea of a verification path will be used in the algorithm that we develop next.
Deriving server-host trust

if there exists relationships of the form
  a) $V S_1 \text{trusts}. \text{exe} H_0$ with ($X, S_1$)  
  b) $V S_0 \text{trusts}. \text{ver} V S_1$ with ($X, S_2$)
then we can infer a new relationship of the form
  $V S_0 \text{trusts}. \text{exe} H_0$ with ($X, S_1 \cap S_2$)

Deriving server-server trust

if there exists relationships of the form
  a) $V S_1 \text{trusts}. \text{ver} V S_1$ with ($X, S_1$)  
  b) $V S_1 \text{trusts}. \text{ver} V S_2$ with ($X, S_2$)
then we can infer a new relationship of the form
  $V S_1 \text{trusts}. \text{ver} V S_2$ with ($X, S_1 \cap S_2$)

Deriving owner-host trust

if there exists relationships of the form
  a) $V S_1 \text{trusts}. \text{exe} H_0$ with ($X, S_1$)  
  b) $A \text{trusts}. \text{ver} V S_0$ with ($X, S_2$)
and if there exists a belief relationship of the form
  c) $A \text{believes}. \text{exe} H_0$ with ($X, S_3$)
then we can infer a new trust relationship of the form
  $A \text{trusts}. \text{exe} H_0$ with ($X, S_1 \cap S_2 \cap S_3$)

Deriving owner-server trust

if there exists relationships of the form
  a) $A \text{trusts}. \text{ver} V S_0$ with ($X, S_1$)  
  b) $V S_0 \text{trusts}. \text{ver} V S_1$ with ($X, S_2$)
and if there exists a belief relationship of the form
  c) $A \text{believes}. \text{ver} V S_1$ with ($X, S_3$)
then we can infer a new trust relationship of the form
  $A \text{trusts}. \text{ver} V S_1$ with ($X, S_1 \cap S_2 \cap S_3$)

Fig. 2. Deriving new trust relationships

5.2 Trust Derivation Algorithm

We can now detail a trust derivation algorithm (Figure 3), which we extend from the one presented in [21] by incorporating the idea of using beliefs in the process of deriving new trust relationships. The algorithm demonstrates how the derivations just explained can be systematically applied in a system described by an initial set of trust relationships. The goal of this algorithm is to generate from this initial set of trust relationship expressions, a set of tuples $\mathcal{HS}$, that describe owner-host trust relationships that exist between any given agent owner platform, $A$, in the system and all other host platforms in the system. The algorithm works on the elements within two sets, $\mathcal{HS}$ and $\mathcal{N}$.

$\mathcal{HS}$ is a set of tuples, with each tuple consisting of a host platform, $H$, as well as a set of state components, $C$. Initially, $\mathcal{HS}$ is empty and the algorithm will append elements to it during its execution. At the termination of the algorithm,
each tuple in $\mathcal{H} \mathcal{S}$ represents a new host platform, $H$, in which the agent owner platform $A$ can establish a new derived owner-host trust relationship with the form $A \text{ trusts } \text{ex} \ H$ with $(X, C)$.

$\mathcal{N}$ is also a set of tuples, each tuple representing a possible next step in a verification path. The constituent components of each tuple are:

- a verification server, $VS_i$, which is the next possible entity in a verification trust path;
- a sequence $seq = [A, VS_j, VS_k, ..., VS_l]$ which represents the sequence of the verification path traversed so far;
- a set of code/state components $Sc$ which represents the code/state components for which trust is still applicable on the given verification path.

The expression $seq \bullet VS_j$ is used to indicate that a new entity $VS_j$ is being appended to a sequence $seq$ of a verification path. At the start, $\mathcal{N}$ is initialized with the tuples that correspond to all initial trust relationships of the form $A \text{ trusts } \text{ver} \ VS_i$ with $(X, Sc_j)$, where $A$ is the current agent owner platform to which the algorithm is being applied to.

### 5.3 Trust Derivation Algorithm - Example

Consider a system consisting of a set of host platforms, $\mathcal{H} \mathcal{E}$, verification servers, $\mathcal{V} \mathcal{E}$, code/state components $\mathcal{S} \mathcal{E}$ and a single agent owner platform, $A$.

\[
\begin{align*}
\mathcal{V} \mathcal{E} &= \{VS_0, VS_1, VS_2, VS_3\} \\
\mathcal{H} \mathcal{E} &= \{H_p, H_q\} \\
\mathcal{S} \mathcal{E} &= \{s_0, s_1, s_2, s_3, s_4\}
\end{align*}
\]

At the start, we assume that the system already has the following initial trust and belief relationships:

1. $A \text{ trusts } \text{ver} \ VS_0$ with $(X, SE)$
2. $A \text{ trusts } \text{ver} \ VS_1$ with $(X, SE)$
3. $A \text{ trusts } \text{ver} \ VS_2$ with $(X, SE)$
4. $A \text{ believes-} \text{ver} \ VS_0$ with $(X, \{s_1, s_2\})$
5. $A \text{ believes } \text{ex} \ H_p$ with $(X, \{s_2, s_3\})$
6. $VS_0 \text{ trusts } \text{ver} \ VS_i$ with $(X, \{s_0, s_1, s_2\})$
7. $VS_1 \text{ trusts } \text{ex} \ H_p$ with $(X, \{s_1, s_2\})$
8. $VS_2 \text{ trusts } \text{ver} \ VS_i$ with $(X, \{s_1, s_4\})$
9. $VS_3 \text{ trusts } \text{ex} \ H_p$ with $(X, \{s_3, s_4\})$
10. $VS_4 \text{ trusts } \text{ex} \ H_p$ with $(X, \{s_3, s_4\})$

We now proceed to apply the algorithm to $A$ to determine all the new trust relationships that can be derived with the host platforms in the system. We start by initializing $\mathcal{N}$ with all the trust relationships that originate from the agent owner platform $A$ (1, 2 and 3).

\[
\mathcal{N} = \{VS_0, [A, VS_1, \mathcal{S} \mathcal{E}], VS_1, [A, VS_2, \mathcal{S} \mathcal{E}], VS_2, [A, VS_4, \mathcal{S} \mathcal{E}]\}
\]

\[
\mathcal{H} \mathcal{S} = \{}
\]
Signature
\[ C = \mathbb{P}(SC) \]
\[ HostTuple = H \times C \]
\[ HS = \mathbb{P}(HostTuple) \]
\[ HS ::= \{ (H_1, C_1), (H_2, C_2), \ldots \} \]
\[ seq = Ownerplatform \times VS \times \ldots \times VS \]
\[ PathTuple = VS \times seq \times SC \]
\[ N = \mathbb{P}(PathTuple) \]
\[ N ::= \{ VS_1, seq_1, SC_1 \}, \{ VS_n, seq_2, SC_2 \}, \ldots \]

Initialisation
\[ TBS = \{ \text{Set of initial trust and belief relationships} \} \]
\[ HS = \{ \} \]
\[ N = \{ VS_1[A, VS_1, SC_1], VS_2[A, VS_2, SC_2], \ldots, VS_j[A, VS_j, SC_j] \} \]
where \( A \) is the current agent owner platform to which the algorithm is being applied and \( VS_1, VS_2, \ldots, VS_j \) are all the verification servers for all trust relationships \( A \text{ trusts } VS \) with \( (X, SC) \in TBS \)

boolean foundtuple;
Do until \( N = \emptyset \):

Select a PathTuple \( \langle VS_i, seq_i, SC_i \rangle \) from \( N \)

for every \( VS_i \text{ trusts } ex \ H_k \text{ with } (X, S_i) \in TBS \)

if \( A \text{ believes } ex \) \( H_k \text{ with } (X, S_n) \in TBS \)

begin

foundtuple = false

for every HostTuple \( \langle H_j, C_j \rangle \) in \( HS \)

if \( H_k = H_j \)

\[ C_j := C_j \cup (S \cap S_m \cap SC) \], foundtuple = true

if not foundtuple

\[ HS ::= HS \cup \langle H_k, (S \cap S_m \cap SC) \rangle \]

end

end

for every \( VS_i \text{ trusts } ex \ VS_n \text{ with } (X, S_i) \in TBS \)

if \( A \text{ believes } ex \ VS_n \text{ with } (X, S_i) \in TBS \)

if \( VS_n \notin seq \)

\[ N ::= N \cup \{ VS_n, seq \text{\bullet } VS_m[SC \cap S_n \cap SC] \} \]

end

\[ N ::= N \setminus \{ VS_k, seq, SC \} \]

Fig. 3. Trust derivation algorithm
After the first pass of the algorithm, we have
\[ \mathcal{N} = \{ \langle V_S, [A, V_S], \{ s_1, s_2 \} \rangle, \langle V_S, [A, V_S], SE \rangle \} \]
\[ \mathcal{H} = \{ \} \]
After the second pass of the algorithm, we have
\[ \mathcal{N} = \{ \langle V_S, [A, V_S], SE \rangle, \langle V_S, [A, V_S], SE \rangle \} \]
\[ \mathcal{H} = \{ \{H_\mathcal{R}(s_2)\}\} \]
After the third pass of the algorithm, we have
\[ \mathcal{N} = \{ \langle V_S, [A, V_S], SE \rangle \} \]
\[ \mathcal{H} = \{ \{H_\mathcal{R}(s_2)\}\} \]
After the fourth pass of the algorithm, we have
\[ \mathcal{N} = \{ \} \]
\[ \mathcal{H} = \{ \{H_\mathcal{R}(s_2, s_3)\}\} \]
Thus we can form a new trust relationship of the form
\[ A \text{ trusts } H_p \text{ with } \langle \mathcal{H}_3(s_2, s_3) \rangle \]

6 Deploying the Framework

We discuss intuitively how the proposed framework could be deployed in a mobile agent system. Consider a community of host platforms, verification servers and agent owner platforms with trust relationships already established among themselves. A new agent owner platform that wishes to participate in the community will need to establish trust relationships with one or more verification servers. Through a short interaction with a selected verification server, the agent owner platform could determine the host platforms that the server in question has a trust relationship with. The agent owner platform can initially ascertain the reliability of that verification server by composing a mobile agent and launching it to a host platform, with the execution trace being submitted back to both the agent owner platform and the verification server. The results reported back by the verification server are checked for consistency with the validation of the trace by the agent owner platform.

Once the agent owner platform is satisfied, it establishes a trust relationship with the verification server (with respect to specific components) and executes the trust derivation algorithm to obtain new trust relationships with other host platforms. This provides it with a potential itinerary for future mobile agents that it wishes to launch (in the event that the agent owner platform supplies a predefined itinerary), or useful information that can be embedded in the agent itself (should the agent be capable of dynamically deciding its itinerary while it migrates). New verification servers that join the community can establish trust relationships with existing verification servers in a similar manner. New host platforms on the other hand, could advertise their presence through a registry service after which they can be tested for reliability in hosting mobile agents by verification servers who express interest in establishing trust relationships with them.

The key to the evolution of the trust relationships in this framework is the verification of an execution trace submitted by a host platform to a verification
server. Trust relationships are initially established as described above, and remain static as long as all traces checked are valid. The moment a verification server detects an invalid trace, the nature of its current trust relationships with the offending host platform will be altered. This could range over several possible alternatives: severing all existing trust relationships, severing some trust relationships or degrading existing relationship(s) by reducing the number of components that the relationship(s) is valid for.

Returning to the analogy that we introduce at the start of Sect. 3 (i.e. the verification server being roughly equivalent to a CA in a PKI), we note that the CA is trusted to maintain the integrity of the key-to-name binding within a certificate. Certificate revocation is employed when such integrity becomes suspect (for example, due to a suspected key compromise before the expiry of the certificate). This is typically implemented using a periodic publication mechanism such as certificate revocation lists (CRLs), which can be accessed by other entities that need to validate certificates. A verification server, on the other hand, is trusted to maintain the integrity of mobile agent execution, and alters its trust relationship (degradation or destruction) with the offending platform when a violation of this integrity is detected. Information about this relationship change (trust information) is then propagated to all other verification servers or agent owner platforms (the trustors) that have established trust relationships with the server that detected the violation (the trustee). This will in turn result in a corresponding change in trust relationships on those servers and platforms as well. In effect, trust information is equivalent to a CRL in a PKI, with the difference that trust information is propagated instead of being published. Agent owner platforms could use the event of reception of trust information as a trigger to execute the trust derivation algorithm again in order to recalculate their new trust relationships with existing host platforms.

The actual mapping between the detection of a violation in an execution trace and the subsequent change affected in a trust relationship (destruction or degradation) is a function of a security policy which can be either administered locally or globally administered. The discussion of such policies and their implication on the trust dynamics of the system as a whole is beyond the scope of this paper, but remains important work to be accomplished in studying the effects of a trust model in a mobile agent security framework.

7 Conclusion

This paper proposes the incorporation of a trust model as part of a security framework for mobile agents. We argue that the notion of trust can aid in a more flexible and scalable deployment of existing code security techniques. We also suggest that the manner in which trust would be employed in a wide scale security infrastructure for mobile agents has many parallels to the way it is used currently in a distributed authentication system such as a public key infrastructure. Based on this motivation, we propose a simple security framework for a mobile agent system that resembles the structure of a public key infrastructure. Drawing from existing work on trust relationships in such an infrastructure, we define several trust relationships for a mobile agent system. We then demonstrate
how new trust relationships could be derived from existing ones, and present an
algorithm to formalize our approach.

We believe that the material developed here is representative of the initial
work required in the construction of a complete trust model for a mobile agent
system. Such a model would permit a detailed insight into the complex inter-
actions that involve trust in a mobile agent system. Future work in this direc-
tion could involve a more precise and formal definition of trust relationships (includ-
ing, for example, explicit negative trust relationships) specific to mobile agents.
There will also be a need to investigate how execution tracing (and other existing
code security techniques) could be modified to fit effectively within the structure
of such a framework.

References

1. C. Adams and S. Lyold. Understanding Public Key Infrastructure: Concepts,
3. W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Authen-
tication and state appraisal. In European Symposium on Research in Computer
   Security, number 1146 in LNCS. Springer-Verlag, 1996.
4. W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Issues and re-
   quirements. In Computer Communications, Special Issue on Advances in Research
   and Application of Network Security, October 1996.
5. L. Gong. Java Security Architecture (JDK1.2). Technical report, Sun Microsys-
tems, March 1998.
7. F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious
   hosts. In Mobile Agents and Security, number 1419 in LNCS. Springer-Verlag,
   1998.
tions, Special Issue on Advances in Research and Application of Network Security,
   November 2000.
10. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed
    systems: Theory and practice. ACM Transactions on Computer Systems, 10(4),
11. D. B. Lange and M. Oshima. Seven good reasons for mobile agents. Communica-
12. G. Necula and P. Lee. Safe kernel extensions without run-time checking. In Pro-
    ceedings of the 2nd Symposium on Operating System Design and Implementation
    (OSDI '96), Washington, October 1996.
13. R. Opplieger. Security issues related to mobile code and agent-based systems. Com-
    puter Communications, 22(12), July 1999.
Evaluating the Security of Three Java-Based Mobile Agent Systems

Sebastian Fischmeister, Giovanni Vigna, and Richard A. Kemmerer

Department of Computer Science
University of California Santa Barbara
{sfischme,vigna,kem}@cs.ucsb.edu

Abstract. The goal of mobile agent systems is to provide a distributed computing infrastructure supporting applications whose components can move between different execution environments. The design and implementation of mechanisms to relocate computations requires a careful assessment of security issues. If these issues are not addressed properly, mobile agent technology cannot be used to implement real-world applications. This paper describes the initial steps of a research effort to design and implement security middleware for mobile code systems in general and mobile agent systems in particular. This initial phase focused on understanding and evaluating the security mechanisms of existing mobile agent systems. The evaluation was performed by deploying several mobile agents systems in a testbed network, implementing attacks on the systems, and evaluating the results. The long term goal for this research is to develop guidelines for the security analysis of mobile agent systems and to determine if existing systems provide the security abstractions and mechanisms needed to develop real-world applications.

Keywords: Mobile agent systems, computer security, security testing.

1 Introduction

Recently mobile code has attracted a great deal of interest from both industry and academia. The ability to dynamically deploy application components across the network is a powerful mechanism to improve the flexibility and customizability of applications.

Mobile code is a general concept that encompasses a number of different approaches to reconfigure the location of the components of a distributed application [7]. The most common form of code mobility is code on demand, which is the download of executable content in a client environment as the result of a client request to a server. A well-known example of this approach is the download of Java applets or Javascript code in a WWW browser. A different form of code mobility is represented by the upload of code to a server. The uploaded code is executed by the server and possibly the results of the computation are sent back to the client. This form of mobility, also known as remote evaluation [13], allows the client to execute a computation close to the resources located at the server’s side so that network interaction can be reduced. Common examples are represented by the use of SQL to perform queries on a remote database or the upload of PostScript code to a remote printer. A third form of code mobility is represented by the mobile
agent paradigm. In this case, mobile components can explicitly relocate themselves across the network, usually preserving their execution state (or part thereof) across migrations. Examples of systems supporting this type of mobility are Telescript [17] and D’Agents [8].

Past research on mobile code security has mainly focused on code on demand and remote evaluation [6]. These forms of mobility are easier to deal with because they encompass a single interaction in the transfer of a code component. Some of the results achieved in these areas have been applied to the mobile agent approach, but the problem of creating a distributed computing infrastructure where agent-based applications belonging to different (usually untrusted) users can execute concurrently has not been solved yet [15]. In most cases, mobile agent systems (MASs) are proof-of-concept prototypes whose focus is on sophisticated mobility mechanisms; security is left as future work. Other systems provide some basic security mechanisms and primitive support for the definition of security policies, but the provided mechanisms are far from being a sound, comprehensive security solution. If the security problem is not solved in a reliable way, the applicability of mobile agent technology in the real world will be impossible.

This paper describes the first steps of a research effort aimed at the development of secure mobile agent systems. As a preliminary phase in this research effort, it was decided to assess the security provided by existing MASs. In this phase a number of MASs that provide security mechanisms were installed on a testbed network in the Reliable Software Lab at UCSB. The network is composed of hosts running various operating systems, such as Sun Solaris 2.x running Sun’s reference implementation of the Java Development Kit (JDK) 1.1.8, Linux 2.x running the JDK 1.1.7, and Microsoft’s Windows NT 4.0 with JDK 1.1.7. Attacks were launched against the MASs under exam, and the results were analyzed.

The results of the security analysis for a subset of the MASs that were collected and installed are presented in this paper. The subset includes Aglets SDK 1.1, Jumping Beans 1.1, and Grasshopper 1.2.2.3. The remainder of this paper is organized as follows. Section 2 presents some terminology by describing an abstract mobile agent system, and it also reviews some basic security terminology. Sections 3, 4, and 5 present the results of instantiating attacks against the authorization mechanisms of the systems under analysis. Section 6 draws some conclusions and outlines future work.

2 General Framework for the Analysis

Before discussing each of the mobile agent systems it is important to define some common concepts, abstractions, and terminology. This framework will then be used to define some general attack classes, which can then be instantiated on particular systems. The definitions presented in this section were obtained by the analysis of a number of existing systems and their security models [8,9,12,14], as well as by the OMG’s MASIF specification [10].

An abstract mobile agent system is shown in Figure 1. The main components are mobile agents, places, agent systems, regions, and principals. A mobile agent is a computational unit and it consists of a code space, an execution state, and a data space. The code space contains a set of references to code fragments that can be invoked during the
execution of the agent. The code space includes both references to fragments that are owned by the agent (e.g., the code that specifies the behavior of the application component implemented by the agent) and references to external classes that can be part of a place, system, or region (e.g., the code of procedures that implement system services). The execution state contains all the information related to the evolution of the agent, e.g., the execution stack, the code fragment currently being executed, and the program counter. The data space contains references to external resources that can be accessed by the agent, e.g., a reference to an open file.

The execution of agents is supported by places. Each place provides a local infrastructure to a visiting mobile agent. The place infrastructure supports the execution of particular procedures as defined in the associated code repository and provides access to local resources (e.g., a database or a local printer). Access to local resources is regulated by the place’s security system. The security system comprises three subsystems whose tasks are authentication, authorization, and accounting. Each subsystem contains a policy that specifies how the security functionality is configured (e.g., the CPU quotas to be assigned to incoming agents in the case of the accounting subsystems), a set of security resources that represent dynamic information about the state of the system, (e.g., the current credentials for a visiting agent in the case of the authentication system), and a code repository that contains the definition of the procedures used to implement the security subsystem mechanisms.

Several places may be grouped within an agent system. Places inside an agent system may share resources, code, or security mechanisms and, in general, have a privileged relationship with each other. Moving an agent between places in the same agent system

Fig. 1. A mobile agent system.
and interaction among agents within the same agent system is considered less expensive than interaction or mobility between different agent systems. Usually an agent system is implemented on a single host. An agent system has a structure that is similar to the structure of a place. Its resources, code repository, and security system are shared by the contained places. For example, the authentication system of a place may define its authentication procedures on the basis of those defined at the agent system level.

Agent systems may be grouped in *regions*. A region represents a security domain where network-wide resources are accessed following a uniform policy. Like places, and agent systems, a region is defined in terms of code repository, resources, and security systems. For example the accessible nodes within the region may be specified as resources at the region level. As another example, role-based access control policies may be specified at the region level and then enforced locally by the agent systems.

Agents, places, systems, and regions are associated with a number of *principals* that represent real-world entities such as a person, an organization, or a company. Principals are responsible for the definition or the actions of a specific component of a region (e.g., see [9]). Principals may be associated with particular tasks or responsibilities and their definition may span a place, a system, or a region. For example, a principal may be responsible for the definition of the code fragments used to check the identity of a moving agent inside a region, or it may identify the owner of a resource available at a place.

Traditionally security mechanisms have been classified into authentication mechanisms, authorization mechanisms, and accounting (or resource control) mechanisms. Authentication mechanisms determine who the principal(s) associated with a particular component in a system is (are). Authorization mechanisms determine the acceptable actions of a component on the basis of its associated principal, as determined by the authentication process. The set of possible actions is specified by a policy that given a subject, an object, and the action to be performed, specifies if the requested access should be granted or not. Accounting mechanisms regulate the amount of resources that can be accessed by a component and may be used as a basis for billing procedures. In this paper the analysis is limited to authorization mechanisms.

Authorization mechanisms are analyzed by means of an *access matrix*. Intuitively, the access matrix helps to determine what the possible *access space* for a component is: that is, what other components in the model can be accessed, e.g., by means of an object reference or a file descriptor. The access matrix contains rows and columns labeled with the components of the model. Each cell in the matrix holds the type of access that the component referenced in the corresponding row is allowed for the component referenced in the column. The type of access can be *direct, indirect, or non-existent*. Direct access implies that access can be performed through a direct reference, e.g., through an object reference. An indirect reference specifies that access to the object is implicit by means of a system/subsystem relation or some other association. For example, the execution state of an agent may be indirectly accessible by the agent itself, even though the agent has no means to access a representation of its stack directly. This could be accomplished, for example, by having the agent access it indirectly by means of diagnostic and exception handling routines.
An access matrix can be local, remote, or external. A local access matrix describes access to elements in the same place or system. A remote access matrix specifies access to components in different systems of the same region. An external matrix describes the case where access can cross protection domain boundaries.

The analysis of a system is performed by analyzing the different access matrices and filling in the types of access allowed between components implemented in the particular system. For each possible access, one must determine what are the possible operations and what subset of these operations would actually be permitted. Then each operation is exercised and the outcome is verified against the defined policy.

In the following, we present the results of analyzing the authorization security for three Java-Based systems. All three of these systems use access control lists (ACLs) to implement the access matrix.

3 Aglets

The Aglets Software Development Kit [2] (Aglets SDK) is a Java-based mobile agent system developed by IBM Tokyo Research Laboratory in Japan. The version analyzed and evaluated in this paper is the beta version, 1.1 beta2. Recently, Aglets became an open source project. Its current release is 2.0b.

3.1 The Aglets Model

In the Aglets SDK mobile agents are called “aglets”. The code space of an aglet contains a set of private Java classes (the implementation of the aglet) and references to classes in the runtime system. Aglets are implemented as threads in a Java Virtual Machine and their execution state is represented by the thread’s stack and the corresponding program counter. The data space of an aglet contains references to system resources (e.g., sockets and files) and references to other aglets or to local objects that act as wrappers to provide access to particular resources (e.g., a database). Although the Aglets model does distinguish between places and agent systems, the software that is shipped with the system does not support multiple contexts. A single place inside a single agent system is mapped to a component called the “Tahiti” server. Regions are not present in the Aglets’ systems. The mapping from our abstract model to Aglets is shown in Table 1.

The Tahiti server supports agent execution, provides mechanisms for agent mobility, and implements the security mechanisms. The code repository for the Tahiti component is a set of Java classes that implement the runtime system. Local resources are implemented as stationary agents or object wrappers. The Aglets agent system provides a simple authentication subsystem based on host identifiers and no accounting or resource control system is provided. Authorization is enforced by an implementation of the Java Security Manager interface. The Aglets system defines a policy description language to define access control lists for resources such as files, sockets, and runtime objects. These ACLs can be configured depending on the agent’s source host. For details about available permissions see the Aglets white paper [11].
Table 1. Realization of the Abstract Model in Aglets

<table>
<thead>
<tr>
<th>Model</th>
<th>Aglets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Agent</td>
<td>Aglet</td>
</tr>
<tr>
<td>Place</td>
<td>Context</td>
</tr>
<tr>
<td>Place Resources</td>
<td>Internal objects and Aglets</td>
</tr>
<tr>
<td>Agent System</td>
<td>Tahiti</td>
</tr>
<tr>
<td>Agent System Resources</td>
<td>Internal objects and Aglets</td>
</tr>
<tr>
<td>Region</td>
<td>missing</td>
</tr>
</tbody>
</table>

3.2 Authorization Attacks in Aglets

Some attacks have already been identified by developers [11] or have been theoretically shown in research papers [16]. So this paper only describes attacks that were novel at the time of the tests.

*Code repository attacks.* Starting with the access control list, an attack to obtain a reference to the code repository of the Aglets system was attempted. The code repository is not directly accessible by the agent through a reference, and, therefore, it was necessary to obtain the associated information indirectly. We found that by using the Java reflection classes it was possible to disclose information about the system’s code repository. To perform the attack, the agent first throws an exception. The exception stores a snapshot of the current execution stack trace. The stack trace stored in the exception is then analyzed and all the class names referenced in the stack are stored for further processing. Once a number of classes have been identified, the Java reflection classes are used to obtain the constructor, attributes, methods, interfaces, and superclass of the class. By examining the signatures of the methods, more classes are found. These classes are added to the ones found in the first phase. The discovery process stops after each class has been analyzed and stored. At this point portions of the code have been revealed. In the final phase the classes are examined to find if there are any static methods or attributes. These are particularly useful because they allow an agent to perform operations without the need for an object reference.

*Security policy attacks.* The access gained with the previous attack established the basis for an attack against the security policy component. More precisely, we found that the policy database can be accessed by using a static method. This means that it is possible to access the policy database even without having any reference to the policy object. This is not a problem per se, but when write access to the policy object was attempted, it was found that modifications to the policy database are not checked by the Security Manager. So it is possible to add or modify all policies without getting any security exceptions, effectively compromising the security of the system.

*Graphic user interface attacks.* Part of the analysis focused on the possibility of an agent accessing the graphic system of the Aglets platform. In fact, access to the graphic interface
allows an agent to interact with the user sitting at the host graphic console. In principle, the Aglets system only allows agents to create windows with a warning banner. This is to prevent a malicious agent from spoofing legitimate applications (e.g., a login prompt) that may be used to induce the user to insert sensitive information. We found that, due to a bug in the implementation, the permission “showWindowWithoutWarningBanner” is completely useless. Although an agent is not granted the permission, the agent is able to open frames and dialogs (neither of them includes a warning label). This vulnerability was exploited by creating a spoofed login prompt that simulated an operating system request for user authentication. The agent would then obtain username and password and mail them back to the user.

4 Jumping Beans

Jumping Beans [4], developed by AdAstra Engineering, is a commercial framework for implementing mobile agent applications. The analysis in this paper is based on version 1.1. The current version is 2.1.1.

4.1 The Jumping Beans Model

In the Jumping Beans framework mobile agents are called “Mobile Applications”. The code space of a mobile agent includes application-specific Java classes and classes that are part of the Java runtime system. A mobile agent component is implemented as a Java thread and the associated execution state is the thread stack and program counter. The data space may include references to other agents or to external objects.

Jumping Beans does not distinguish between agent systems and places. An agent system instance is called “Agency” and provides only one place. The code repository for an agency includes Java classes for the runtime and site-specific classes for the implementation of local services. Agency resources can be implemented in two ways: they are either represented by mobile agents or they are directly bound to the agent system. It is possible to define one agent system local object per instance. For example, the object may be used to implement a broker service or a wrapper for an external database.

Jumping Beans provides the concept of region. A region is controlled by a component called “Server”. Agent systems within a region have to register with the server, which maintains access control lists for region resources and monitors agent systems and agents in a centralized way. Table 2 provides an overview of the mapping between our abstract model and Jumping Beans.

4.2 Authorization Attacks in Jumping Beans

Jumping Beans implements an authorization system that supports access control lists for certain resources (e.g. network, file system, etc.). For a more detailed list see the Jumping Beans white paper [5]. The agent system policies are set by the administrator through the region’s server. Authorization is enforced by the agent systems. The agent system receives the ACLs from the region server and enforces them through an implementation
Table 2. Realization of the Abstract Model in Jumping Beans

<table>
<thead>
<tr>
<th>Model</th>
<th>Jumping Beans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>Mobile Application</td>
</tr>
<tr>
<td>Place</td>
<td>Agency</td>
</tr>
<tr>
<td>Place Resources</td>
<td>Internal objects and Agents</td>
</tr>
<tr>
<td>Agent System Resources</td>
<td>Internal objects and Agents</td>
</tr>
<tr>
<td>Region</td>
<td>Server</td>
</tr>
</tbody>
</table>

of the Java Security Manager. Starting from version 1.1 Jumping Beans also includes a role model for agent system owners. Therefore, it is possible to define access control lists for groups and to assign users to groups. Every mobile agent has a separate permission set and access control list. As a consequence of mobile agent migration, an agent’s permissions may get more restrictive, but never less restrictive.

Unauthorized access to the contents of code fragments is implemented by bytecode obfuscation and “final” classes, which are classes that cannot be subclassed. Both mechanisms are not reliable. Bytecode obfuscation makes it harder to reverse engineer Java bytecode, but does not prevent it; a determined attacker may successfully decompile and reverse-engineer the Java classes. The final class mechanism was successfully attacked by removing the final flag in the obfuscated bytecode and creating a malicious subclass. Although the bytecode is obfuscated it is still possible to disclose data, code, and flow control by using the exception mechanisms and the reflection functionalities provided by the Java runtime, as discussed in the previous section for Aglets.

In addition, as mentioned before, Jumping Beans uses the least trust principle (an agent can only become more restricted). However, when analyzing the implementation of the least trust principle, we discovered that it had been implemented without any exceptions. Because of this, the mechanism can be exploited to perform an attack against the access capabilities of a server. To be more specific, if an agent removes all access privileges to itself, then it is impossible even for the region controller to remove the agent from the target agent system. The agent system’s state has to be manually deleted (otherwise, the agent would be restarted after a reboot) and the system has to be restarted.

Graphic user interface attacks. In analyzing the access to the graphic system we found that the GUI is implemented in a separate thread. Because of this, after a window has been opened it no longer belongs to the agent. So it is possible for an agent to open window frames and move onto the next host. After migration, all the windows that were opened remain open. The successful implementation of this attack opens a window the size of the whole screen. This window cannot be closed except by closing the entire virtual machine, disabling the agent system.

Runtime system calls attacks. In attempting to access the system’s runtime code repository a complete check of the available system-related calls was performed. The Security Manager blocked most of the attempts but, due to an incomplete implementation of the
Security Manager, it was possible to invoke the static method “System.exit()”, which is the exit routine provided by the Java runtime. The net effect of this call is to shut down the whole system\(^1\).

## 5 Grasshopper

The Grasshopper mobile agent system [1] is developed by GMD FOKUS and distributed by IKV++ [3]. Grasshopper is the reference implementation for the OMG’s MASIF specification [10]. The current version of Grasshopper is 2.2. The analysis of the system was performed using Grasshopper version 1.2.2.3.

### 5.1 The Grasshopper Model

The Grasshopper model closely follows the one described in the MASIF specification. A mobile agent is called “Service” and an agent system is called “Agency”. Agencies contain “Places” and are organized in “Regions”. The basic infrastructure is accessible via the agent system and the local infrastructure has to be implemented in separate agents. The mapping from our abstract model to Grasshopper is shown in Table 3.

<table>
<thead>
<tr>
<th>Model</th>
<th>Aglets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>Service</td>
</tr>
<tr>
<td>Place</td>
<td>Place</td>
</tr>
<tr>
<td>Place Resources</td>
<td>Agents</td>
</tr>
<tr>
<td>Agent System</td>
<td>Agency</td>
</tr>
<tr>
<td>Agent System Resources</td>
<td>Internal objects and Agents</td>
</tr>
<tr>
<td>Region</td>
<td>Region</td>
</tr>
</tbody>
</table>

### 5.2 Authorization Attacks in Grasshopper

Grasshopper’s authorization system is similar to the one implemented in the Aglets SDK. However, the implementation of the Java Security Manager in Grasshopper is incomplete.

*Trusted code base attacks.* Similar to the Aglets SDK, Grasshopper uses trusted classes. These classes override the Security Manager and are not checked for access. In the case of Grasshopper this leads to a security leak. The third party trusted class `javax.swing.JInternalFrame` can be used to exit the virtual machine. Therefore, it is possible to exit the server.

\(^1\) This attack also bypasses the persistency mechanisms built into the system, making recovery impossible.
Graphic user interface attacks. In analyzing the access to the graphic system we found that the `checkAwtEventQueueAccess` method has not been implemented. By exploiting this vulnerability it was possible to access the event queue associated with the graphic interface and trace the graphic events. Through the event references it was possible to obtain a handle to graphic components external to the agent. The components were then controlled by sending spoofed events. An attack that sends the key-code “Alt-Shift-Q”, which quits the Grasshopper agent system, was implemented. The attack also monitors the event queue for the appearance of a dialog asking the user to acknowledge the quit command and sends a return key event, simulating the “confirmation click”. By doing this, it was possible to bypass the authorization system and to shutdown the Grasshopper agent system.

System properties attacks. By analyzing access to system properties we found that there is no security check on calls to the `checkPropertyAccess` method. By exploiting this vulnerability it is possible to access and modify any property that is available in the system, for example `agency.name`, `agentsystem.protocol`, `region.registry.host`, or `user.home`.

Policy system attacks. When trying to test the access to the policy system we found that, similar to the Aglets system, the policy is accessed through static methods and variables. Although access to the policy object is successfully enforced and special permissions are needed to access the policy object, it is still possible to instantiate a new policy object. Since the policy object is static, the new instance is automatically the valid policy. Although it does not immediately affect the system, the new policy will affect the system the first time that the system manager opens the policy configuration dialog.

6 Conclusions

This paper presented some initial results of a research effort aimed at the analysis of the security issues in mobile agent systems. Three Java-Based mobile agent systems implementing security mechanisms were installed on a testbed network, these systems were analyzed, and numerous attacks were launched against them. The analysis found many interesting vulnerabilities.

The long term goal of this study is to understand the security issues in MASs and to provide a reference model that can help in abstracting security mechanisms and in defining attack classes in a way that is independent of a particular technology. By doing this, the security analysis results can be reused as guidelines to evaluate the security of other MASs. In addition, the use of a reference model highlights the security abstractions available in the different languages. Complex applications may require sophisticated security abstractions such as policies, different types of principals, and so on. If these concepts are not available, they have to be developed on top of the existing system, which is usually time-consuming and error-prone.

In this paper we concentrated on the attacks performed by a mobile agent against the authorization mechanisms. Many other attacks were suggested by the analysis, and other systems have been installed in the testbed network. Future work will focus on
completing the security analysis of the additional systems and in developing a reference model.

The next step in this research effort will be to build on the experience gained from the security analysis and develop guidelines for the design and development of secure mobile agent systems. Eventually the guidelines will be used to develop a secure agent system that could be effectively used to develop mission-critical mobile agent applications.

References

Formal Specification and Verification of Mobile Agent Data Integrity Properties: A Case Study

Xavier Hannotin, Paolo Maggi, and Riccardo Sisto

Dip. di Automatica e Informatica - Politecnico di Torino
Corso Duca degli Abruzzi 24, I-10129 Torino, ITALY
maggi@athena.polito.it

Abstract. The aim of the work presented in this paper is to check cryptographic protocols for mobile agents against both network intruders and malicious hosts using formal methods. We focus attention on data integrity properties and show how the techniques used for classical message-based protocols such as authentication protocols can be applied to mobile agent systems as well. To illustrate our approach, we use a case study taken from the literature and show how it can be specified and verified using some currently available tools.

1 Introduction

The use of software architectures based on mobile agents to develop distributed computing systems is gaining more and more attention because it seems to exhibit advantages over the traditional client-server paradigm in several applications. For example, the use of mobile agents normally allows to save bandwidth between the user end-terminal and the network, which is very useful when employing mobile terminals. Thus, mobile agents potentially can become a widely used new paradigm for distributed computing. However, to make this paradigm acceptable, it is necessary to manage the various security problems arising when it is adopted [1,2]. Such problems generally fall into two main categories. On one hand, it is necessary to protect hosts from malicious agents coming from the network, on the other hand it is necessary to protect agents from malicious hosts and network intruders. While the first kind of problem has been studied extensively, at the moment only some partial solutions are available for the second one (e.g. [3,4,5,6,7]).

In general, all the solutions adopted to ensure security properties in distributed systems are based on some kind of cryptographic protocol which, in turn, uses basic cryptographic operations such as encryption and digital signatures. Despite their apparent simplicity, such protocols have revealed themselves to be very error prone, especially because of the difficulty generally found in foreseeing all the possible attacks and all the possible behaviors of various parallel protocol sessions. For this reason, a lot of attention is being paid to formal methods which can help in developing error-free protocols or in analyzing the vulnerability of existing protocols. Up to now, such methods have been success fully employed to formally verify security properties of classical message-based
protocols, such as authentication protocols, whereas they have not yet been applied to analyze the security of agent-based systems. It is worth noting that the specification and verification of security issues related to mobile agent systems involves new aspects not encountered in classical cryptographic protocols. In particular, in addition to the classical threats, such as those related to possible alteration of messages being transmitted, it is also needed to model the fact that the correct behavior of an agent is not guaranteed when the agent is being executed in an untrusted host. This means that the behavior of an agent potentially becomes unpredictable each time it visits an untrusted host.

This paper addresses such new aspects and explores the possibility of applying existing formal specification and verification techniques to those cryptographic mechanisms specifically designed for the protection of agents from their environment, with a particular emphasis on agent data integrity, which is the most typical property of interest.

In particular, we decided to focus attention on the CSP-based approaches, which have already been extensively used to specify and verify classical cryptographic protocols [8,9,10]. For what concerns verification techniques, we restrict our attention on the model checking ones, which have the nice feature of not requiring excessive expertise to be used.

The rest of the paper is organized as follows. In section 2, we present a sample mobile agent cryptographic protocol that will be used throughout the article to illustrate our modeling approach and its potentials. In section 3 we present the modeling approach in general terms, whereas in section 4, we show how it can be used to provide a formal model of an instance of the sample protocol and to define data integrity properties. In section 5 we present some verification results, obtained using the CSP-based tools Casper and FDR. Section 6 concludes.

2 A Sample Protocol for Mobile Agents Data Integrity

A mobile agent (MA) is a program that can migrate from one network host to another one while executing. Mobile agents are executed by agent interpreters that run on each host and communicate by message passing. An agent migrating from one host to another host consists of a static part, typically including the agent code and, possibly, some static data, and a dynamic part, including all the agent elements that can change over time (program counter, stack, variables, etc.).

In this section, we present a simple protocol which aims at the integrity of a data gathering mobile agent that runs on several possibly untrusted hosts. This agent goes to several hosts and simply picks up pieces of data on its way. For example, the agent could be a shopping agent dispatched to visit different companies and find out the prices at which they sell a given product, so as to select the cheapest company. Data integrity of such an agent means that a host cannot tamper with the data already collected without being detected. This is a classical problem for which different protocols have been proposed [3,4,5,6,7].

The specific protocol we consider here was proposed in [6] by Corradi et al.. The idea is that agents carry along a cryptographic proof of the data they
have already gathered. This proof prevents hosts from tampering with the data already collected without being detected.

For the description of the protocol we use the following notation. \( \text{hash}(\cdot) \) is a cryptographic hash function, i.e., a function which, theoretically, cannot be inverted (\( x \) cannot be deduced from \( \text{hash}(x) \)). Encryption of \( x \) by public (private) key of host \( H \) is denoted \( \{ x \}_{PK(H)} \) (\( \{ x \}_{SK(H)} \)).

We also take some notation from [6], where MIC stands for “Message Integrity Code” and is the cryptographic proof we have just mentioned. \( C \) is a cryptographic counter, which is incremented by successive applications of \( \text{hash} \) by the agent and \( AD \) is the list of already collected data. The hosts where data have to be collected are decided by the agent dynamically, in such a way that each host is visited at most once. The hosts are denoted, in order of visit by the agent, \( H_{i_0} \) which is the initiator, and \( H_i \) (\( 1 \leq i \leq n \)), which are the hosts where data must be collected. The initiator initially creates the agent, sends it out and, at the end of the computation, receives it with the collected data. Each host \( H_i \) (\( 1 \leq i \leq n \)) has a piece of data \( D_i \) that will be collected by the agent.

\( AD_i \) and \( MIC_i \) are respectively the collected data and the MIC value after the agent has left \( H_i \). Similarly, successive values taken by the cryptographic counter are denoted \( C_i \). CID is the (static) code of the agent. It is signed by a trusted party for authentication and it is carried along from host to host with the agent. The agent moving from host \( H_{i-1} \) to host \( H_i \) can be represented by a message containing CID, \( AD_{i-1} \), \( MIC_{i-1} \) and \( C_i \).

2.1 Protocol Description

- **Initialization:** \( H_0 \) generates a secret number \( C_0 \). It creates the agent and passes \( C_i = \text{hash}(C_0) \) to it.

- **First Hop:** The agent encrypts \( C_1 \) with \( PK(H_1) \) to let it be accessible only on \( H_1 \), and then moves to \( H_1 \), carrying with itself only the encrypted \( C_1 \) (i.e., \( \{ C_1 \}_{PK(H_1)} \)). The collected data list \( AD_0 \) and the initial MIC \( MIC_0 \) are empty.

- **On host \( H_1 \):** After the agent has reached \( H_1 \), it asks \( H_1 \) to decrypt \( \{ C_1 \}_{PK(H_1)} \), thus obtaining \( C_1 \) and collects \( D_1 \), so having \( AD_1 = \{ D_1 \} \). Then it computes \( MIC_1 = \text{hash}(D_1, C_1) \), and it increments the cryptographic counter by computing \( C_2 = \text{hash}(C_1) \).

- **Hop \( i \) (\( 2 \leq i \leq n \)):** The agent encrypts \( C_i \) with \( PK(H_i) \) and then moves to host \( H_i \) carrying with itself the already collected data \( AD_{i-1} \), the cryptographic proof \( MIC_{i-1} \), and the value of the cryptographic counter encrypted with \( H_i \)'s public key: \( \{ C_i \}_{PK(H_i)} \).

- **On host \( H_i \) (\( 1 \leq i \leq n \)):** After having decrypted \( \{ C_i \} \) \( PK(H_i) \), the agent collects \( D_i \) and appends it to the collected data list, so computing \( AD_i = AD_{i-1} \cup \{ D_i \} \). It then computes a new proof by:

\[
MIC_i = \text{hash}(D_i, C_i, MIC_{i-1})
\]

and increments the cryptographic counter.

- **Last hop:** The agent encrypts \( C_n \) with \( KP(H_0) \) and then moves from \( H_n \) back to \( H_0 \) carrying with itself the whole data \( AD_n \), the computed checksum \( MIC_n \), and \( \{ \text{hash}(C_n) \} \) \( PK(H_0) \).
Termination: $H_0$ receives from the agent $AD_m$, $MIC_m$, and $\{hash(C_m)\}_{FK(H_0)}$. From the values of $C_0$ and $AD_m$, $H_0$ can compute $MIC_m$ and check that it is the same that has just been received from the agent. If any difference is found, the agent data is considered to be invalid. This guarantees the integrity of validly collected data.

Note that $\{hash(C_m)\}_{FK(H_0)}$ is never examined by $H_0$. For this reason, in our model reported in the following section, the last host does not send it to the initiator host.

A host cannot modify the already collected data, basically because it cannot retrieve $C$ from $hash(C)$. Since $H_i$ does not know $C_j$ ($j < i$), it cannot modify $D_j$ ($j < i$) since it will not be able to reconstruct a valid MIC.

As also noted in [6], this solution does not work if the agent visits a host to collect data twice, or if malicious hosts cooperate.

Note also that, since already gathered data are not encrypted, the protocol just explained allows intermediate hosts to read them if needed. Anyway it is also possible to encrypt gathered data if these ones must be unreadable to intermediate hosts.

3 Modeling a Mobile Agent System

Formal models of cryptographic protocols typically are composed of a set of principals which send messages to each other according to the protocol rules, and an intruder, representing the activity of possible attackers. The intruder can perform any kind of attack: it can not only overhear all the transmitted messages, learning their contents, but it can also intercept messages and send new messages created using all the items it has already learned, as well as new nonces. So the intruder can fake messages and sessions.

Since such models are meant to reveal possible security flaws in the protocols and not flaws in the cryptosystems used by the protocols, cryptography is modeled in a very abstract way and it is assumed to be “perfect”. This means that:

- the only way to decrypt an encrypted message is to know the corresponding key;
- an encrypted message does not reveal the key that was used to encrypt it;
- there is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key.

Although such assumptions are obviously not completely true for real cryptosystems, they represent the properties of an ideal cryptosystem, so they are useful to isolate the flaws of the protocol itself. In other words, any flaw found with this model is a real protocol flaw, but it is possible that the model does not reveal other weaknesses due to the used cryptosystems.

In order to model a mobile agent system, we use a technique quite similar to the one just described, based on the same assumptions about perfect cryptography and intruders. Agents are not modeled as autonomous mobile principals, but
the whole agent-based system is represented at a lower level of abstraction, closer to the real system. Principals represent hosts which, by their agent execution platform, can execute mobile agent code. The migration of an agent from host to host is represented by a message, exchanged by the principals that represent the involved hosts, containing the agent code and data.

Since the integrity of the static agent code and the static agent data is a problem shared by all mobile agents, it can be solved by the MA platform, independently on the particular functionality of the agent. Since we are not interested in validating this part of the protocol, we assume that this problem is already solved in a reliable way and we do not model code transmission explicitly, but we simply assume that trusted hosts always execute the right code. So each agent hop is represented by a message containing only the dynamic part of the agent data. For example, in modeling the protocol described above, messages would just contain the collected data followed by the MIC and C values.

The main new aspect in mobile agent cryptographic protocols with respect to classical authentication protocols is the possibility of having attacks due to intruders and to untrusted hosts that may alter the behavior of agents hosted by their execution platform in an unpredictable way.

Modeling agents by messages exchanged by the hosts helps us in taking all such issues into account. Let us assume that we have a single untrusted host $H_u$. Attacks due to $H_u$ can be taken into account in the above model giving the intruder the possibility of totally replacing it. To obtain this, it is enough to give the intruder all the secrets known by $H_u$, which, in our example, coincide with $H_u$'s private key. Having the private key of $H_u$, the intruder can totally replace $H_u$, i.e. it can intercept any message directed to $H_u$, decrypt it exactly as $H_u$ could do and forge any message $H_u$ could produce in response to it. In other words, the intruder incorporates the behaviors of all possible network intruders as well as those of all possible untrusted hosts. This models any kind of malicious behavior of $H_u$, including any modification in the execution of the mobile agent on $H_u$, as well as the case in which the agent is sent by $H_u$ to a host different from the one where it should go.

This approach can be easily extended to model environments with several untrusted hosts: inserting all their private keys in the initial intruder knowledge. This corresponds to modeling several untrusted hosts that can cooperate. The intruder process knows the keys of all the untrusted hosts and so can replace each of them and use the total knowledge of all of them.

Modeling untrusted hosts that can cooperate is adequate for most applications. Nonetheless, a solution can be found even for cases in which it could be useful to model untrusted hosts that are unable to cooperate. These cases are difficult to model using the modeling approach explained above, because the specifier cannot control how the intruder is modeled, the only thing that can be specified about the intruder being its initial knowledge. However, the model with cooperating untrusted hosts includes, as a special case, the one with uncooperating hosts. Indeed, by analyzing the first model, we can find out all possible attacks against the protocol, including those that do not require any untrusted host cooperation. So a way out of this problem is to analyze all the attacks
reported by the analysis tool and then filter out the ones involving cooperating untrusted host.

For what concerns the specification of data integrity properties, they can be expressed in the same way authenticity properties are specified in classical authentication protocols. Indeed, requiring that some data that is considered valid at a given site has actually been delivered by the expected one is really an authenticity property. For example, in the protocol presented above, the property of interest is that if $H_0$ believes that some data, $D_0$, has been produced by host $H_a$, then $H_b$ has really produced it.

4 The Sample Protocol Model

In this section we show the model of an instance of the protocol reported in section 2 using the Casper notation.

Casper [10] is a freely available tool that takes a description of a cryptographic protocol, written using a notation similar to the one normally appearing in the academic literature, as an input and computes a CSP specification of the protocol suitable for being checked by the model checker FDR.

CSP ("Communicating Sequential Processes") is a process algebra originally devised by Hoare [8]. It allows us to describe systems as a number of components (processes) which operate independently and communicate with each other over well-defined channels.

FDR [9] is a CSP model-checking tool marketed by Formal Systems that inputs an encoding of one or more CSP processes in a CSP machine readable dialect and then checks refinement relations between pairs of processes.

Many case studies in which CSP and FDR have been successfully applied to discover attacks upon cryptographic protocols can be found in the literature (e.g. [11,12,13]). Unfortunately, the task of developing CSP specifications for cryptographic protocols is quite hard and very time-consuming. The main reasons for these problems are that, differently from other process algebras like spi calculus [14], CSP does not define cryptographic primitives and so these must be explicitly implemented using complex CSP primitives. Moreover, the intruder must be explicitly modeled as a CSP process. This makes CSP specifications of cryptographic protocols also difficult to read and to understand.

Anyway, using Casper and its simpler input notation, these difficulties can be overcome.

For simplicity, we deal with a fairly small instance of the protocol. In this instance (Fig. 1), two agents are sent out from the initiator $H_0$, visit two hosts, identified as $H_1$ and $H_2$, and come back to $H_0$. $D_1$ and $D_2$ are the data gathered by the agents on $H_1$ and $H_2$ respectively. Hosts $H_0$ and $H_1$ are trusted, whereas $H_2$ is not. We can express the data integrity property asserting that the data collected at $H_1$ should not be modified when the agents pass through $H_2$. Of course, it is possible to extend the model to incorporate an arbitrary number of visited hosts. Moreover, we can have more than two agents.

The complete code of our model, written in the language accepted by Casper, is reported in Fig. 2, and includes two agents.
A Casper input file can be conceptually split into two main parts, each one containing four different sections. Each section is headed by a line beginning with the character "#". The first part describes in an abstract way how the protocol operates, whereas the second part deals with the particular instance of the protocol to be checked.

In the **Free variables** section, the variables and functions related to the abstract description of the protocol operation are defined. Variables $H_0$, $H_1$ and $H_2$ represent respectively host $H_0$, $H_1$ and $H_2$. Variables $D_1$ and $D_2$ represent the data gathered by an agent on $H_1$ and $H_2$ respectively. Variable $C_0$ represents the secret number $C_0$. Functions $SK$ and $PK$ take a host as an argument and return respectively its secret key and its public key. The statement $InverseKeys = \{PK, SK\}$ means that $SK$ and $PK$ return keys that are inverses of each other when they are applied to the same host. At the end, $hash$ is a cryptographic hash function.

In the **Processes** section the roles played by the different partners involved in the protocol and their initial knowledge are described. In our model, the first role, called **INITIATOR**, represents the process running on abstract host $H_0$. Its initial knowledge consists of the identity of the abstract host $H_1$, the value of nonce $C_0$, the public key function $PK$ (i.e. the public key of all hosts) and the secret key of $H_0$ (i.e. $SK(H_0)$). The second role, called **PROCESSHOST1**, represents the process running on abstract host $H_1$. It initially knows the identity of the abstract hosts $H_0$ and $H_1$, the value of $D_1$, the public key function $PK$ and the secret key of $H_1$. The last role, called **PROCESSHOST2**, represents the process running on abstract host $H_2$. Its knowledge initially consists of the identity of abstract host $H_0$, the public key function $PK$ and the secret key of $H_2$. It is worth noting that in a Casper model a process needs to know the identity of each host it wants to send a message to.

The **Protocol description** section specifies the sequence of messages of the protocol and the tests a host performs when receiving messages. The notation used in this section is fairly intuitive, being similar to the one normally used in the academic literature. However, some attention is needed, because some aspects
# Free variables
H0, H1, H2 : Host
D1, D2 : Data
C0 : Secret
SK : Host -> SecretKey
FK : Host -> PublicKey
Inverseskeys = {FK, SK}
hash : HashFunction

# Processes
INITIATOR[H0, H1, C0] knows FK, SK[H0]
PROCESSHOST1[H1, H0, H2, D1] knows FK, SK[H1]
PROCESSHOST2[H2, H0, D2] knows FK, SK[H2]

# Protocol description
0. -> H0 : H1
2. H1 -> H2 : D1, hash(D1, C1 % hash(C0)) % MIC1, \\
   {hash(C1 % hash(C0)) % C2}[FK(H2)]
3. H2 -> H0 : D2, hash(D2, C2 % hash(C1 % hash(C0))), \\
   MIC1 % hash(D1, C1 % hash(C0))

# Specification
Agreement[H1, H2, [D1]]
Agreement[H2, H0, [D2]]

# Actual variables
Host0, Host1, Host2, T : Host
Data1, Data2 : Data
C01, C02 : Secret

# Functions
symbolic FK, SK

# System
INITIATOR[Host0, Host1, C01]; INITIATOR[Host0, Host1, C02]
PROCESSHOST1[Host1, Host0, Host2, Data1]
PROCESSHOST1[Host1, Host0, Host2, Data1]
PROCESSHOST2[Host2, Host0, Data2]
PROCESSHOST2[Host2, Host0, Data2]

# Intruder Information
Intruder = T
IntruderKnowledge = {Host0, Host1, Host2, FK, SK[Host2]}

Fig. 2. Casper description of Corradi et al.'s protocol.
are not explicitly expressed. For example, tests on messages are implicitly defined by Casper. Each message has got a number. Message number 0 does not belong to the protocol itself. It is a conventional message used to model the protocol start-up: the environment sends a message to H0 to notify it that it has to start a protocol run sending a message to H1. In message 1, H0 sends to H1 the value \texttt{hash(C0)} encrypted with the public key of H1. In fact, the notation \{m\}_k represents message m encrypted with key k, and the notation \texttt{f(m)} represents the application of function f to message m. It is to be noted that whenever the result of a function is sent in a message, Casper implicitly assumes that both the sender and the receiver should be able to calculate the function, and it introduces a corresponding check in the automatically generated CSP code. To avoid this check, it is necessary to use the notation \texttt{m\textasciitilde v}, where m is a message and \texttt{v} is a variable: in this case, the receiver must simply store m in variable \texttt{v}, without trying to interpret it. Similarly, the notation \texttt{v\textasciitilde m} represents that the sender sends the contents of the variable \texttt{v} which must be interpreted by the receiver as being of the form given by \texttt{m}. The \%-notation is normally used to model situations where a party wants to send a message to a receiver that is not supposed to interpret the message, but instead forward it to a third party in a subsequent message. In our model, since \texttt{H1} cannot compute \texttt{hash(C0)} (it does not know the value of \texttt{C0}), we must use the notation \texttt{hash(C0) \% C1}.

In message 2, H1 sends to H2 the value of \texttt{D1, hash(D1, C1 \% hash(C0))} and \texttt{hash(C1)} encrypted with the public key of H2. This last value will be stored by H2 in a temporary variable called C2, while the previous one will be stored in MIC1. In message 3, H2 sends to H0 the values of \texttt{D1 and D2 and hash(D2, C2, MIC1)}. C2 will be interpreted by H0 as \texttt{hash(C1 \% hash(C0))} and MIC1 as \texttt{hash(D1, C1 \% hash(C0))}.

The Specification section reports the security properties we want to check. We use a kind of authentication proposed by Casper and called Agreement. The statement Agreement(H1, H0, \{D1\}) is an assertion that is true if, at the end of the protocol run, H1 is correctly authenticated to H0 and the two hosts agree upon the value of D1 (note that this happens if and only if the protocol ensures the integrity of the data D1 gathered by an agent on abstract host H1). Similarly, the statement Agreement(H2, H0, \{D2\}) is an assertion that is true if, at the end of the protocol run, H2 is correctly authenticated to H0 and the two hosts agree upon the value of D2 (in our protocol instance this assertion should be false because H2 is not trusted).

The next sections describe the actual protocol instance to be checked. In order to be able to consider also attacks that exploit the presence of more than one agent, we have to instantiate more copies of the processes just defined, each one corresponding to a different agent run. For simplicity, we define only two instances for each process. In the Actual variables section the variables needed to describe the protocol instance are defined. Variables Host0, Host1, Host2 and T represent respectively hosts H0, H1, H2 and the intruder. Variables Data1 and Data2 contain respectively the actual values gathered by agents on hosts H1 and H2. Variables CO1 and CO2 hold the secret numbers generated by the initiator, respectively for the first and the second agent.
The Functions section contains only the statement symbolic PK, SK which means that functions PK and SK are not explicitly defined but that Casper itself has to produce the results of the applications of such functions.

The System section gives a description of the protocol instance, instantiating the processes defined in the Processes section with actual parameters.

In the last section, the Intruder information section, we define the intruder’s identity to be T and define that the intruder initially knows all the hosts’ identities and public keys, i.e. the function PK, and the private key of H2. Note that SK(Host2) belongs to the intruder initial knowledge since H2 is untrusted.

5 Checking the Sample Protocol

Compiling the model just presented using Casper (version 1.2.3 beta), we obtain an equivalent CSP model. If verified using FDR (version 2.75), the model generates a states graph with 7232 states and 41420 transitions. The time needed to check the protocol security properties is about 13 minutes on a PC running the RedHat Linux 7.0 operating system on an AMD Athlon 850 MHz CPU with 512 Mb of RAM. As expected, the first property (i.e. the one concerning integrity of the data collected at the trusted host) is checked successfully, while the second one is not. We also tried the verification of a protocol instance with four hosts including an initiator, one trusted host and two untrusted hosts, to see if the tool would discover the attack on the protocol, but unfortunately the model was too large to be checked on the above machine.

6 Conclusions

In this paper we have explored how some formal specification and verification techniques, originally developed to check various security properties of cryptographic protocols, can be applied to check integrity properties of mobile agents as well. Up to our knowledge, this is the first documented attempt in this direction.

Mobile agent systems can be modeled by means of the messages exchanged by the hosts where they are executed, in a way quite similar to the one used for authentication protocols. In this way, untrustedness of hosts can be modeled simply making their secrets known to the intruder. An example of such a modeling approach has been presented for a sample data integrity-preservation mechanism.

A clear formal specification as the one that has been presented in this paper is very important to unambiguously and precisely describe a mobile agent security mechanism and its properties. It expresses not only the contents of the exchanged messages, but all the aspects that are relevant to guarantee the security properties of interest, including the checks that must be done when messages are received. So a formal specification of this kind is a valid and practical basis for a correct implementation.

From the verification point of view, we found that the CSP-based tools Casper and FDR, which have been used to formally verify several classical cryptographic protocols by model checking, can be used to check mobile agent system
models developed as explained in this paper. The maximum complexity of the
systems that can be checked in this way is still quite low. However, it is to be
noted that research on model checking of cryptographic protocols is providing
more and more efficient solutions, and some new quite efficient model check-
ers for security properties have recently been announced [15,16]. Although not
yet available, such tools will probably make verification of mobile agent system
models cost-effective.

References

Agents: Cooperative, Rational and Adaptive Information Gathering on the Inter-
431–446
2. Farmer, W., Guttman, J., Swarup, V.: Security for mobile agents: Issues and
Diego, Department of Computer Science and Engineering (1997)
Agent Security. Lecture Notes in Computer Science No. 1419. Springer-Verlag:
Heidelberg, Germany (1998) 137–153
5. Karcioth, G., Asokan, N., Giller, C.: Protecting the computation results of free-
roaming agents. In Rothermel, K., Hohl, F., eds.: Proceedings of the 2nd Inter-
national Workshop on Mobile Agents. Volume 1477 of Lecture Notes in Computer
applications. In: of the 19th IEEE International Conference on Distributed Com-
puting Systems Workshop [ICDCS'99], Austin, Texas, IEEE Computer Society
for internet trading. In Zhang, C., Lukose, D., eds.: Proceedings of the 4th Aus-
tralian Workshop on Distributed Artificial Intelligence on Multi-Agent Systems :
Theories, Languages, and Applications (DAII-98). Volume 1544 of LNAl, Berlin,
Germany, Springer (1998) 183–193
Cliffs (NJ), USA (1985)
Proceedings of The 10th Computer Security Foundations Workshop, IEEE Compu-
Transactions on Software Engineering 23 (1997) 659–669
12. Schneider, S.: Verifying authentication protocols with CSP. In: PCSFW: Pro-
cceedings of The 10th Computer Security Foundations Workshop, IEEE Computer
Society Press (1997)
13. Roscoe, A.W.: Modelling and verifying key-exchange protocols using CSP and
FDR. In: Proceedings of the Eighth Computer Security Foundations Workshop


Lime Revisited
Reverse Engineering an Agent Communication Model

Bogdan Caranu, Marco Tulio Valente, and Jan Vitek
Dept. of Computer Sciences
Purdue University

Abstract. Lime is a middleware communication infrastructure for mobile computation that addresses physical mobility of devices and logical mobility of software components through a rich set of local and remote primitives. The system's key innovation is the concept of transiently shared tuple spaces. In Lime, mobile programs are equipped with tuple spaces that move whenever the program moves and are transparently shared with tuple spaces of other co-located programs. The Lime specification is surprisingly complex and tricky to implement. In this paper, we start by deconstructing the Lime model to identify its core components, then we attempt to reconstruct a simpler model, which we call CoreLime, that supports fine-grained access control and can better scale to large configurations.

1 Introduction

Traditional computational models are based on the assumption that software and devices are deployed before being used, and that once deployed configurations are relatively static. Wireless computing and ad-hoc networks invalidate such assumptions as both devices and the software that runs on them may be mobile. To address this issue, a number of theoretical models such as Ambients [6] and Seal [15] have adopted migratory computations as their key linguistic abstraction. In these theoretical studies migratory computations or mobile agents can model both physical and logical mobility. But in practice dealing with physical mobility has proved much more challenging than software mobility. Not surprisingly, most practical mobile agent systems [10,3] have focused on providing mechanisms for moving code, and have mostly ignored mobility of devices.

Our experience with the implementation of a medium-sized agent application [11] suggests that high-level communication primitives are the main shortcoming of most agent systems. Designing communication middleware for physical mobility is a challenging task. Mobile systems have markedly different characteristics from traditional distributed or concurrent systems. Communication in a mobile system is inherently:

- Transient and Opportunistic: Communication patterns are shaped by the nature of an environment in which hosts are intermittently connected to the network and agents can leave a host at any time. Communication thus tends to be opportunistic; applications take advantage of resources that
happen to be available at a particular time without relying on their continued availability. Communication protocols must accommodate long latencies and time outs caused by the sudden departure of an interlocutor or disconnection of the agent itself.

- **Anonymous and Untrusted:** Interactions can be based on services offered rather than on the identity of the entity providing those services. Agents do not necessarily have to know each others names and locations to interact as long as the needed services are being provided. The corollary of anonymity is that interlocutors do not necessarily trust each other which implies that the communication infrastructure must provide the mechanisms needed to implement secure communication protocols.

In 1999 Murphy, Picco and Roman [14,12] introduced Lime, an elegant combination of Gelernter’s Linda [8] with reactive programming. The design’s goal was to provide a simple communication model for mobile environments. Lime introduces the notion of *transiently shared tuple spaces*. In the model each mobile entity is equipped with its own individual tuple space which moves whenever that entity moves. These individual tuple spaces are silently merged by Lime as soon as several agents are located on the same host, thus creating temporary sharing patterns that change as agents enter and leave the host. Furthermore *ad hoc* federations of hosts can be created dynamically. In this case, Lime merges the tuple spaces of each host into a single seamless federated tuple space. Transient sharing solves several problems of tuple space communication models in the context of mobile environments. At the local level, it introduces a notion of ownership for tuples that is beneficial for resource accounting purposes. Furthermore, tuple space migration allows mobile entities to suspend an ongoing interaction and resume it whenever both agents happen to be colocated again. At the federated level, transient sharing provides a model of a distributed space in the face of mobility.

The original goal of this work was simply to extend Lime with the access control mechanisms needed to implement secure interaction between untrusted parties and use that model in the implementation of a new mobile agent system for limited capacity connected devices. Along the way we realized that the Lime specification was somewhat complex and difficult to implement and that the model appeared to have some ingrained inefficiencies. These suspicions were confirmed by preliminary experiments with a prototype implementation. This paper documents our attempts to understand Lime and to provide a scalable and secure implementation of its key ideas. We start by providing a new formalization of the core concepts of Lime as a process calculus. This gives a well understood starting point for reasoning about Lime programs and can be seen as a specification for implementers. Then we define CoreLime, an even simpler calculus which does not have some of the inherent inefficiencies of Lime. Finally we describe security extensions that we are adding to an implementation of CoreLime.
2 Lime: Middleware for Mobile Environments

This section introduces the Lime middleware communication infrastructure for mobile environments. When necessary we will differentiate between the implementation of Lime multiprocessing, and its specification, Lime [13].

Lime basics. Programs in Lime are composed of agents equipped with possibly many tuple spaces. Agents run on hosts with active tuple space managers. The basic tuple space operations available in Lime are familiar from Linda systems. Agents can deposit data in a tuple space with a non-blocking out operation, remove a datum with a blocking in or a non-blocking inp. They can further obtain a copy of a tuple with rd and rdp.

Reactive programming. Lime introduces the concept of reactions. A reaction can be viewed as a triple (t, s, p) consisting of a tuple space t, a template s and a code fragment p. The semantics of a reaction is that whenever a tuple matching s is deposited in t, the code fragment p should be run. The main difference between the blocking rd and reactions is that all matching reactions are guaranteed to be run when a matching tuple is found. Furthermore, Lime specifies that reactions are atomic: in other words while p executes, no other tuples space operation may be processed. The code of a reaction is allowed to perform tuple space operations and may thus trigger other reactions. Lime executes reactions until no more reactions are enabled. To avoid deadlocks reactions are not allowed to issue blocking tuple space operations such as in or rd.

Location-aware Computing. Lime lets agents perform operations on tuple spaces of other agents by the means of location parameters. Location parameters restrict the scope of tuple space operations. For the out operation, a location parameter can be used to specify the destination agent of a tuple. Its semantics is that Lime will deliver the tuple to the destination as soon as the destination agent becomes reachable. While the destination agent is not reachable tuples remain under the ownership of their creator. One way to represent this ownership information is to think of each tuple as having two additional fields current and final such that current denotes the current owner of the tuple and final its destination. A tuple for which current ≠ final is in transit (also called misplaced in Lime). Lime implementations need not maintain these fields explicitly, they are useful for the exposition though.

Transitively Shared Spaces. By default, the tuple spaces of different agents are disjoint and agents can not use tuple spaces to communicate. The key innovation in Lime is to support a flexible form of tuple space sharing referred to as transient sharing. An agent can declare that some of its tuple spaces are shared. The Lime infrastructure will then look for other spaces, belonging to different agents, with the same name and silently merge them into a single apparently seamless space. The sharing remains in effect as long as the agents are co-located. Although the model does not provide for agent mobility, the underlying assumption is that agents can leave a host at any time. When this occurs, Lime will
break up the tuple space and extract all tuples which have the departing agent in their current field. Transient sharing simplifies the coding of application communication protocols as explicit location parameters can be omitted to search the entire shared space.

**Federated Spaces and Mobile Hosts.** The last and most ambitious part of Lime is the support for federated spaces. A federated space is a transiently shared tuple space that spans several hosts. Federations arise as a result of hosts issuing the engage command. Hosts can leave a federation by issuing an explicit disengage command. The semantics of Lime operations are not affected by federations, it is up to the implementation to provide the same guarantees as in the single host case. This complicates the implementation and imposes some constraints on the use of Lime primitives. In particular, Lime\_mp introduces weak reactions and limit (strong) reactions to a single host. A weak reaction may be scoped over multiple hosts, but it adds an asynchronous step between identification of the tuple and execution of the reaction code. Tuples that may trigger weak reactions are first set aside, and then the user reactions are executed atomically. In Lime\_mp, a weak reaction is implemented by registering one strong reaction on every node of the federation.

3 Deconstructing Lime

This section presents a language that formalizes the coordination model proposed by Lime. We depart from Murphy’s formalization by choosing an operational semantics in the style of the asynchronous π-calculus [9,2,1]. The main reason for the departure is that it allows for a self-contained semantics, which does not have to rely on extraneous definitions. Furthermore, we hope to obtain a compact and simple formalization. The main difference between our formalism and π-calculus is the use of generative communication operations instead of channel-based primitives. The idea of embedding a Linda-like language in a process calculus has been explored in depth in previous work [4,7].

Table 1 defines the syntax of the Lime calculus. We assume a set of names $N$ ranged over by meta-variables, $a, s, h, x$. Basic values, ranged over by $v$, consist of names and tuples. Tuples are ordered sequences of values $(v_1, \ldots, v_n)$. A tuple space $T$ is a multiset of tuples. We use the symbol $\exists a \in N$ to denote the distinguished unspecified value. As usual this value is used to broaden the scope of matching operations.

A configuration is a pair composed of a set of agents $A$, a tuple space $T$ and a global set of names $X$. Each agent $a \in A$ is written $a_k[P]$, where $P$ is the process running inside the agent and $h$ the name of the host on which the agent is running. Agent tuple spaces are modeled by a single global tuple space $T$. Additional information attached to each tuple will let us distinguish ownership and current location of tuples. Agents can have multiple private tuple spaces represented by disjoint views over the global tuple space $T$. These private tuple spaces are identified by names, and any two private tuple spaces with the same name are considered to be transiently shared. The names used over several hosts in the system are recorded in the set $X$, ensuring their unicity.
Table 1. Lime calculus syntax

\[
egin{align*}
\text{Prog} & ::= A, T, X \\
A & ::= \varepsilon | \alpha[P] | A \\
P & ::= 0 | P | Q | \lambda P | \text{out } v \\
& \quad | \text{inv } v, x.P | \text{rd } v, x.P | \text{move } h.P | \text{react } v, x.P
\end{align*}
\]

Processes are ranged by \( P \) and \( Q \). The inert process \( 0 \) has no behavior. Parallel composition of processes \( P \parallel Q \) denotes two processes executing in parallel. Replication of processes \( \lambda P \) denotes an unbounded number of copies of \( P \) executing in parallel. The restriction operator \( \lambda x.P \) generates a fresh name \( x \) lexically scoped in process \( P \).

The \text{out} operation expects a tuple \( v = \langle v' a s \rangle \) as argument. The first element of the tuple is the value \( v' \) (a tuple itself) to be output, \( a \) is the destination agent and \( s \) is the tuple space. The \text{in} and \text{rd} primitives expect an argument tuple \( \langle v a a' s \rangle \) and the name of the variable that will be bound to the result. The argument tuple consists of the template to match \( v \), the current owner of the desired tuple \( a \), the destination of the desired tuple \( a' \) and the tuple space \( s \). The unspecified value can be used to broaden the scope of input operations, e.g., if both current and destination fields are left unspecified, the entire space will be searched.

The \text{move} primitive can be used by agents to migrate between connected hosts. When an agent performs this operation all the tuples that have the agent as the current location are removed from the source host, moved with it, and inserted in the destination host. The primitive \text{react } v, x.P \) is used to register a reaction on the host where the agent that executes it resides. The first argument is the tuple that has to be matched in order for the reaction to be triggered, the second argument is a variable and the third argument a process, called the body of the reaction, that will be executed atomically upon occurrence of such an event.

3.1 Semantics of Lime

We now give an operational semantics for the Lime calculus. For clarity we split the semantics in three sets of rewrite rules. The semantics is defined in Table 2 and will be detailed next.

Primitive operations. The first set of rewrite rules defines tuple space operations, and is of the form \( A, T, X \rightarrow A', T', X' \) where a configuration is a pair \( A, T, X \) such that \( A \) is a set of agents, \( T \) is a tuple space, and \( X \) is a global set of names. Each step of reduction represents the effect on the program and tuple space of executing one Lime primitive operation.
The input \( \text{inv}(v, x, P) \) and read \( \text{rdv}(v, x, P) \) operations try to locate a tuple \( v' \) that matches \( v \). If one is found, free occurrences of \( x \) are substituted for \( v' \) in \( P \). In the case of the input, the tuple is removed from the space. The definition of pattern matching, written \( v \leq v' \), allows for recursive tuple matching. Values match only if they are equal or if the unspecified value occurs on the left hand side.

Output \( \text{out} \) is asynchronous in Lime, and thus has no continuation. Each output tuple \( \langle v \ a \ s \rangle \) is first transformed into a Lime value tuple, i.e. \( \langle v \ a' \ s \rangle \), and added to the global space. The Lime value tuple format has two agent names, \( a \) is the current agent that “owns” the tuple and \( a' \) is the destination agent. We say that a tuple for which \( a \neq a' \) is misplaced. This can occur only if the destination is not connected. The auxiliary function \( mkt \) makes a new Lime value tuple. If it can not locate the destination the tuple will be misplaced otherwise the tuple will be delivered.

Agent move operations \( \text{moveb}(v, P) \) change the location of the current agent. Furthermore, an auxiliary function \( mvt \) moves all the tuples to the new host. Finally, reaction operation \( \text{react}(v, x, P) \) creates a Lime reaction tuple and deposits it in the global space. Here \( v \) is expected to have the form \( \langle v' \ a' \ a'' \ s \rangle \) such

<table>
<thead>
<tr>
<th>Table 2. Lime calculus operational semantics</th>
</tr>
</thead>
</table>

**Reductions**

\[ v \rightarrow v' \]

\[
\begin{align*}
\text{inv}(v, x, P) & \rightarrow a_0[P[v'/x] \mid Q, T, X] \quad (T1) \\
\text{rdv}(v, x, P) & \rightarrow a_0[P[v'/x] \mid Q, T, X] \quad (T2) \\
\text{out}(v') & \rightarrow a_0[P] \mid Q, v \cup T, X \quad (T3) \\
\text{moveb}(v', P) & \rightarrow a_0[P] \mid Q, T, X \quad (T4) \\
\text{react}(v, x, P) & \rightarrow a_0[P] \mid Q, T, X \quad (T5)
\end{align*}
\]

\[
\begin{align*}
\langle v \rangle r_{h}(P[v'/x]) & , T' , X \Rightarrow_{\text{T}} \langle v \rangle r_{h}(\emptyset) , T'' , X \quad (R1) \\
T & \Rightarrow_{\text{T} \{\} } T' \quad (R2) \\
T & \Rightarrow_{\{ \}} T \quad (R3) \\
\Rightarrow: & \\
A , T , X \rightarrow A' , T' , X & \Rightarrow_{T} A' , T' , X \quad (G1) \\
A , T , X \equiv A' , T' , X & \Rightarrow A' , T' , X' \quad (G2)
\end{align*}
\]

The rules are subjected to the following side conditions:

- \( v \leq v' \) (T1) \( T = \text{move}(a, k, T) \)
- \( v \leq v' \) (T2) \( T = \langle v' \ a' \ s \rangle \quad \text{if } T = \langle v \ a \ s \rangle \cup T' \wedge v \leq v' \)
- \( v = \text{mkt}(v', a, h, Q) \) (T3) \( v \leq v' \) (R1) \( a \neq a' \quad \text{if } T = \langle v' \ a' \ s \rangle \cup T' \wedge v \leq v' \)
Table 3. Structure congruence, pattern matching and auxiliary functions

**Structural Congruence Rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \mid Q \equiv Q \mid P$</td>
<td>( (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P ) (SC1)</td>
</tr>
<tr>
<td>$|P| \equiv P |P|$</td>
<td>( P \equiv Q \Rightarrow (\nu x)P \equiv (\nu x)Q ) (SC2)</td>
</tr>
<tr>
<td>$P \mid Q \Pi R \equiv R \Pi Q \mid P$</td>
<td>( (\nu x)(\nu y)P \equiv P \mid (\nu x)Q, \text{if } x \notin fn(Q) ) (SC3)</td>
</tr>
<tr>
<td>$P \equiv Q \Rightarrow a_0[P], T, X \equiv a_0[Q], T, X$</td>
<td>( (\nu x)[aP] \equiv (\nu x)[a_0P], \text{if } x \neq a ) (SC4)</td>
</tr>
<tr>
<td>$(\nu x)[a_0[P] \mid b_0[Q]] \equiv (\nu x)[a_0[P] \mid b_0[Q]], \text{if } x \neq b, x \notin fn(Q)$</td>
<td>( (\nu x)a_0[P], T, X \equiv a_0[P], T, x \cup X, \text{if } x \notin X ) (SC5)</td>
</tr>
</tbody>
</table>

**Pattern Matching Rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leq x \Rightarrow x$</td>
<td>$\nu \gamma \leq \nu' \gamma' \ldots \nu_n \leq \nu'_n$ $\langle \nu_1 \ldots \nu_n \rangle \leq \langle \nu'_1 \ldots \nu'_n \rangle$</td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mkt(\langle a' \ s \rangle, a, h, Q) = \langle a \ a' \ s \rangle$, if $Q \equiv a'_{\nu}[P] \mid Q'$</td>
<td>$mkt(\langle a' \ s \rangle, a, h, Q) = \langle a \ a' \ s \rangle$, otherwise</td>
</tr>
<tr>
<td>$\text{melt}(a, h, {}) = {}$</td>
<td>$\text{melt}(a, h, \langle x \Pi \rangle \cup T) = \langle a \ b \rangle \langle x \ Pi \rangle \cup \text{melt}(a, h, T)$</td>
</tr>
<tr>
<td>$\text{melt}(a, h, v \cup T) = v \cup \text{melt}(a, h, T)$</td>
<td></td>
</tr>
</tbody>
</table>

that $\nu'$ is the value template, $a'$ is the current agent for the tuple to match, $a''$ is the destination agent of the tuple to match and $s$ is its tuple space. Reaction tuples will have the form $\langle \nu' a' a'' s \rangle \langle a b \rangle \langle x \Pi \rangle$ where $a$ is the agent that registered the reaction, $h$ is its location, and $P$ is the reaction's body.

**Reactions.** The second set of three rewrite rules defines the semantics of reactions. In the Lime calculus, reactions are stored in the tuple space, as distinguished tuples hidden from normal user code. Thus to evaluate a reaction we need only have a tuple space as it contains both normal data and the reactions defined over that data. The rules are of the form $T \stackrel{\\sim}{\longrightarrow} T'$ where $T$ is a tuple space and $S$ is the multiset of tuples that are candidates to trigger a reaction. All candidates in $S$ will be examined. When all reactions have completed executing, the new tuple space $T'$ is returned. In the simplest case, if there are no candidates the global tuple space is left as is. If there is a candidate tuple, but it does not trigger any reaction, the rules discard it and proceed to analyze the remaining candidates. Finally, if a reaction matching one of the candidates has been found, then the reaction is removed from the global tuple space. We assume that move commands may not occur in the body of the reaction.
Global computation. The last set of two rewrite rules simply combines the primitive rules with the reaction rules and specifies that after every primitive step, a step of reaction is run.

3.2 Restrictions and Extensions

The semantics presented above can be viewed, in some sense, as an ideal semantics because operations are allowed to operate over the entire federated tuple space and strong atomicity guarantees are enforced throughout. \texttt{Lime}_{\text{imp}} places three additional restrictions on the calculus. Rather than burdening the semantics with those restrictions we will summarize them here.

\textbf{R0} In output and input operations, \texttt{out}(v \ a \ s) and \texttt{in}(v \ a \ d' \ s), x, the tuple space field \( s \) can not be the unspecified value. For output operations, the destination \( a \) can not be unspecified.

\textbf{R1} The body \( P \) of a reaction \( \langle \texttt{react}(v \ a \ d' \ s), x, P \rangle \) is not allowed to contain blocking operations such as \texttt{in} and \texttt{rd}. This restriction prevents reactions from locking up the global tuple space.

\textbf{R2} Lime reaction tuples are selected for activation only if the tuple that triggers the reaction is located on the same host as the agent that registered the reaction. This restriction avoids requiring a lock on the whole federation while a reaction is running.

The semantics has omitted some features of Lime. These can be considered as extensions to the basic Lime calculus semantics.

Probes and group operations. Non blocking input operations and operations that return groups of tuples can be defined easily in the formalism. We did not include them not to burden the semantics and syntax of the calculus.

Weak reactions. \texttt{Lime}_{\text{imp}} includes one kind of distributed reactions, the so-called weak reactions. They are implemented by loosening the atomicity of reactions and introducing an asynchronous step between the identification of the candidate tuples and execution of the reaction. Execution of the reaction remains atomic on the host on which the agent that registered the reaction resides. Weak reactions are implemented by registering a strong reaction on every host in the federation. These strong (local) reactions will notify the host that registered a reaction of the insertion of a matching tuple.

Host engagement and disengagement. We chose not to model engagement and disengagement of hosts. Engagement could be modeled by creating a set of new agents running on a fresh host identifier (defined with \( \langle p \ h \rangle \)). To model disengagement we would have to add a connectivity map that indicate which hosts are connected.
4 Critical Assessment of Lime

During our evaluation we found several inefficiencies in both Lime$_{spec}$ and Lime$_{map}$ which we believe must be addressed if Lime is to gain widespread acceptance. These problems stem from the strong atomicity and consistency imposed by Lime$_{spec}$. Even when weakened in the implementation those requirements make Lime implementations overly complex, full of potential synchronization problems and quite inefficient. Even worse from a user point of view, the cost of the advanced features is paid even by applications that do not use them. Empirical experiments were run on a network of PC with a 400 Mhz Dual Pentium II processor using SunOS 5.6 and Sun's VM of JDK 1.2.2. The machines were connected by a 10 Mbits Ethernet network.

4.1 Reaction Livelocks

Lime$_{spec}$ requires that reactions be executed atomically until a fixed point is reached. All other tuple space operations on the current host are blocked until reactions terminate. This is a heavy price to pay in a highly concurrent setting. Reaction atomicity implies that the runtime cost of a Lime out is entirely unpredictable.

Since reaction bodies are normal programs, termination can not be guaranteed. The following expression react $v, x.($out $v')$ will never terminate as we require that the reaction body reduces to $0$. In Lime$_{map}$ similar issues arise because of the use of unrestricted Java code fragments in reaction bodies. There is a related problem which occurs with the once-per-tuple reactions. A once-per-tuple reaction can trigger itself recursively by outputting the very tuple it is interested in, as in the program react$_{once$p-t}$ $\langle v \ a \ a \ s \rangle$ , $x$.out $\langle v \ a \ s \rangle$. While one may argue that this particular example can be prevented by careful coding, it is much harder to prevent independently developed applications from creating mutually recursive patterns by accident. Non-terminating reactions present a serious problem for Lime$_{map}$. Firstly, they block the entire tuple space of the current host, and since disengagement is global and atomic in Lime, they can prevent disengagement procedures from terminating, thus blocking the entire federation.

4.2 Implementation of Once-per-Tuple Reactions

The semantics of once-per-tuple reactions is that every tuple should be distinguishable from all others so that Lime can ensure that reactions are indeed only triggered once per tuple. In Lime agents can move, taking their tuples with them. The question then becomes: if an agent leaves a host and then comes back, are its tuples going to trigger reactions [5]. Lime$_{spec}$ provides an answer to this question since it requires that every tuple be equipped with a globally unique identifier (GUID). The obvious implementation strategy for once-per-tuple reactions is then to store the GUIDs of the tuples it has already reacted to. One drawback of this approach is that reactions may need to store an unbounded amount of data to remember all tuples seen, especially if GUIDs are made sufficiently
large to provide some reasonable likelihood of unicity. Furthermore, unicity of GUIDs can be difficult to ensure in practice. In LimeRepl, for instance, agents are moved with Java serialization. In this form it is easy to create a copy of an agent along with all of its tuples. To provide real unicity guarantees the implementation would have to protect itself against replay attacks which would complicate considerably the mobility protocols.

4.3 Federated Space Operations

Federated spaces are distributed data structures which can be accessed concurrently from many different hosts. LimeRepl places strong consistency requirements on federated spaces. The challenge is therefore to find implementation strategies that decrease the amount of global synchronization required. The approach chosen by LimeRepl is to keep a single copy of every tuple on the same host as it’s owner agent. Federated input requests are implemented by multicast over the federation. Blocking requests are implemented by weak reactions which register a strong (local) reaction on every host of the federation and a special reaction on the host of the agent that issued the input request. Then whenever one of the local reactions finds a matching tuple the originating host is notified and if the agent is still waiting for input the tuple is forwarded. The problem with this approach is one of scalability. For every federated input operation, all hosts in the federation have to be contacted, new reactions created and registered. Then once a tuple is found, the reactions have to be disabled. From a practical standpoint having additional reactions on a host slows down every local operation as the reactions have to be searched for each output. We argue that federated operations are inherently non-scalable and furthermore that they impact on the performance of applications that do not use them, even purely local applications that do not have to go to the network.

Experiment 1. We compared the use of remote unicast operations against federated operations by a simple program composed of $n$ agents, each running on a different host and owning one integer, that computed the sum of the values in parallel. The results show that the running time of the version using federated operations is 53% to 88% higher for 6 agents. Unfortunately, we were not able to scale the experiment as the current Lime implementation deadlocks after 6 hosts.

Experiment 2. We conducted another simple experiment to assess the impact of remote communication on local operations. In this experiment two co-located agents communicate by exchanging messages over the shared tuple space. At the same time, a number of remote agents located on different nodes communicate via federated operations. We noticed that communication latency increased for local operations as we increased the number of remote hosts communicating, throughput dropped from .38 messages per second to .18 when going from 0 to 6 remote agents.
4.4 Atomicity of Engagement and Disengagement

In \texttt{Lime}_{imp}, hosts joining or leaving a federation must be brought to a consistent state. This boils down to making sure for engagement that all of the weak reactions that hold over the federation be enforced for the new host. For each weak reaction, a strong reaction must be registered on the incoming host. For the disengage operation, all weak reactions registered by agents currently on that host from all other hosts in the federation have to be de-registered. Since both operations are atomic it means that tuple operations are blocked while hosts are being added or removed from the configuration. Furthermore, one may question the choice of requiring explicit disengagement notification in the context of mobile devices. If a mobile device moves out of range or loses connectivity, it is not likely that it will have the time to send a message beforehand.

5 Back to Basics: CoreLime

The initial goal of our research was to add security primitives to \texttt{Lime}, but the problems that we detected while trying to understand its implementation convinced us that we had to simplify the model. Our approach is twofold, first we will provide a simpler incarnation of \texttt{Lime} that we call CoreLime which is a non-distributed variant of \texttt{Lime} with agent mobility. The syntax and semantics of most \texttt{Lime} operations is retained, the main restriction is that operations are scoped over the local host only. The second part of our research will be to define semantics for the remote operations provided in \texttt{Lime}. For these we plan to give a translation to CoreLime using agent mobility to specify remote effects.

5.1 Semantics of CoreLime

The main difference between \texttt{Lime} and CoreLime is that we tried to lift all global synchronization requirements. To do so we have restricted all operations to their local variant and rely on agent mobility as the single mechanism for modeling remote actions. A further change to \texttt{Lime} is that we removed the atomicity requirement on reactions. In our variant, reactions execute concurrently to user code. This allows for a much simpler semantics without the need for auxiliary reductions. The semantics is summarized in Table 4.

The main changes required are the following. Input operations must check the location of tuples matched, any tuple retrieved by an \texttt{in} or \texttt{rd} must belong to a co-located agent. This constraint is enforced by the side condition on the transitions, where the auxiliary function \texttt{loc} returns the host where a tuple or an agent is located. Output and move reduction can trigger reactions, these are represented by a new process \texttt{R} running in parallel. The auxiliary function \texttt{react} will create a single new agent on the current host with as body the parallel composition of matching reactions. This is done for each matching tuple \texttt{v} such that \texttt{v} is substituted for the parameter of the reaction body.
Table 4. Semantics of CoreLime.

Reductions

\[ a_0[ \text{inv} \, x.\ P \,|\ | \, Q \,|\ , \, x' \, \cup \, T , \, X \, \rightarrow \, a_0[ P[x'/x] \,|\ | \, P' \,|\ | \, Q \,|\ , \, T , \, X ] \ (T1) \]
\[ a_0[ \text{rdr} \, x.\ P \,|\ | \, Q \,|\ , \, x' \, \cup \, T , \, X \, \rightarrow \, a_0[ P[x'/x] \,|\ | \, P' \,|\ | \, Q \,|\ , \, x' \, \cup \, T , \, X ] \ (T2) \]
\[ a_0[ \text{out} \, x' \,|\ | \, Q \,|\ , \, T , \, X \, \rightarrow \, a_0[ P ] \,|\ | \, Q \,|\ , \, v \, \cup \, T , \, X ] \ (T3) \]
\[ a_0[ \text{move} \, h'.\ P \,|\ | \, Q \,|\ , \, T , \, X \, \rightarrow \, a_0[ P ] \,|\ | \, Q \,|\ , \, R \,|\ , \, T' \,|\ , \, X ] \ (T4) \]
\[ a_0[ \text{react} \, v,\ x.\ P \,|\ | \, Q \,|\ , \, T , \, X \, \rightarrow \, a_0[ P ] \,|\ | \, Q \,|\ , \, \langle v \, \langle a \, h \rangle \, \langle x \, P \rangle \rangle \ | \, T \,|\ , \, X ] \ (T5) \]

The rules are subjected to the following side conditions:

(1) if \( v \leq v' \land \text{loc}(v') = h \)
(2) if \( v \leq v' \land \text{loc}(v) = h \)
(3) \( v = \text{mkt}(v', a, h), R = \text{react}(\{v\}, h, T) \)
(4) \( T' = \text{mut}(a, h', T), R = \text{react}(\text{sel}(a, T, T'), h', T') \)

Functions

\[ \text{mkt}(v \, a' \, s), a, h = \langle v \, a' \, s \rangle, \text{if loc}(a') = h \]
\[ \text{mkt}(v \, a' \, s), a, h = \langle v \, a \, s \rangle, \text{otherwise} \]
\[ \text{react}(\{\}, h, T) = 0 \]
\[ \text{react}(v \cup V, h, T) = (v r) \, \text{r}_h[\text{sel}(v, h, T) | \text{react}(V, h, T) \]
\[ \text{sel}(v, h, \{\}) = 0 \]
\[ \text{sel}(v, h, \langle v' \, (a' \, P') \rangle \cup T) = P[v/x'] | \text{sel}(v, h, T), \text{if } v \leq v' \land \text{loc}(v') = h \]
\[ \text{sel}(v, h, v' \cup T) = \text{sel}(v, h, T) \]
\[ \text{mut}(a, h, \{\}) = 1 \]
\[ \text{mut}(a, h, v \, a' \, s \cup T) = \langle v \, a' \, s \cup \text{mut}(a, h, T), \text{if loc}(a') = h \]
\[ \text{mut}(a, h, v \, a \, s \cup T) = \langle v \, a \, s \cup \text{mut}(a, h, T), \text{if loc}(a') = h \]
\[ \text{mut}(a, h, v \, a' \, h' \cup (x \, P') \cup T) = \langle v \, a \, h' \cup (x \, P') \cup \text{mut}(a, h, T) \]
\[ \text{mut}(a, h, v \cup T) = v \cup \text{mut}(a, h, T) \]
\[ \text{sel}(a, T, T') = \{v \in T' | (a ? a ?) \leq v \lor (v' \, a' \, s) \leq \text{v} \land (v' \, a' \, s) \in T \} \]

Remote operations. Removing remote operations from the semantics does not prevent an agent from accessing tuple spaces of remote hosts. To exemplify this, we show how an agent can dispatch a new agent \( r \) to another host \( h' \) to perform a remote \( \text{in} \). When a matching tuple is found, the agent \( r \) returns to the issuing host \( h \) and outputs the value found with a tag \( y \) that identifies the operation.

\[ a_0[ \text{rin} \, h'.\ (v' \, a' \, s'), x.\ P \,|\ | \, P' ] \triangleq (v y) \, (v r) \, a_0[ \text{in} \, (y, ?), a, a, \delta, x.\ P \,|\ | \, P' ] \]
\[ \text{n}_h[ \text{move} \, h'.\ \text{in} \, (v', a', \delta), x.\ \text{move} \, h.\ \text{out} \, (y, x), a, \delta] \]
5.2 Security Extensions for CoreLime

The security extensions described here are being incorporated in our implementation of CoreLime. Unless otherwise specified we assume that the LineImp interfaces have been retained. We propose two new mechanisms for adding support for security into Lime. The first is a capability-based access control model for controlling access to tuple spaces. The second is an extension of the concept of reactions called filters.

Fine-grained access control. Fine-grained access control mechanisms provide applications the means to control access to shared tuple spaces. For instance, it may be desirable to restrict some applications to have write-only access, i.e. they can add tuples to a space but not remove them, or conversely, they may be allowed to input tuples but not to add new ones. Reactions may be even more sensitive than other operations as a process that can register reactions may inspect all tuples put in the space. The central idea is one inspired from Cardelli's Ambients in which ambient names behave as capabilities and can be given out in restricted form.

In CoreLime, each tuple space has a name, an instance of the abstract interface CoreLimeName, see Figure 1. Names can always be compared for equality, with the same purpose as in Lime, that is, spaces with the same name can be shared. But in addition names are used by CoreLime to implement access control. Each name acts as a capability and is checked to validate every operation. CoreLime provides two implementations for CoreLimeName, one is the class LimeName which is provided for backwards compatibility with Lime programs. Instances of LimeName contain a string and do not implement any access control.

```java
interface CoreLimeName {
    boolean isEqual(CoreLimeName); // Compare two names.
    CoreLimeName clone(); // Return a name with same rights.
    void forbidRead(); // Remove the right to remove tuples from space.
    void forbidWrite(); // Remove the right to insert tuples in space.
    void forbidRegister(); // Remove the right to register reactions.
    void forbidClone(); // Remove the right to clone this name.
}

final class LimeName implements CoreLimeName {
    LimeName(String name);
}

final class CoreLimeCapability implements CoreLimeName {
    CoreLimeCapability();
}
```

Fig. 1. CoreLime Capability interface.

The class CoreLimeCapability implements the full functionality of the interface CoreLimeName. Each capability contains a globally unique identifier generated when the object is created, and is hidden from user code. The isEqual
method compares GUIDs for equality. Since user code can not see the name of the tuple space, in order to create shared spaces there must be some name exchange protocol in place. The agent that creates a name will hand out copies of that name to other agents after having set the access rights associated to the name to the proper level. For instance, the following code fragment creates a new name and outputs a copy of that name which can not be further distributed and which forbids writes to the tuple space.

```java
CoreLimeName privateKey = new CoreLimeCapability(); // unrestricted name
CoreLimeName publicKey = privateKey.clone();
publicKey.forbidClone(); // publicKey can not be copied
publicKey.forbidWrite(); // publicKey can not be used to out
templ = new Tuple();
templ.addActual(publicKey);
tuple.out(templ);
```

The implementation of CoreLime ensures that non-cloneable keys can not be copied and that only one instance can be serialized. Whenever names are serialized the CoreLime implementation will digitally sign the name to prevent tampering during transit. The interface to names only allows capabilities to be removed, thus if we write `publicKey.forbidRegister()` it means that this particular capability will never again allow a reaction to be registered for the particular tuple space.

**Security filters.** Security filters are a special kind of reaction that can be installed on a tuple space to perform some actions at each Lime operation. The goal of filters is to allow even finer-grained security policies to be coded such as policies that look at the values of tuples being inserted or extracted from the space. Filters get as input the tuple given as argument to the Lime operation and may return a modified tuple or null if the operation should fail. Multiple filters can be defined for the same space and operations. They will be chained in an implementation specific order.

Consider for example an output filter that checks that first field of every output tuple bears an agent name.

```java
class AgInFilter extends CoreLimeFilter {
  ITuple filterOutput(ITuple val) {
    if (val.get(0) instanceof AgentLocation)
      return val;
    else return null;
  }
}
```

Further checking can be enforced by adding another filter that appends the name of the agent that produced the tuple to each value being output.

```java
class AccFilter extends CoreLimeFilter {
  ITuple filterOutput(ITuple val) {
    val.addFormal(CoreLime.getCurrentAgentName());
    return val;
  }
}
```
The above filters can be chained in the following manner:

```java
ts.registerFilter(new AccFilter());
ts.registerFilter(new AgNmFilter());
```

Filter expression are owned by agents, just as reactions. They move with them and are merged when a space is transiently shared.

One important difference between reactions and filters is that filters can be applied to input expressions as well as to the registering of reactions. For example, it is possible to enforce that an agent will only be able to remove the tuples it has inserted with the following input filter:

```java
class InputAccFilter extends CoreLineFilter {
    ITuple filterIn(ITuple val) {
        val.addFormal(CoreLine.getCurrentAgentName());
        return val;
    }
}
```

This filter is symmetric to the AccFilter in that it appends the name of the agent to all input requests. The class CoreLineFilter implements the basic functionality of filters which is to let every value through unchanged. Subclasses need only override the methods for which some behavior is needed. The interface of the filter abstract class is:

```java
abstract class CoreLineFilter {
    ITuple filterIn(ITuple);
    ITuple filterOut(ITuple);
    ...
    Reaction filterRegister(Reaction);
}
```

6 Conclusions

This paper revisited Lime, a middleware communication infrastructure for mobile computation that addresses physical mobility of devices and logical mobility. We have argued that the original Lime specification is costly and complex to implement. We have proposed a smaller and lighter variant of Lime, which we call CoreLime, that has none of the global synchronization and atomicity requirements of Lime. Furthermore we have presented the access control mechanisms built into our extension the Java implementation of Lime. We are currently working towards a production quality implementation of CoreLime. In future work, we plan to translate the distributed features of Lime into CoreLime.

Acknowledgments. This research was supported by a grant from Motorola, CERIAS and CAPES. We wish to thank the authors of Lime for answering our questions.
References

Dynamic Adaptation of Mobile Agents in Heterogenous Environments

Raimund Brandt and Helmut Reiser

1 skyguide
Postfach 1518, 8058 Zürich-Flughafen, Switzerland
raimund.brandt@skyguide.ch
2 Munich Network Management Team,
University of Munich, Dept. of CS, Oettingenstr. 67, D-80338 Munich, Germany
reiser@informatik.uni-muenchen.de

Abstract. Mobile agents must be prepared to execute on different hosts and therefore in different execution environments. Even when a homogeneous execution environment is offered by abstracting the underlying heterogeneity, there are scenarios like IT-management, where mobile agents are forced to contain environment dependent implementations. The aim of this work is to equip mobile agents with a flexible capacity to adapt to a range of different environments on demand.

We discuss different forms of adaptation and draw a distinction between static and continuous forms. Our solution for dynamic adaptation provides a concept for exchanging environment dependent implementations of mobile agents during runtime. Dynamic adaptation enhances efficiency of mobile code in terms of bandwidth and scalability.

1 Introduction

Due to their increasing size and accelerated growth today’s computer networks have a complex and heterogenous structure. Code mobility seems to be a promising approach to keep the advantages of such networks and to overcome some of its disadvantages. Code mobility can be defined as the capability to dynamically change bindings between code fragments and the location where they are executed [CPV97]. Fuggetta et al. [FPV98] give an overview of the existing technologies, design paradigms and applications of mobile code. Code mobility can also be described as motion of executable code over networks towards the location of resources that are needed for the execution. Therefore, mobile code must cope with changing environments because of its motion between hosts. A special design paradigm of mobile code is the mobile agent paradigm [FPV98]. Mobile agents can be defined as programs that act autonomously on behalf of a user and travel through a network of heterogenous machines. Therefore, mobile agents can be faced heavily with the problem of frequently moving in heterogenous environments.

* This work was completed while the author was working at the Munich University of Technology.
The discrepancies between heterogeneous environments can be alleviated by introducing common abstractions, such as those implicit in operating systems (file systems, etc.), virtual machines (Java core libraries) or runtime systems for mobile agents [LO98]. This implicates an abstraction of resources. In certain cases this representation of resources is not sufficient. On one hand because only a subset of resources can be abstracted or on the other hand if a resource is abstracted only a subset of functionality is accessible. Thus, mobile agents have to include under certain circumstances environment dependent code which is only needed in a few environments but moved over the network to all machines.

For instance, a scenario can be found in the field of network management where mobile agents seem to be a promising technology [BPW98,FKK99]. Java programs can even be executed on small devices, e.g., using Java Micro Edition (J2ME), which come in a particular variety of concrete forms and can overcome the heterogeneous character of such devices. Nevertheless, the execution environment may differ due to the type of the JVM and the available resources which supports only a restricted subset of functionality. On the other hand the same code may be executed on workstations offering a JVM with full functionality. This discrepancy can lead to mobile code providing environment dependent logic. Another example for a mobile agent carrying environment dependent code, is an end–to–end Quality of Service (QoS) management where agents may roam to prepare various and heterogeneous network equipment to conform to central policies, whose enforcement however is environment–specific [e.g., via different vendor APIs or via special operating system dependent calls]. The illustrative example which we use throughout this paper is a much simpler scenario: The configuration of a set of distributed web clients in a heterogeneous environment. We are aware of the fact that there are some non–agent tools for the configuration of remote web clients and it is not compelling to use mobile agents. However, our example is very intuitive and we have chosen it as an efficient means to display all of the characteristic features of our approach.

Such a mobile agent can not be implemented purely in Java. For example, the configuration of the default web browser in Windows NT is based on entries in the registry database and not accessible through the Java API. Apart from the operating system the configuration also depends on the kind of web browser. A conventional approach to implement such a mobile agent might be an if-then-else construct with conditional branches which are interchangeably executed depending on the environment. In the following this solution is denoted as static customization. Under certain circumstances static customization is not very efficient. Because of the hard coded relation between the number and nature of environments and the executable code that supports the environments, the maintenance and scalability of the mobile agent is restricted. The second inefficiency is the transport of environment dependent code through the network. The environment dependent code might only be used on a few machines. But especially in the case of a mobile agent the rarely needed code must be carried over the whole route along with the mobile agent.
The intention of this work is to offer a methodology for creating a mobile agent which is able to adapt itself to the environment where it is currently running. The variation is not achieved simply by entering an appropriate section of code, but by composing an environment-specific version of the agent that assembles only appropriate constituents. The result leads to a slimmer version, and to less movement of code across the network. This includes a concept for exploring the environment and the dynamic exchange of code parts as needed in order to work properly in the detected environment. The exchange of code parts is carried out without termination of the mobile agent. This technique is denoted as *dynamic adaptation*. It is intended to improve mobile agents by limiting the transport of code which is actually needed.

After discussing the term adaptation and the state-of-the-art of adaptation in section 2, the proposed concept for dynamic adaptation is presented in section 3 followed by a short overview of a prototypical implementation in section 4. This system implements a configuration management architecture using mobile agents for configuring web browsers. Section 5 investigates under which circumstances mobile agents can benefit from dynamic adaptation; the last section concludes the paper and discusses future work.

2 State-of-the-Art of Adaptation

Adaptation techniques as found in literature are used within different contexts. The techniques differ in the objectives, which are addressed and in the methodologies to meet the targets. In this section we investigate two margins on a wide scale, labeled static and continuous adaptation. We place our own solution conceptually somewhere in between them and investigate these other adaptation mechanisms to learn which of their concepts could be used fruitful in our dynamic adaptation approach. As a result we will adopt two basic ideas named reconfiguration and context awareness.

2.1 Static Adaptation

Reuse of code is a field where adaptation is mostly applied. It is especially used in component based software engineering (CBSE) [Hei99]. One of the benefits of CBSE is the reuse of existing code and components respectively. The goal is to reduce programming to the wiring of components. Even if components are available for arbitrary functionality, it is probable that not every component fits together with another component or fits into an application because interfaces change over time (software evolution). The reasons therefore can be e.g., syntactical incompatibility or semantic differences of the interfaces. In order to use incompatible components, adaptation can be used to modify the incompatible parts of code in such a way that they fit together. This kind of adaptation used in the field of CBSE can be denoted as *static adaptation* because it is in general applied before compilation time and not during runtime. The input of static adaptation is a component $C$ and a description of the desired modifications.
The output is the modified component $C'$ which fits into the designated application. In [Hei99] a survey and evaluation of component adaptation mechanisms is presented. Some examples can be found in [DH99,KH98,GK97].

Though static adaptation concepts in general do not provide support for adaptation during runtime as needed for dynamic adaptation. However they have as a common element the process of re-configuration where source or executable code is modified or exchanged.

2.2 Continuous Adaptation

For some applications it is important to adjust service parameters to performance degradation of the underlying resources. For instance, multimedia applications which use unreliable connections, e.g., wireless communication or Internet, must modify the representation of data according to the conditions of the network in order to deliver usable results. Changes of the resource conditions may occur without following a certain pattern or any other regularity. The modification of a running application is done by tuning parameters. Such modifications are here denoted as continuous adaptation. In contrast to static adaptation which focuses modification of code continuous adaptation is a totally different approach.

The triggers for the continuous adaptation are continuously changing conditions of resources. For continuous adaptation the resources are monitored and the adaptation process is initiated as the resource conditions change [GBSH00]. The input for continuous adaptation is a running application relying on frequently and strongly changing resources and classes of resource or Quality of Service (QoS-) parameters. The result of the continuous adaptation is the modification of parameters steering the resource usage, data processing or data presentation [Nob00,ADOB98,STW92]. The work presented in this section rather deals with different problems involved in manipulating parameters than dynamic adaptation which exchanges executable code. The common object of continuous adaptation and dynamic adaptation is the detection of the environment in order to determine the appropriate adaptation, also denoted as context awareness. Since information retrieval from the environment is based on application specific sensors, like active badges [WHFG92], and not on generic properties of the hardware and software configuration, as targeted by this work, no particular concept of context awareness is applicable for dynamic adaptation.

3 Framework for Dynamic Adaptation

As introduced in section 1 dynamic adaptation offers a technology for creating mobile agents which are able to adapt themselves to the environment where they are currently running. Starting from this point a methodology for developing adaptable agents and a framework supporting the process of dynamic adaptation will be designed. Adaptation of mobile agents occurs without termination of the agent. The trigger for dynamic adaptation is the movement of code. If the core mobile agent moves to a new host, the dynamic adaptation procedure is
initiated. The input of dynamic adaptation is a set of environment dependent implementations, an environment independent small core agent and a description of the current environment. The result of dynamic adaptation is the selection of the right implementation for the environment and the linking of the selected implementation into the core. Dynamic adaptation differs from static adaptation not only concerning the time of adaptation, but also concerning the adaptation function. Dynamic adaptation selects an implementation from an existing set of environment specific implementations, exchanges code and instantiates code dynamically. Static adaptation transforms existing code into new code.

![Fig. 1. Generic Concept for Dynamic Adaptation](image)

The mobile agent is divided into several environment dependent adaptable parts (s. figure 1, gray colored X, Y) and a small environment independent non-adaptable core. The adaptable parts are exchanged in order to fit into the current environment. The environment independent core and the environment dependent adaptable part form the mobile agent executing its task on a host. The agent programmer develops the core and might also develop the environment dependent parts. However, adaptable parts are normally built by a component developer. The movement from one host to another is done by the small core agent as a vehicle for the computational flow. The core can be used as bootstrapper for the dynamic adaptation. After the arrival on a new host adaptation is applied delivering the mobile agent with its full functionality. Before the mobile agent moves to a new host, the environment specific implementation is dropped and the mobile agent is reduced to the small environment independent core. Thus, only code which is actual needed on every host is moved over the network. Since dynamic adaptation exchanges code and does not tune parameters like continuous adaptation, concepts of continuous adaptation are not applicable to dynamic adaptation. However, the idea of exploring the environment — which
we call context awareness — is taken from continuous adaptation and can be
used in a similar way for dynamic adaptation.

In the following the architectural parts of the framework and the methodology
will be explained. The development tools supporting the building process of
adaptable agents will be presented in section 4.

3.1 Components of Dynamic Adaptation

As learned from static adaptation and continuous adaptation, components for
reconfiguration and context awareness are needed. Thus, the generic architec-
ture must be extended by these two components. The core agent uses an adaptor
for identifying, loading and integrating environment specific methods into the
mobile agent. These adaptors include the context awareness module and the
reconfiguration component. Figure 2 gives an overview of the life-cycle of the
agent including reconfiguration and context awareness. After arriving on a host
the core is running in an environment from which it does not know what hard-
ware, operating system, etc., is used (1). The context awareness component is

![Detailed Concept for Dynamic Adaptation](image)

responsible for the inspection of the environment. It must know which environ-
ment dependent values are important for implementations and how they can be
deduced. In section 3.3 we will see that each environment specific implementa-
tion provides a description of its desired environment. The description can be
extracted from the implementations.

A new detail in this concept is the repository serving environment dependent
implementations and their descriptions. The repository service is used by the
context awareness component to retrieve implementation descriptions (2) called
profiles. With this profiles the context awareness module is able to determine
the execution environment where the core is currently running in. This result
is delivered to the reconfiguration component (4) which loads the appropriate
implementation for the current environment (5) from the repository and links
the implementation into the core (6).
3.2 Reconfiguration

Although context awareness is executed before reconfiguration, the reconfiguration component will be explained first because it determines the structure of the core and the implementations and helps to understand the context awareness concept. As set up in the requirements analysis in [Bra01] the linking of the implementations into the core must be done without termination of the mobile agent and with a high level of transparency to the core. This implies for instance that adaptation must not be initiated by the core. Another requirement is the transparent invocation of methods.

From this requirements an OO design pattern is derived, which must be followed by the agent programmer developing mobile agents using dynamic adaptation as presented in this work. Note that this limits the application area of dynamic adaptation to OO technology. The use of several environment dependent implementations which can be mapped is known as strategy pattern [GHJV95] which can be implemented through an abstraction by interfaces. The agent programmer must define an environment independent interface which is implemented by all implementations providing the same functionality but for different environments. The environment independent interface is denoted as functionality interface and an environment dependent class implementation is named implementation class. The functionality interface is specified by the agent programmer and the implementation classes for different environments are developed by component developers. Classes implementing the same functionality interface form an implementation group.

The connection between implementation classes and the core is realized by an adaptor class. The adaptor class, a kind of a stub with additional functionality, is used within the core instead of implementation classes. It initiates adaptation and delegates method calls to the currently loaded implementation class. Code

![Diagram](image.png)

**Fig. 3.** Adaptor Class for IMemory Functionality Interface
for the adaptor can be generated from the functionality interface description, like CORBA stubs \cite{OH98} are generated out of IDL interfaces. We provide such a generator as described in section 6.

In the example of the browser configuration, the mobile agent needs to acquire system information like the size of physical memory. For this information retrieval operating system and CPU architecture specific implementation classes are needed. There are environment dependent implementation classes for every supported environment. The agent programmer must declare the functionality interface `IMemory` declaring the method `getPhysicalMemory()` which is implemented by all implementation classes, delivering physical memory size. Figure 3 depicts the usage of the adaptor class in the example application. The adaptor `IMemoryAdaptor` is used in the core of the mobile agent for accessing information about the memory. This adaptor is generated out of the functionality interface `IMemory` by the adaptor generator. The core moves without implementation classes, but with the adaptor over the network. When it comes to a host, e.g. a PowerPC running AIX, the adaptor initiates adaptation by calling context awareness and reconfiguration which loads the suitable class, in this case the implementation class `Memory_PPC_AIX`.

\section*{3.3 Context Awareness}

The function determining the name of the concrete implementation class for the environment where the mobile agent is currently running is done by the context awareness component. The result of context awareness function is a description of the environment and the environment dependent attributes. The difficulty is that only the component developer, which implements environment dependent implementation classes, knows what environment dependent attributes his implementation needs. To solve this problem we introduced \textit{profiles}.

With each implementation class exactly one \textit{implementation profile} is associated, specified and implemented by the component developer. This profile is loaded and executed in the current environment where the mobile agent is running. The result of the execution of an implementation profile is an \textit{environment profile} which can be used to decide which implementation class can be used in the detected environment.

It is important to realize that profile information, while strictly belonging to implementation classes, should be kept apart from them in terms of object structure, because of the stages involved in the decisions taken during the adaptation process: Profiles have to be acquired at a new site, in order to determine whether implementation classes have to be brought in as well. Hence, the profiles act like (small) probes that precede (optional) migration of (larger) implementation classes over the network as the mobile agent moves between different hosts.

The implementation profile includes several \textit{profile values} and code to calculate this values.
Profile value stand for a certain environment property, such as installed operating system or CPU architecture. The profile value includes methods to retrieve the actual value from the environment. We call this code generating function. To compare profile values with the value requested by the implementation class, we use other methods and call them matching function, which are also part of the implementation profile.

For instance an implementation class which has the functionality to configure Netscape running on an X86 with WindowsNT would have an implementation profile like depicted in figure 4.

Executed on a PowerPC running AIX and Netscape as default web browser it would generate the environment profile values for the environment through the generating function as shown in figure 5. After comparing the profile values of the implementation class and the profile values of the environment the context awareness concludes that the implementation class Configuration.X86_WINNT.NETSCAPE is not suitable for the environment because the CPU architecture and the operation system does not match. The profile of another implementation which implements the same functionality interface must be found and executed.

This is a very simple but expressive example. The profile values are simple attributes which can be deduced relatively easy. The generating function can be simple too, such as comprising a call to System.getProperty("os.name") in Java. However, the concept is also useful for more complicated configuration tasks. An adaptable Agent configuring, e.g., an SAP application might need implementation profiles including ABAP calls to determine specific SAP parameters.

4 Implementation of a Configuration Management Agent

After the presentation of the architecture providing dynamic adaptation for mobile agents this section deals with the implementation of the adaptation framework and a mobile agent configuring browsers. The implementation of the adaptation framework is independent of the mobile agent's configuration task and independent of the agent system. The configuration of the browser relies on the adaptation mechanism. It implements the configuration of a set of web browsers running on various operating systems and CPU architectures. In our implementation, the configuration is brought to the different hosts by the mobile
agent using the Voyager agent system platform [Obj00]. Since the mobile agent is relying on the adaptation framework, it will be described first and then we will continue with the implementation of mobile agent.

4.1 Adaptation Framework

The basis for the adaptation framework is Java. It offers useful functionality for dynamic adaptation, e.g., dynamic class linking, and reflection. The adaptation framework includes three components. Two stand alone applications – the adaptor generator and the repository – and a set of classes which are integrated into the mobile agent through the adaptors including functionality for context awareness and reconfiguration. Further classes are provided as profiles and profile values for the mobile agent programmer to describe the designated environment of the implementation class. Figure 6 gives an overview of the components involved into adaptation by the example of configuring disk and memory cache of a browser.

---

**Fig. 6.** Overview of the Adaptation Architecture for the Configuration Management Agent

The adaptors are generated by the adaptor generator presuming that the adaptation design pattern has been followed by the mobile agent programmer. That means the adaptable parts are realized as implementation classes and the functionality interface between the core and the adaptable parts is described as a Java interface. The adaptor generator reads the Java byte code of the interface, i.e., a Java class, and produces the adaptor class in Java source code. The adaptor class is used in the core instead of the implementation classes. By convention the adaptor class name is derived by the adaptor generator from the functionality interface name:
<interface name> → <interface name>Adaptor

The adaptor class implements the methods as declared in the interface. The body of the method implementations contains the adaptation and the delegation of the method call to the instance of an implementation class. The adaptation includes the context awareness module and reconfiguration component. The name of the implementation class is resolved by the context awareness module and the right implementation class is loaded by the reconfiguration component. The actual method is executed by the instance of the loaded implementation class. Since the adaptor generator needs to retrieve the interface name and the method declarations from the interface, it introspects the interface by using Java reflection.

As depicted in figure 6 the methods setDiskCache() and setMemoryCache() which have been declared in the functionality interface IConfiguration are implemented by the adaptor class IConfigurationAdaptor. The adaptor class contacts the adaptation, realized by two classes, ContextAwareness and Loader, for loading the right implementation class.

Assuming Configuration.XY is the right implementation class, for the current environment where the core is running, the method calls, setDiskCache() and setMemoryCache(), from the core are delegated by the adaptor class to the instance of implementation class Configuration.XY.

The context awareness is realized by the class ContextAwareness which loads the implementation profiles of all implementation classes over the network and executes them. The execution of the implementation profiles includes the generation of environment profiles and the comparison of the profile values. The implementation profiles are served by the repository to the context awareness. The implementation profile is realized as a Java class containing the set of profile values. A profile value is also represented by a sub class of the abstract class ProfileValue.

In figure 7 the hierarchy of the profile values used for the operating system are depicted. From the abstract super class ProfileValue the concrete class OperatingSystem is derived implementing the method getEnvProfileValue for retrieving the name of the operating system in the current environment. The class OperatingSystem represents a type of a profile value. For instance CpuArchitecture and DefaultWebBrowser might be other profile value types needed by the implementation class descriptions in the example of the configuration for the browser. The classes Linux, AIX, HP-UX, Solaris, which are grouped together as Unix flavors, and UNIX are classes that are used by the programmer of the implementation classes (component developer) describing the necessary environment. The properties of the profile values can be mapped into the OO hierarchy as shown in the case of Unix. The component developer simply uses the class UNIX if the...
implementation class is suitable for the Unix flavors. The comparison of the implementation and environment profile values is done by comparing the super classes of the profile values.

The implementation profile and the profile values must be integrated into the implementation class by the component developer. Every implementation class includes a method `getProfile()` which delivers the profile values. The body of this method realizes the environment description of the suitable environment. Figure 8 gives an example for an implementation class suitable for a x86 host running WindowsNT and Netscape as configured default web browser.

```
public Profile getProfile() {
    Profile result = new Profile();
    new ProfileValue()
        new WindowsNT(),
        new X86(),
   );
    return result;
}
```

**Fig. 8. getProfile()**

The loading of the implementation classes is done by a modified Java class loader. The class `Loader` loads the implementation class according to the class name delivered from the context awareness. The implementation class is loaded by the `Loader` class from the repository through the the class `RepositoryClient` (s. figure 6). The same class is used by `ContextAwareness` for communication with the repository.

In the current implementation the repository is a stand alone application serving the profiles and implementation classes. For keeping the autonomy of the mobile agent the chosen repository concept provides proxy repositories which are started a long the route of the mobile agent. This provides still high level of autonomy and keeps the possible communication overhead caused by adaptation relatively low.

We distinguish between central repository and proxy repositories. Communications between a mobile agent and a repository should be "sufficiently local" to make efficient use of bandwidth. For this purpose a neighborhood metric can be defined depending on the application scenario. Using this metric the agent can determine the "nearest" repository, with e.g., one repository proxy serving per subnet.

### 4.2 Mobile Agent for Configuration Management

For the example application using dynamic adaptation, a mobile agent has been designed for the configuration of the default web browser. The task of the mobile agent is to visit a set of workstations, to retrieve local system information (physical memory, free disk space) and according to this information to change the parameters of the default web browser. This includes the setting of memory cache size and disk cache size. Adaptation is needed for the information retrieval which must be done in a system, operating system and CPU architecture specific
way and cannot be implemented in pure Java. Further on, adaptation is used for setting parameters of the browser. The setting depends on the browser and the operating system.

The core mobile agent is implemented in pure Java using Voyager [Ob99] as agent system. Following the adaptation design pattern the functionality interfaces IMemory (retrieving physical memory), IDisk (retrieving free disk space) and IConfiguration (setting the browser parameters) have been declared. The adaptor generator creates the according adaptor classes out of the functionality interfaces IMemoryAdaptor, IDiskAdaptor and IConfigurationExceptionAdaptor. A set of implementation classes for each functionality interface has been written supporting various environments like WindowsNT/x86, AIX/PowerPC, Linux/x86 and browser Netscape and Internet Explorer. The exact choices foreseen depended on available platforms in our test lab.

5 Evaluation of Dynamic Adaptation

Dynamic adaptation promises a reduction of used bandwidth by paying runtime overhead due to context awareness and loading of implementation classes. Therefore, the dynamic adaptable configuration management agent has been compared against a monolithic agent with the same functionality. The monolithic agent transports the whole code for all environments and implements static customization with if-then-else statements for the different environments. For determining the gain of bandwidth the size of code, which is moved over the network, has been measured.

The monolithic agent is built from the core of the dynamic adaptable agent plus statically linked implementation classes. Therefore, the amount of code which is moved over the network for running both agents differs only concerning the size and number of implementation classes and profiles (profiles are only needed for dynamic adaptable agents not for the monolithic agent). Following this considerations, the code size of implementation classes (inclusive dynamic libraries for eventual needed native code) and the profiles have been measured (see Table 1).

As explained above the dynamic adaptable agent has to load all profiles for context awareness but only one single implementation class for execution. Whereas the monolithic agent has to move without profiles since it uses simple if-then-else statements for environment detection but has to carry all implementation classes.

For determining the code size, which is specific for the monolithic agent, the sum of all implementation classes is calculated (see figure 9). To determine the average code size, which is specific for the dynamic adaptable agent, the sum is calculated of all profiles (in the formula \( k \) denotes the number of implementation groups) and the average size of implementation classes belonging to one implementation group. As from each implementation group one implementation class is loaded over the network the average size of the implementation classes has been chosen to get a mean value for an implementation class over all en-
### Table 1. Size of executable code

<table>
<thead>
<tr>
<th>Implementation group</th>
<th>size of serialized profiles [byte]</th>
<th>environment</th>
<th>size of implementation class [byte]</th>
<th>size of dynamic library [byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMemory</td>
<td>1046</td>
<td>AIX, PPC</td>
<td>1724</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux, X86</td>
<td>1788</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WindowsNT, X86</td>
<td>978</td>
<td>18209</td>
</tr>
<tr>
<td>IHarddisk</td>
<td>1052</td>
<td>AIX, PPC</td>
<td>2415</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux, X86</td>
<td>2599</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WindowsNT, X86</td>
<td>1090</td>
<td></td>
</tr>
<tr>
<td>IDefaultWebClient</td>
<td>891</td>
<td>Unix</td>
<td>1276</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WindowsNT, X86</td>
<td>1069</td>
<td>19785</td>
</tr>
<tr>
<td>IConfiguration</td>
<td>1408</td>
<td>Unix, Netscape</td>
<td>2601</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unix, Lynx</td>
<td>2335</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WindowsNT, Netscape</td>
<td>2781</td>
<td>20187</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WindowsNT, IExplorer</td>
<td>1392</td>
<td>19785</td>
</tr>
</tbody>
</table>

code size (monolithic agent) =
\[
\sum_{i=1}^{l} sizeOf(IMemory_i) + \sum_{i=1}^{m} sizeOf(IHarddisk_i) + \sum_{i=1}^{n} sizeOf(IDefaultWebClient_i) + \sum_{i=1}^{o} sizeOf(IConfiguration_i)
\]

with

\(l, m, n, o\) number of environments supported by respective implementation group

*Fig. 9. Code size of monolithic agent*

environments (cf. figure 10). The result of the comparison is as expected a lower code size in the case of the dynamic adaptable agent (11526[byte]) than in the case of the monolithic agent (22323[byte]).

The costs for gained bandwidth is a runtime overhead, which consists of two parts: the execution of context awareness and the time for loading the implement-

code size (dynamic adaptable agent) =
\[
\sum_{k=0}^{l} sizeOf(profile_k) + \sum_{k=1}^{m} sizeOf(IMemory_k)/l + \sum_{k=1}^{n} sizeOf(IHarddisk_k)/m
\]
\[
+ \sum_{k=1}^{n} sizeOf(IDefaultWebClient_k)/n + \sum_{k=1}^{o} sizeOf(IConfiguration_k)/o
\]

*Fig. 10. Code size of dynamic adaptable agent*
tation classes. To measure this overhead the runtime of the different methods have to be compared. Tests have been done on IBM PowerPC running AIX 4.3.3 and on Intels running Windows NT or Linux.

In table 2 the average runtimes (arithmetic mean) are shown calculated from 100 measurements. These measurements have been done on top of a Intel Celeron with 366 MHz, 64 MB memory running SuSE Linux 6.3 with Kernel 2.2. The repository has been installed locally to disregard unsteady network delays. In the last column of the table the overhead ratio for dynamic adaptation

\[
\text{runtime of adaptable agent} - \text{runtime of monolithic agent} \over \text{runtime of adaptable agent}
\]

is given. The runtimes are measured by taking a time stamp before the method call and a time stamp after the method has returned. The difference between the two timestamp has been taken as method runtime. In case of the dynamic adaptable agent methods are called on adaptor instances, whereas in the monolithic agent the methods are called directly on instances of implementation classes. The methods are executed in the same order as listed in table 2 (the method order of the table corresponds to the computational flow). Partly the measured runtimes

<table>
<thead>
<tr>
<th>adaptor implementation group</th>
<th>method</th>
<th>average function runtime ms</th>
<th>runtime ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-memory</td>
<td>getPhysicalMemorySize()</td>
<td>946</td>
<td>9</td>
</tr>
<tr>
<td>m-harddisk</td>
<td>getFreeDiskSpace()</td>
<td>211</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>getTotalDiskSpace()</td>
<td>48</td>
<td>45</td>
</tr>
<tr>
<td>m-configuration</td>
<td>setDiskSpace()</td>
<td>407</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>setMemoryCache()</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

of the dynamic adaptable agent are almost equivalent to the runtimes of the monolithic agent (in the case of getTotalDiskSpace() and setMemoryCache()), partly the runtimes of the dynamic adaptable agent are higher (in the case of getPhysicalMemorySize(), getFreeDiskSpace() and setDiskSpace()). This pattern can be explained by the architecture of dynamic adaptation. For the first method out of each implementation group the dynamic adaptable agent has a runtime overhead. Because if an adaptor object (implementing the interfaces of an implementation group) is accessed for the first time, calling a certain method, context awareness is executed for the implementation group. Context awareness determines the suitable implementation classes, loads and instantiates them before the method can actually be executed. For all subsequent method calls in this implementation group the overhead is minimal or even equal zero. Because following calls are just propagated by the adaptor to the pre–loaded implementation classes without latency. Thus, the first method execution on an adaptor
within the dynamic adaptable agent has a higher runtime than the execution of the same method in the monolithic agent.

From these measurements following rules can be deducted to get a guideline when dynamic adaptation can improve the overall system in terms of efficient bandwidth usage: If a large number of different environments must be supported, which results in a large number of implementation classes and implementation classes are in general big sized (an implementation class should be bigger than the sum of all profiles of an implementation group), dynamic adaptation may be a better choice than the conventional customized version as a monolithic agent. Furthermore the runtime overhead for adaptation and loading implementation classes becomes negligible, if the environment dependent method has a long running time on a host or if the agent uses the dynamically loaded method more than once.

6 Conclusions

The motivation for dynamic adaptation is to improve mobile agents in terms of efficiency. The code which is moved over the network is limited to the parts that are environment independent and needed everywhere. Environment dependent parts are only transferred when needed. As a result of studying the state-of-the-art in adaptation, two fields of adaptation have been found: static adaptation and continuous adaptation. Both can not entirely fulfill the demands of dynamic adaptation. Thus, a new concept has been developed, which was influenced by methodologies from static adaptation and continuous adaptation, i.e. reconfiguration and context awareness.

6.1 Contribution

The concept of dynamic adaptation has been implemented as a framework using Java technologies. The framework includes the following parts:

- adaptor generator — context awareness
- loader — repository

The adaptor generator automates the creation of adaptors for the application programmer. The functionality interfaces are read by the adaptor generator and transformed into adaptor classes using Java reflection. The output of the adaptor generator is an adaptor class in Java source code.

The context awareness includes the frame for profiles, several basic profile values like operating system, CPU architecture and default web browser which are needed for the example application. Profile values for a future application must be created as needed. Further more, the context awareness includes an execution environment for the profiles embedded into the adaptors.

The loader extends the default Java class loader. It loads the appropriate implementation class as specified by the context awareness into the adaptor class. Both the context awareness and the loader rely on the service of a repository.
which serves the implementation profiles and the implementation classes. In order to minimize the impact on the autonomy of the mobile agent the concept of proxy repositories has been created. Proxy repositories reside on hosts closer to the mobile agent and reduce communication overhead when loading profiles or implementation classes for adaptation. Because of using reflection the concept relies on programming languages which support this technology and the modification of the framework is necessary if another programming language than Java is used.

6.2 Future Work

The current implementation of the repository holds the instances of all implementation classes in memory in order to get the according implementation profiles. This is sufficient if only a small number of implementation classes are needed as in the case of the sample application. Since a strength of dynamic adaptation is the gain of bandwidth in the case of a high number of implementation classes with a big size, the repository of the current implementation may become a bottleneck. The solution may be the loading and instantiating of each implementation class at startup time of the repository. After the startup the separated profiles are saved only.

Complete transparency to the application has not been achieved. Adaptors hide most of the adaptation mechanism, but are still visible to the core agent. A further improvement concerning transparency would be the implementation of an adaptor generator which generates Java byte code during runtime and not Java source code as in the prototype implementation.

The authors wish to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of the paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of Technology, and the Leibniz Supercomputing Center of the Bavarian Academy of Sciences. Its webserver is located at http://wwwmnmteam.informatik.uni-muenchen.de.

References


http://java.sun.com/j2me/.


Fast File Access for Fast Agents

Eugene Gendelman, Lubomir F. Bac, and Michael B. Dillencourt

Department of Information and Computer Science
University of California
Irvine, CA 92717-3425 USA
{egendelm, bc, dillenco}@ics.uci.edu

Abstract. Mobile agents are a powerful tool for coordinating general purpose
distributed computing, where the main goal is high performance. In this paper
we demonstrate how the inherent mobility of agents may be exploited to achieve
fast file access, which is necessary for most general-purpose applications. We
present a file system for mobile agents based exclusively on local disks of the
participating workstations. The mobility of agents allows us to make all file op-
erations local, which significantly reduces access time. We also demonstrate
how code files and special system files can be handled efficiently in a local-
disk-based environment.

1 Introduction

The most popular target of today's mobile agent systems is Internet computing. The
important characteristics of systems targeting Internet are security and heterogeneity.
To achieve these properties most mobile agent systems [1-5] are implemented in a
scripting language such as Java [14], Tcl/Tk [15], or Telescript[16].

In contrast, agent systems specializing on large general-purpose applications, which
are traditionally targeted by parallel and distributed systems, have different require-
ments. First of all, security [17] is not such an important issue here: the agents are
cooperative with each other, as they work together on solving one problem. The appli-
cation runs on a set of trusted nodes, which treat agents as user programs. On the other
hand, the computational speed of the system becomes critical.

There are many aspects defining the performance of the system. These include load
balancing, optimized agent code, and efficient and dynamic use of available comput-
ning resources. For many applications one of the main issues affecting the performance
of the system is efficient file access—the main focus of this paper.

With mobile agent systems we can identify four different types of files. These in-
clude data files, code files, profile/configuration files, and checkpoint files. Figure 1
illustrates the organization of the system and the connections between the different
types of files and system components. We assume a network of workstations or PCs.
Each machine runs a specialized server, called daemon (D), which manages all agents
on its machine. Each daemon supports one or more places for agent execution, called
logical nodes (N). Agents (A) run in the logical network, which can have an arbitrary, application-specific topology.

![Diagram](image-url)

**Fig. 1.** File types in MESSENGERS

There is no global shared file system. Instead, each machine has its own local disk and thus each daemon has its own file system (FS). Thus any file can only be accessed from the machine in whose file system it currently resides. Such a fully distributed file system has several benefits compared to traditional file systems: (1) It provides very fast, local access to application files; (2) It removes the load from the central file server; (3) It utilizes available resources, namely local disks; and (4) It decreases checkpointing overhead for fault tolerance.

We now describe the four types of files the system must be able to handle, and point out the specific problems associated with each file type that need to be solved in the absence of a shared file system.

**Data Files.** General-purpose computations require that agents be able to create, destroy, and use files to maintain the results or inputs of their computations. The mobility of agents offers a unique way to speed up application file access, which is not possible in message-passing systems. In a message passing system, if several processes use the same file, a shared file system must be used. If this is not available, the file must be moved between machines as needed. With mobile agents, we have the option of moving the code to the data, rather than moving the file. Thus a shared file system is unnecessary. In our approach each file is logically attached to the logical node where it was created. To access the file, an agent must be present at the corresponding logical node.

Figure 2(a) illustrates the situation where a stationary process (or thread) Pd running on a machine D3 needs to access a file f.dat that is stored on the disk of a different machine (D2). This requires a shared file system that makes the location of the file transparent to the process, or the file must explicitly be copied to D3. Figure 2(b) shows the same situation with mobile agents. In this case, the agent, currently executing in logical node d on machine D3 can hop to node b, to which the file is attached. As a result, all application file accesses are always local.
Fig. 2. File access scheme. a) Message passing system on the shared file system. b) Mobile agent system on non-shared file system: the file f.dat is logically attached to the logical node b. Agent can access the file by first hopping to b.

Problems: To support load balancing or crash-recovery, logical nodes may be remapped to different machines at runtime. Any file attached to a migrated node must be moved together with the logical node.

Code Files. An agent consists of a program, local data, and a current execution state. As the agent hops between machines, all three components must somehow be made available on each destination machine. Since the code is usually the largest portion of the agent, optimizing its transfer is important for performance.

There are several ways the agents’ code can be handled. In most systems, agents carry their entire code whenever they hop. This approach is necessary for agents working in wide area networks, for example, when searching for the information on the Internet, but is not optimal for performance-oriented systems. If a copy of the agent code can be made available at all participating daemons, then agents don’t need to carry their code at migration. Furthermore, the code can easily be compiled into the native code of each machine, saved as a library, and then dynamically linked when needed, thus avoiding the overhead of interpretation or on-the-fly compilation [6, 7].

Problems: Predistributing code minimizes the agent size at migration, but it would not be efficient to distribute all possible code libraries to all local file systems, since any
given agent may visit only a small subset of the participating machines. Furthermore, agents are spawned dynamically, and thus it is not known at initialization time which agent code will actually be needed for a given application. Section 3 will present the necessary mechanisms to solve this problem by distributing library code dynamically, on demand.

**Configuration File.** To run a mobile agent application, the user must specify the system configuration, in particular, the names and addresses of the initial set of daemons and their interconnections. This information is supplied in a configuration file.

*Problems:* Unlike data files, which are associated with logical nodes, and code files, which are associated with agents (Figure 1), only a single copy of the configuration is needed for the entire system. The problem, however, is that this file must be accessible not only by the already running daemons, but also by new daemons added to the system at runtime; each new daemon needs to know which other daemons are currently running in the system and how to contact them. Given that there is no shared file system, where should this file be kept? The solution, discussed in Section 4, is to provide central authority, called the MCommander (Figure 1).

**Checkpoint Files.** Checkpoint files are used for failure recovery. They contain a snapshot of the daemons at the time of a checkpoint. Thus a separate checkpointing file is associated with every daemon. The checkpoint information is used to recover the daemon after the failure [8].

*Problems:* Storing the checkpoint file on the local disk is problematic—if the entire node crashes, the file becomes unavailable, which defeats the purpose of maintaining a checkpoint. At the same time, the absence of a shared file system makes it impossible for the daemon to write the checkpoint file to another machine’s file system. This problem is addressed in section 5. It describes the principles of replication and regeneration mechanisms, and presents a comparison of checkpointing overhead using local disks and NFS. The detailed description of the solution to this problem is described in [9].

The local disk-based file system has been implemented as part of the MESSENGERS project [10]. Agents in MESSENGERS are called Messengers—a term we will use throughout the rest of the paper to refer to mobile agents.

## 2 Application Files

As was said in the introduction, the application files are associated with the logical nodes, instead of the physical machines.
As logical nodes migrate from one daemon to another due to load balancing [11, 12], or daemon failure [8], the connected node files also migrate. This is illustrated in Figures 3. The node files are sent to the new location using TCP/IP while packing node structures for migration, so that when the node migrates, its files are already available. The overhead of such migration is presented below.

Fig. 3. Redistribution of logical nodes. a) New daemon added. b) Failed daemon removed

Since Messengers always access all files locally, we measured the resulting performance gain. We have compared the time for the most common operations using the shared file system NFS supported on Unix, with the same operations performed using our local disk-based file system (LDFS).

Table 1 presents the results (given in microseconds). For most operations, LDFS is considerably faster than NFS. Specifically, opening and closing a file is 41.2 times faster, and writing to a file is 14 times faster on LDFS.

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Open/Close (msec)</th>
<th>Read (msec)</th>
<th>Write (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NFS</td>
<td>LDFS</td>
<td>NFS</td>
</tr>
<tr>
<td>0</td>
<td>3.71</td>
<td>0.09</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>0.00</td>
</tr>
<tr>
<td>1,000</td>
<td>-</td>
<td>-</td>
<td>0.04</td>
</tr>
<tr>
<td>10,000</td>
<td>-</td>
<td>-</td>
<td>0.42</td>
</tr>
<tr>
<td>100,000</td>
<td>-</td>
<td>-</td>
<td>4.80</td>
</tr>
</tbody>
</table>

These measurements show the advantage of using LDFS for the applications, where the agent needs to access only one file at the time. These applications include Monte-Carlo simulations and grid-based individual-based simulations. For example, if we want to simulate the ecosystem, the terrain is represented with logical nodes, and animals are programmed with agents. At each step the “evolution” information is recorded in files on every logical node.

In the applications, where agents need to access more than one file at the time, where files are on different nodes, it is the programmer’s responsibility to make file accesses efficient. The basic idea is to prefetch the data from the remote file. We implemented a simple application that merges two files into a third file. We compared the
sequential solution (sequential/NFS) with three agent-based solutions. Each file was accessed from a different machine. In the most straight-forward solution (LDFS/1 agent) a single agent was used to hop between machines and bring data blocks from the two input files, merge them, and write to the output file. In the more sophisticated solution, there were eight agents. Each of the two input machines had four agents hoping between it and the output machine with agents synchronizing and exchanging data on the output machine via a rendezvous. We ran the more sophisticated solution in two different configurations. In the first setup (LDFS/8 agents), each file was stored on the local of its respected machine, and LDFS was used. In the second setup (NFS/8 agents), all files were stored on a central server and were accessed using NFS.

We used the NFS provided by the department: NFS version 2, ran Solaris 7 on Sun Sparc Ultra-1. The caching was not set up. Experimental machines are connected to the server with 10 Mbps Ethernet.

Table 2 shows the results of this experiment. We merged two files of strings, 100 bytes each. The resulting file size was 100 Mb. The results are network-dependent. On the Ethernet, when a single agent is used, the merging time is 55% slower than the sequential merging, and the implementation with prefetching is 58% faster. On the collision-free switch the difference in results is much smaller, but still the program with a single agent is a little slower than sequential program, and LDFS implementation with multiple agents is faster.

<table>
<thead>
<tr>
<th></th>
<th>Ethernet</th>
<th>Collision-free switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential (NFS)</td>
<td>176</td>
<td>172</td>
</tr>
<tr>
<td>LDFS 1 agent</td>
<td>273</td>
<td>69</td>
</tr>
<tr>
<td>LDFS 8 agents</td>
<td>100</td>
<td>89</td>
</tr>
<tr>
<td>NFS 8 agents</td>
<td>172</td>
<td>58</td>
</tr>
</tbody>
</table>

The main drawback of keeping application files on local disks rather than a shared file system is increased overhead during file migration. We have investigated this problem quantitatively. For this experiment, a grid of 1000 logical nodes was created, so that each node had four connected links. Then the nodes were migrated one by one to another daemon. The migration of each consecutive node started only after the previous one had completed. The total migration time was recorded, and the migration time for one node was computed. In these experiments we wanted to measure the correlation between the size of the node file and the migration time. Table 3 presents the node migration time in NFS and LDFS with different sizes of the node files.

In the case of NFS the migration time is independent of the file size, since files are not migrated together with nodes. For LDFS, migration time grows proportionally to the size of the node file. Interestingly, for file sizes up to 30 Kbytes the migration times in NFS and LDFS are the same. This is because of the overlapping of file migration with packing of logical nodes for migration. By the time a node is ready to be sent, the node file is already sent, and it does not delay node migration.
3 Code File Distribution

For any given application, one of the local file systems is designated as the master file system (MasterFS). This initially contains all the agents’ code libraries. Whenever a daemon receives an agent whose code library it does not yet have, it requests it from the daemon on the MasterFS. A copy of the file is sent through the TCP/IP channel and stored on the local file system for future use.

To avoid excessive traffic on the network, we use a hierarchical approach to code file distribution. All daemons sharing a file system form a group and choose one file system to be the coordinator (FS coordinator) [13]. By convention, this is always the daemon with the lowest IP. Whenever a daemon is missing a library, it requests it from its FS coordinator. The FS coordinator talks to the MasterFS coordinator, which transfers the library to the local FS coordinator. Then the local FS coordinator broadcasts to its group that the library is available.

3.1 Maintaining Library Consistency

As libraries are being modified, we need a way to make sure that all the daemons use the same version of the library. In order to do this, whenever the library is copied from the MasterFS, another file is created, containing a time stamp of when the file was last modified on the MasterFS. Before using the library, the daemon can verify that the modification stamp, stored locally, is the same as the current modification stamp of the library on the MasterFS. If modification stamps differ, then a copy of the library has to be loaded from the MasterFS.

3.2 Loading Libraries to a New Daemon

Figure 4 shows how libraries are loaded when a new daemon is added to the system. There are two cases, depending on whether the new daemon is the first one created on

<table>
<thead>
<tr>
<th>File Size (Kbytes)</th>
<th>NFS Migration Time (sec)</th>
<th>LDFS Migration Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>10</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>30</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>50</td>
<td>0.05</td>
<td>0.11</td>
</tr>
<tr>
<td>100</td>
<td>0.05</td>
<td>0.18</td>
</tr>
<tr>
<td>150</td>
<td>0.05</td>
<td>0.25</td>
</tr>
<tr>
<td>200</td>
<td>0.05</td>
<td>0.31</td>
</tr>
<tr>
<td>1000</td>
<td>0.05</td>
<td>1.35</td>
</tr>
</tbody>
</table>
the current machine or whether there is already at least one daemon running. The second case, illustrated in Figure 4(a), is simpler so we consider it first.

![Diagram](image_url)

**Fig. 4.** Loading libraries to a new daemon

Assume that daemon $D_{\text{new}}$ is being added to the system. Since there are other daemons ($D_{1}$ through $D_{n}$) running on the current machine, the local file system already has all necessary code files. Furthermore, Daemon $D_1$ (the one with the lowest IP address, by convention), serves as is the local FS coordinator. The newly created daemon $D_{\text{new}}$ contacts this FS coordinator (step 1), which returns to it a list of currently active code libraries (step 2). The new daemon then loads these from the local file system (step 3).

Figure 4(b) illustrates the second case, where the newly created daemon, $D_{\text{new}}$, is the first one on the current machine. Consequently, the local file system may or may not contain all the necessary code files. To find out which files it needs, the daemon first contacts the FS coordinator, $D_n$, on the MasterFS (step 1). This coordinator replies by returning to the calling daemon the list of currently active code files (step 2). The daemon checks if these exist on its local file system (step 3). If so, the daemon simply loads them from the local file system; otherwise, it again contacts the master FS coordinator (step 4), which sends it the missing files as a message (step 5). The daemon loads these into memory, but also saves them on the local file system for future use (step 6).

### 3.3 Loading Messengers Libraries on Demand

Section 3.2 described the loading of code libraries to a newly created daemon. In this section we describe the distribution of code libraries required by Messengers migration. For any Messenger in the system, daemons that execute it need to have the Messengers code libraries. There are several ways a new Messenger can arrive at a daemon. A new Messenger can be injected by the user from the shell, a Messenger can be injected on the current node by another Messenger, or a Messenger can hop in from another node.

One approach to distributing a new library is loading it on every daemon in the system as soon as the Messenger is created. The advantage of this approach is that it minimizes the wait time for loading a new library, as waiting occurs only once, before
the library is loaded on every daemon. On the other hand, some libraries might be loaded on nodes that never use them.

Alternately, the library can be loaded on demand. In this case libraries are loaded only on the machines that actually need them. However, the time Messengers are waiting for the library to load will increase.

![Fig. 5. Loading MESSENGERS libraries on demand](image)

Both approaches seem to be acceptable, but we have chosen the first one, because it allows a much cleaner design of the protocol and its integration into the system. Figure 5 illustrates the protocol to add a new Messenger library under this approach. When a new Messenger arrives at a daemon (D6), the daemon checks if the corresponding library is already loaded. If not, the Messenger is suspended, and the daemon sends a request for the library to the FS coordinator on the MasterFS (step 1). When MasterFS coordinator receives the request, it broadcasts a `loadLib` message to all daemons on its own machine; in our example, this includes the daemons D2 and D3 (step 2). It also broadcasts the same message to all FS coordinators on other machines; this includes the daemon D4 (step 3). Each FS coordinator checks whether the latest version of the necessary library exists on its local file system. If not, the coordinator requests a new copy from the MasterFS coordinator (step 4). When the library arrives (step 5), the FS coordinator informs all local daemons; in our example, D4 broadcasts the information to D5 and D6 (step 6). Finally, all daemons load the library from their local file systems (step 7). Daemon D6 then restarts the suspended Messengers, whose arrival triggered the library loading protocol.

4 Dynamic System Configuration

4.1 Configuration File and MCommander

Before starting the system, the user creates a system configuration profile (SCP). In the SCP the user lists all the physical nodes that might participate in the computation,
their file systems, and all the system and user-created libraries that will be used by the application.

![Diagram](image)

**Fig. 6.** System architecture

Since the daemons cannot share a common file to exchange configuration information at startup, we use a mediator process for this purpose, called MCommander. This also serves as an interface between the user and the MESSENGERS system. The system structure is presented in figure 6, where D1 through D4 are daemons, and fs1 through fs3 are file systems.

The MCommander is the first process started in the system. In its initialization phase it loads the topology of the system from the SCP. The user then gives commands to the MCommander, such as add/remove daemon, inject new agents, or restart the system.

### 4.2 AddDaemon Protocol

The use of the MCommander solves the problem of exchanging the communication port numbers among active daemons. When the MCommander receives the command from the user to add a new daemon, it starts a daemon on a remote machine through the `xsh` command and passes its own port number as an argument to the daemon. If `xsh` is not available, the user can manually login on the remote machine and run the command provided by the MCommander. When the new daemon is started, it sends a message to the MCommander, containing its port number. Then the MCommander sends to the new daemon a list of port numbers of already active daemons. In the same message, the MCommander sends information about the current system topology: which system libraries have to be loaded, and what daemons are on what file system.

Figure 7 summarizes the AddDaemon algorithm. The MCommander first starts a Messengers daemon on a remote machine, passing it as an argument the MCommander’s port number (step 1). The new daemon establishes a receiving socket, and sends its port number to the MCommander (step 2). The MCommander sends to the new daemon the list of all active daemons with their port numbers, and corresponding file systems. It also sends the list of all system and user libraries that are currently being
used in the system (step 3). The new daemon loads the system, user, and Messengers libraries, and announces itself as an active daemon (step 4). All active daemons add the new daemon to their active daemon’s lists, along with its port number and file system. If the xrsh command is not available, step 1 is skipped. The protocol must then be started manually from step 2, with the MCommander’s port number passed as a command line argument.

<table>
<thead>
<tr>
<th>MCommander</th>
<th>New Daemon</th>
</tr>
</thead>
</table>
| **Step 1**
start messengers on new daemon | **Step 2**
contact MCommander                              |
| **Step 3**
add new daemon to the list of active daemons
send list of active daemons, their FS and list of needed libs | **Step 4**
Load libs, announce to everyone as added daemon |

Fig. 7. AddDaemon algorithm

5 Checkpoint Files

The fault tolerance of the system is provided with a failure-recovery protocol [8]. During the system execution, all participating daemons periodically save their state to the stable storage. The collection of the checkpoint files of all the daemons constitutes a consistent system state, which is used to restart the system in case of node failure. If a shared file system is used, then every checkpoint file is accessible by every daemon. When a checkpoint is stored on the local disk, the data stored on the disk becomes inaccessible when the node fails. The solution is to store the checkpoint locally, and also to send a copy to a neighboring machine. Then, in case of a failure, the second copy can be used to restart the failed node.

5.1 Replication and Regeneration

The file systems that could potentially support system nodes are given unique file system ids. These are determined by the locations of the file systems in the configuration file. File systems hosting active processes are arranged in a logical ring according
to their file system ids; i.e., file system i is connected to its successor, i+1, and its predecessor, i−1, modulo the total number of file systems.

At each checkpoint, the daemon saves its state to its local file system, and sends a copy of its checkpoint file to one of the daemons residing on the file system next in the ring. As a result, each checkpoint file has two copies. A ring with four file systems is illustrated in Figure 8(a). The lower rectangles represent file systems, the circles above represent daemons running on machines on these file systems. The letters in the top rectangles represent daemon checkpoints logged on the current file system.

![Fig. 8. Replication on LDFS: a) System before the failure. b) Recovery after the failure of node c. c) System after the checkpoint](image)

In case of node failure, the failed daemon is restarted on the next file system, and the lost copies of checkpoints are regenerated. For example, if node c fails (Figure 8(b)), its checkpoint is loaded by node d. Also, a copy of checkpoint file of c is copied to fs8, and copies of checkpoint files of daemons a and b are copied to fs6. The system configuration after the next checkpoint is shown in Figure 8(c). The details of the replication and regeneration mechanisms are presented in [9].

The other file types discussed earlier also have to be replicated. The application files are replicated together with checkpoint files, as they are a part of the checkpoint. The replication depends on the file operation mode. Files open only for reading have to be replicated only during the first checkpoint, or after the failure of one of the supporting file systems. In the case of write-only files, in which all writes append data to the end of the file, the increment written since the previous checkpoint is appended to the end of the replica file. In the case of read/write files, in which writes are allowed to modify the file, the entire file is replicated at every checkpoint.

The profile and Messenger code libraries are replicated only to the first two active file systems. The replication happens only during the first checkpoint, or if one of these file systems fails. In the latter case these files are copied to the new second active file system.

### 5.2 Performance

We measured the checkpoint overhead induced by both saving the checkpoint to the NFS and to the local disk. In our experiments the logical nodes are connected in a single logical ring, as shown in the Figure 9. There is one Messenger that continuously travels around the ring. When it completes 1000 rounds, the application terminates. There are no other Messengers in the system. (Otherwise the size of the checkpoint would vary.)
The application was run without checkpointing, with checkpointing on NFS, and with checkpointing on LDFS. The checkpoints were taken every 30 seconds. The total execution times and the numbers of checkpoints taken were recorded. By subtracting the time the application took to complete without checkpoints from the execution time of the application with checkpoints, and dividing this by the number of checkpoints, we derived the time overhead of a single checkpoint:

\[ T_{\text{checkpoint}} = \left( T_{\text{total with checkpoints}} - T_{\text{total without checkpoints}} \right) / N_{\text{checkpoints}} \]

We measured the results for systems consisting of different numbers of daemons. The number of logical nodes per daemon was held constant, and the checkpoint size was always 1Mb. The results of the experiments are shown in table 4.

We repeated this experiment on a set of machines connected through the Ethernet and a set of machines connected through a collision-free switch. The difference in checkpoint overhead was negligible. From this table we can conclude that NFS and LDFS show comparable performance when checkpoint size is around 1Mb. On the 8 daemon system LDFS was faster than NFS.

**Table 4. Checkpointing overhead for the checkpoint size 1Mb.**

<table>
<thead>
<tr>
<th># of daemons</th>
<th>Total checkpoint overhead (%)</th>
<th>Time of the single checkpoint (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NFS</td>
<td>LDFS</td>
</tr>
<tr>
<td>2</td>
<td>2.1%</td>
<td>3.3%</td>
</tr>
<tr>
<td>4</td>
<td>3.3%</td>
<td>4.0%</td>
</tr>
<tr>
<td>6</td>
<td>5.0%</td>
<td>4.3%</td>
</tr>
<tr>
<td>8</td>
<td>7.5%</td>
<td>4.6%</td>
</tr>
</tbody>
</table>

Then we repeated our experiment with checkpointing size of 10Mb. The results are presented in table 5. This time we took checkpoints every 3 minutes. Checkpointing on NFS is faster than on the LDFS on the Ethernet, but slower than the LDFS where workstations are connected by a collision-free switch.

**Table 5. Checkpointing overhead for the checkpoint size 10Mb.**

<table>
<thead>
<tr>
<th># of daemons</th>
<th>Total checkpoint overhead (%)</th>
<th>Time of the single checkpoint (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NFS</td>
<td>LDFS</td>
</tr>
<tr>
<td>2</td>
<td>4.7%</td>
<td>8.3%</td>
</tr>
<tr>
<td>4</td>
<td>5.8%</td>
<td>23.1%</td>
</tr>
<tr>
<td>6</td>
<td>8.9%</td>
<td>21.3%</td>
</tr>
<tr>
<td>8</td>
<td>11.4%</td>
<td>21.9%</td>
</tr>
</tbody>
</table>
6 Conclusion

The mobile agent paradigm is suitable for coordinating performance-oriented general-purpose computations, but it must be extended to handle file accesses efficiently. In this paper we described the approach implemented as part of the MESSENGERS system, which implements all files using only the local disks of the participating machines, instead of a shared file system, such as NFS. The local disk-based file system provides support to efficiently deal with four types of files involved in the operation of a mobile agent system:

First, data files accessible explicitly by agents are associated with logical node. By utilizing the inherent mobility of agents, all operations on these files are local, and hence very fast. Code files are distributed on demand under two different scenarios: (1) when an agent hops to a node that does not have a copy of its code, the daemon requests it from a master daemon; (2) when a new daemon is added to the system at runtime, it brings itself up to date in a similar manner, i.e., by requesting copies of all code files distributed so far from the master daemon. In order to join the system dynamically, a new daemon must have a way to determine the addresses of all currently executing daemons. Since no shared file system exists, a special process, called the MCommander, is implemented, who is in charge of maintaining the configuration file necessary to obtain the relevant system-wide information. Finally, each daemon maintains a checkpoint file to permit uninterrupted operation in case of a failure. Each checkpoint file is maintained in duplicate, once on the current node and a second copy on a neighboring node. The application files are duplicated in the same way as checkpoint files. The complete set of all Messenger code libraries and the profile file are replicated only to the first two active file systems. Figure 9 summarizes the distribution and replication of the different file types.

Currently, the MESSENGERS system using the local-disk based file system as described in this paper runs on Solaris and Linux platforms.

![Fig. 9. File distribution in LDFS](image)
References

1. Introduction to the Odyssey API. http://www.genmagic.com/technology/odyssey.html
   Kurt Rothermel, Radu Popescu-Zeletin, editors, Proc. of the First International Workshop
   on Mobile Agents MA’97. (Berlin, Germany), April 1997.
   http://www.uni-kl.de/~AGNehmer/Projekte/Ara/index_e.html
4. Robert Gray, George Cybenko, David Kotz, and Daniela Rus. Agent Tcl. In William
   Cockayne and Michael Zypa, editors, Itinerant Agents: Explanations and Examples with
5. Johansen, D., van Renesse, R. and Schneider, F. B. An Introduction to the TACOMA
   Distributed System, Technical Report 95-23, Dept. of Computer Science, University of
   State Capture of Self-Migrating Computations in Messengers”. ICSE’98 Intl Workshop on
   Computing and Communication in the Presence of Mobility, Kyoto, Japan, April 1998.
   www.ics.uiuc.edu/~bic/messengers/messengers.html
   August 1999
8. Eugene Gendelman, Lubomir F. Bic, Michael B. Dillencourt. An Application-Transparent,
   of 20th International Conference on Distributed Computing Systems (ICDCS 2000). Taipei,
   Taiwan, April 2000
   Local Disk-Based File System for Mobile Agents. Department of Information and Com-
   puter Science, University of California, Irvine, CA. TR 01-21.
10. MESSENGERS. www.ics.uiuc.edu/~bic/messengers/messengers.html
    The Seventh Symposium on the Frontiers of Massively Parallel Computation
12. Robert K. Brunner and Laxmikant V. Kale. Handling Application-Induced Load Imbalance
    http://www.ics.uiuc.edu/~egendelm/prof/publications.html
    Wesley
    General Magic, Inc., Sunnyvale, CA. Appeared in Bradshaw, J., Software Agents,
    AAAI/MIT Press.
    and Requirements. Proceedings of the National Information Systems Security Conference,
    591-597.
Flying Emulator: Rapid Building and Testing of Networked Applications for Mobile Computers

Ichiro Satoh
National Institute of Informatics / Japan Science and Technology Corporation
2-1-2 Hidotsukushi, Chiyoda-ku, Tokyo 101-8430, Japan
Tel: +81-3-4212-2546 Fax: +81-3-3556-1916 ichiro@nii.ac.jp

Abstract. This paper presents a mobile-agent framework for building and testing mobile computing applications. When a portable computing device is moved into and attached to a new network, the proper functioning of an application running on the device often depends on the resources and services provided locally in the current network. To solve this problem, this framework provides an application-level emulator of portable computing devices. Since the emulator is constructed as a mobile agent, it can carry target applications across networks on behalf of a device, and it allows the applications to connect to local servers in its current network in the same way as if they were moved with and executed on the device itself. This paper also demonstrates the utility of this framework by describing the development of typical location-dependent applications in mobile computing settings.

1 Introduction

Advances in networking technology have produced a shift in the nature of mobile computing systems and applications. A new class of mobile computing has enabled portable devices to link up with servers in networks to access information from them and delegate heavy tasks to them. Another new class is context-sensitive devices that have a distinct awareness of their locations and current network environments.

The focus of current research, however, is on the design of network and system infrastructure and context-aware applications for mobile computing. As a result, the tasks of building and testing applications have received little attention so far. This is a serious impediment to the growth of mobile computing, because the development of software for mobile computing devices is very difficult due to the limited computational resources of these devices. Furthermore, the tasks are often tedious and susceptible to errors, because changes in network connectivity and location may lead to sudden and unpredictable changes in contextual information. That is, a change in network and location may imply movement away from the servers currently in use, toward new ones. For example, a handheld device with a short-range radio link, such as Bluetooth, carried across the floors of an office building may have access to different resources, such as printers and directory information for visitors, on each floor. Therefore, to construct a correct application, the developer must test it in the environments of all the networks that
the device might be connected to. However, it is almost impossible for the developer to actually carry a portable computing device to another location and connect it to networks in that location. In fact, nobody wants to go up and down stairs carrying a portable device simply to check whether or not it can successfully print out data at networked printers on its current floor.

This paper presents a new framework for building and testing networked applications for mobile computing. This framework, called *Flying Emulator*, addresses the development of networked applications running on mobile computing devices that can be connected to servers through wired or short-range wireless networks. The key idea of the framework is to introduce a mobile agent-based emulator of a mobile device. The emulator performs application-transparent emulation of its target device for applications written in the Java language. Furthermore, since the emulator is implemented as a mobile agent, it can carry its applications to remote networks according to patterns of physical mobility and test the applications in the environments of those networks. Also, the framework provides a platform for building mobile computing applications from a collection of Java-based software components and allowing such an application to be executed on its target portable device without being modified or recompiled.

The remainder of this paper is organized as follows. Section 2 surveys related work and Section 3 explains a framework for building and testing mobile computing applications. Section 4 briefly reviews my mobile agent system and then presents the design and implementation of the framework. Section 5 demonstrates the usability of the framework through two real-world examples. Section 6 offers conclusions and suggestions for further work.

## 2 Background

There are two different notions of mobility: logical and physical. Physical mobility entails the movement and reconnection of mobile computing devices among networks, while logical mobility involves software, such as mobile code and mobile agents, that migrates among different servers and may use different sets of services on each of them (for example see [5,9,14]).

One of the most typical problems in physical mobility is that the environment of a mobile entity can vary dynamically as the entity moves from one network to another. A lot of research has been proposed to either transparently mask variations in mobility at the network or system level or adapt to the current environment at the application level [1, 12,13]. Nevertheless, current work on these approaches focuses on a location-transparent infrastructure for applications and location-aware applications themselves. Accordingly, the task of building and testing applications has received only limited attention.

Among the recent works, a few researchers have explored emulators of portable computing devices [4,10]. However, those approaches were designed to emulate some limited resources of target devices on standard workstations and networks, whereas the approach presented in this paper is designed to emulate the mobility itself of portable devices. In fact, it is very difficult for an emulator running on a standalone computer to simulate the whole context that its target device can interact with through networks. An extreme approach is to actually carry portable devices and attach them to local networks in the current location, but this is extremely cumbersome troublesome for the developer. Another extreme approach enables an application to run on a local workstation and link
up with remote servers through networks to access particular resources and services provided by remote networks; for example, the InfoPad project at Berkeley [10] and the network emulator of Lancaster University [4]. However, accomplishing this in a responsive and reliable manner is difficult, and the emulators cannot remotely access all the services and resources that are available only within the local networks because of security protection. Moreover, the approach is inappropriate since the network traffic increases when the amount of the data exchanged between the emulator and the remote servers is large. This is a serious problem in testing monitoring applications for gathering a large quantity data from network nodes or sensors.

Logical mobility is just a new design tool for the developers of distributed applications, while physical mobility results from new requirements for distributed applications. As discussed in [14], these two mobilities have been almost unrelated so far, despite their similarities. Although many mobile agent systems have been released over the last few years, few researchers have introduced the logical mobility approach, including mobile agent approach, as a mechanism for extending and adapting context-sensitive applications to changes in their environments by migrating agents or codes to the applications and servers, for example [8,11]. These approaches were designed as infrastructures for executing context-aware applications, so the researchers did not intend to test such applications. On the other hand, the framework presented in this paper is unique among existing research on both physical and logical mobility, because it introduces logical mobility as a methodology for building and testing applications in physical mobility.

3 Approach

The goal of this paper is to present a framework for building and testing mobile computing applications. An important target application of this framework is a network-dependent application, in the sense that it is designed to run on a portable computing device and may often access servers on local networks in the device’s current location either through wired networks such as Ethernet or short-range wireless networks, such as IEEE802.11b or Bluetooth.\(^1\) As the device moves across networks, the environment may change. That is, some new servers become available, whereas others may no longer be relevant. Such an application must be tested in all the network environments that the device could be moved into and attached to. Furthermore, most portable computing devices, including personal digital assistants and mobile phones, support few debugging and profiling aids since they are kept simple to reduce power consumption and weight.

To solve these problems, this framework introduces a mobile agent-based emulator of a portable computing device for use with applications. The key idea is to emulate the physical mobility of a portable computing device by using the logical mobility of the targeted applications (see Figs. 1 and 2). The emulator supports applications with not only the internal environment of its own target portable device, but also the external environment, such as resources and servers provided in the current network. To accomplish this, the framework satisfies the following requirements:

\(^1\) Wireless networking technology permits continuous access to services and resources through the land-based network, even when device’s locations change. On the other hand, my goal is to build and test applications running on a mobile computing device which may be connected to local networks in the current location.
Like other computer emulators, this framework performs application-level emulation of its target portable device to support applications by incorporating a Java virtual machine.

Depending on the movement patterns of its target portable device, the mobile agent-based emulator can carry applications on behalf of the device to networks that the device may be moved into and connected to.

The emulator allows us to test and debug applications with services and resources provided through its current network as if the applications were being executed on the target device when attached to the network.

All applications tested successfully in the emulator can still be performed in the same way without being modified or recompiled them.

Each mobile agent is just a logical entity and thus must be executed on a computer. Therefore, this framework assumes that each of the networks into which the device may
be moved and attached to has more than one special stationary host, called an access
go point host, which offers a runtime system for mobile agents. Each access point host is a
runtime environment for allowing applications running in a visiting emulator to connect
to local servers in its network. That is, the physical movement of a portable computing
device from one network and attachment to another network is simulated by the logical
mobility of a mobile agent-based emulator with the target applications from an access
point computer in the source network to another access point computer in the destination
network. As a result, each emulator is a mobile agent, and thus it can basically carry
not only the code but also the states of its applications to the destination, so the carried
applications can basically continue their processes after arriving at another host as if
they were moved with its targeted device.

In this framework applications are written in JDK 1.1 or 1.2-compatible Java lan-
guage, including Personal Java. However, some typical units of the Java language, such
as Java applications and Java applets, are not always appropriate in developing software
in mobile computing settings. This is because these units are essentially designed to run
in a static context and lack any unified mechanism for reporting contextual changes.
Instead, this framework introduces mobile agent-based software components for build-
ing context-sensitive applications. Another key idea of this framework is to introduce
a hierarchical mobile agent system, called MobileSpaces [15], as infrastructure for the
framework. This system is characterized by allowing multiple mobile agents to be dy-
namically assembled into a single mobile agent. Therefore, it enables us to construct an
application as a collection of mobile agents, like software component technology [18].
Furthermore, such an application can be extensible and adaptable in the sense that it
can dynamically customize its structure and functions to environmental changes in its
context by having mobile agent-based components migrated to it.

Since the framework itself and applications are written in the Java language, the
target portable devices must support this language. Moreover, while the framework does
not require any custom hardware, its current implementation requires its target devices
to offer TCP/IP communication over wired or wireless networks.

4 The Flying Emulator Framework

This section presents the prototype implementation of this mobile agent-based frame-
work, called Flying Emulator, which consists of the following four parts:

Mobile Agent-based Emulator: A mobile agent capable of carrying the target appli-
cations to specified access point hosts on remote networks on behalf of a target
portable device.

Application Runtime System: Middleware, which runs on a portable device, to support
the execution of mobile agent-based applications.

Access Point Runtime System: A runtime system, which is allocated in each network,
to allow the applications carried by an emulator to connect with various servers
running on the network.

2 In fact, most Java Applets and Java Beans can be easily translated into mobile agents in the
MobileSpaces. Moreover, I implemented an adapter for executing Java Applets and Java Beans
within this mobile agent-based components, but it is not compatible with all kinds of Applets
and Java Beans.
**Remote Control Server:** A graphical front-end to the whole system, which allows us to monitor and operate the moving emulator and its applications by remotely displaying their user interfaces on its screen.

The above parts are constructed independently of the underlying system and thus can run on any computer with a JDK 1.1 or 1.2-compatible Java virtual machine, including Personal Java.

![Migration of hierarchical mobile agents](image)

**Fig. 3.** Migration of hierarchical mobile agents.

### 4.1 MobileSpaces: A Mobile Agent System

Like other existing mobile agent systems, MobileSpaces can offer mobile agents as computational entities that can travel over networks under their own control. Furthermore, the system is characterized by the notion of hierarchical mobile agents. That is, the system allows multiple mobile agents to be dynamically combined into a single mobile agent. Fig. 3 shows hierarchical mobile agents and their migration. Each agent can directly access the services and resources offered by its inner agents and it is responsible for providing its own services and resources to the inner agents. This concept is applicable in constructing the mobile agent-based emulators presented in this paper, although it was initially introduced for constructing large and complex applications by assembling multiple mobile agents in distributed computing settings.

The MobileSpaces system is built on a Java virtual machine, and mobile agents are provided as Java objects. When an agent is transferred over a network, the runtime system stores the state and code of the agent, including mobile agent-based applications, in a bitstream defined by Java’s JAR file format that can support digital signatures for authentication. The MobileSpaces runtime system supports a built-in mechanism for transmitting the bitstream over networks by using an extension of the HTTP protocol. Almost all intranets have firewalls that prevent users from opening a direct socket connection to a node across administrative boundaries. Since the mechanism is based on a technique called HTTP tunneling, it enables agents to be sent outside a firewall as HTTP POST requests and responses to be retrieved as HTTP responses.
4.2 Mobile Agent-Based Emulator

The mission of the mobile agent-based emulator is to carry and test applications designed to run on its target portable device. Each mobile agent-based emulator is just a hierarchical mobile agent of the MobileSpaces system. Since every application is provided as a collection of mobile agent-based components, the emulator can naturally contain more than one mobile agent-based application inside itself and can migrate itself and its inner applications to another place. Since such contained applications are still mobile agents, both the applications running on an emulator and the applications running on the portable device are mobile agents of the MobileSpaces system and can thus be executed in the same runtime environment. Actually, this framework basically offers a common runtime system to both its target devices and access point hosts, in order to minimize differences between them as much as possible. In addition, the Java virtual machine can actually shield applications from most features of the hardware and operating system of target portable devices. Fig. 4 illustrates the correlation between the physical mobility of a running device and the logical mobility of an emulator of the device. As a result, the emulator is dedicated to emulating the movement of its target device.

![Diagram](image)

Fig. 4. Emulation of the movement of a mobile computer by migrating a mobile agent-based emulator.

**Emulation of Physical Mobility.** Each emulator can have its own itinerary as a list of hosts corresponding to the physical movement pattern of its target mobile device. The list is a sequence of the tuples of the network address of the destination, the length of
stay, and the name of the method to be invoked upon arrival. An emulator can interpret
its own itinerary and then migrate itself to the next destination. Such an itinerary can be
dynamically changed by the emulator itself or statically defined by the user through the
graphical user interface as shown in Fig. 5. Moreover, the developer can interactively
calculate the movement of the emulator through the remote control server.

When a portable computing device moves in physical space, it may be still running.
On the other hand, the emulator cannot be migrated over networks as long as its inner
applications are running, because they must be suspended and marshaled into a bitstream
before being transferred to the destination. To solve this problem, the framework divides
the life-cycle state of each application into three phases: networked running, isolated
running, and suspended. In the networked running state, the application is running in its
emulator on an access point host and is allowed to link up with servers on the network.
Upon disconnection, the application enters the isolated running state. In this state, it is
still running in its emulator on an access point host but is prohibited from communicating
with any servers on the network. The suspended state means that the emulator stops its
inner applications while keeping their execution states. This state corresponds to that
of a device that is sleeping to save battery life and avoid the risk of accidental damage
while moving.

For example, the movement of a suspended and disconnected device corresponds
to the suspended state. The movement of a running and then disconnected device is
simulated by the combination of the isolated running state on the source or destination
host for a specified duration and the suspended state only while migrating. Each emulator
maintains the life-cycle states of its inner applications. When the life-cycle state of
an application is changed, the emulator dispatches certain events to the application as
mentioned in the Appendix.

The Java virtual machine can marshal the heap blocks of a program into a bitstream,
but not its stack frames when migrating them, so it is impossible for a thread object
to migrate from one virtual machine to another while preserving its execution state. Instead, these events enable an application that has one or more activities using the Java
thread library to explicitly stop and store them before migrating over networks.

**Emulation of Portable Computing Devices.** The Java VM supports instruction-level
emulation of target portable devices and each emulator permits its inner applications to
have access to the standard classes commonly supported by the Java virtual machine
as long as the target device offers them. In addition, each emulator offers its inner
applications the particular resources of the target devices. The current implementation
of this framework supports emulators for two kinds of portable computing devices:
standard notebook PCs and pen-based tablet PCs running Windows or Linux. Also, the
emulators support several typical resources of portable computing devices; for example,
file storage and user interfaces such as displays, keyboards, and mouse-based pointing
devices.

---

1 Several researchers have explored mechanisms for migrating all the execution states of Java
objects, including threads and stack frames. However, these existing mechanisms are still
pre-mature for my goal, because they cannot transfer most computational resources and do not
often coexist with essential optimization techniques for the Java language.
File Storage: Each emulator can maintain a database to store files. Each file can be stored in the database as a pair consisting of its file/directory path name pattern and its content. Each emulator provides basic primitives for file operation, such as creation, reading, writing, and deletion and also allows a user to insert files into it through its graphical user interface.

Network: When anchored at an access point host, each emulator can directly inherit most network resources from the host, such as java.net and java.rmi packages. In the current implementation, a moving emulator cannot have its own network identifier, such as IP address and port number. However, this is not a serious problem because most applications on a portable device are provided as client-side programs, rather than server-side ones, as discussed in [7].

User Interface: The user interfaces of most handheld computers are limited by their screen size, color, and resolution, and they may be not equipped with traditional input devices such as a keyboard and mouse. Each emulator can explicitly constrain only the size and color of the user interface available from its inner applications by using a set of classes for visible content for the MobileSpaces system, called MobiDoc [16]. As mentioned later, our framework furthermore enables the developer to view and operate the user interfaces of applications in an emulator on the screen of its local computer, even when the emulator is deployed at remote hosts.

4.3 Application Runtime System

Like other mobile agents, each mobile agent in the MobileSpaces system must be executed on runtime systems, i.e., an agent platform, that can create, execute, transfer,
and terminate agents. Since applications designed for running on the device are implemented as mobile agents, this framework needs to offer a runtime system to each target portable device. Each runtime system maintains the execution of applications. Moreover, to make applications aware of environmental changes, each runtime system monitors the environment of the device, including characteristics such as network connectivity and location. Since this framework introduces the publish-subscribe event model used in the Abstract Window Toolkit of JDK 1.1 or later, the system notifies interested applications by invoking certain of their methods when detecting changes. Furthermore, it provides a collection of service methods to allow applications to have access to the device and its external environment, without any particular knowledge of the operating system and hardware of its target device.

You might wonder whether a mobile agent system is too large to run on portable devices. However, the MobileSpaces runtime system is characterized by its adaptability and its structure can thus easily be customized to be as small as possible by removing additional functions, including agent migration over networks. Also, the performance of applications running on the minimal runtime system is almost equal to that of the corresponding applications executed directly on the Java virtual machine.

4.4 Access Point Runtime System

Each access point host is treated as a peep-hole of the resources and services provided in its network from the applications in a visiting emulator. This framework assumes that more than one access point host is allocated in each network to which the target portable device may be attached. Each access point is constructed based on the common runtime system, which can be used for the target portable devices, and runs on a standard workstation without any custom hardware.

Many applications have their own graphical user interfaces (GUIs). To test such applications, the framework should offer a mechanism for letting these GUIs be remotely viewed and operated on the screen of the remote control server, instead of on the screen of their current hosts. The mechanism is built on the Remote Abstract Window Toolkit (RAWT) developed by IBM [6]. This toolkit allows Java programs that run on a remote host to display GUI data on a local host and receive GUI data from it. Each access point host can perform the toolkit, thus allowing all the windows of applications in a visiting emulator to be displayed on the screen of the control server and be operated using the keyboard and mouse of the server. Therefore, none of the access point hosts has to offer any graphics services.

4.5 Remote Control Server

This server is a control entity responsible for managing the whole system. It can run on a standard workstation that supports the Java language. It can always track the locations of all the emulators, because each access point host sends certain messages to the control server whenever a moving emulator arrives or leaves. Moreover, the server acts as a graphical front end for the system and thus allows the developer to freely instruct moving emulators to migrate to another locations or terminate, through its own GUIs. Moreover, by incorporating with a server of the RAWT toolkit, it enables us to view and operate the
GUIs of target applications on the behalf of their moving emulators. Also, it can monitor the status of all the access point hosts by periodically multicasting query messages to them.

5 Experience

To illustrate the utility of the framework, this section briefly describes my experience in building and testing two typical networked applications designed to run on portable computing devices.

Development of a Location-aware Printing Service System. By using the framework presented in this paper, I developed a directory service system for printers in an office building at Ochanomizu University. Each floor of the building is equipped with its own Ethernet-based sub-network and one or more printers, which are of various types and managed by different operating systems. Suppose that a user wants to print out data stored in a portable computer by using a printer on the current floor. This system offers a directory server to the sub-network of each floor. In the current implementation, each server is implemented based on the Jini system [2] and is responsible for discovering printers in only its sub-network. After a portable device attaches to the sub-network of the current floor, the server allocated in the sub-network can automatically advertise its printers to the visiting device. To construct a client-side application for the system, the developer needs to carry the device, attach it to the sub-network of each floor, and then check whether or not it can successfully access every printer on the current floor. We have developed such an application by using the framework. While it was impossible to measure the framework’s benefit in a quantitative manner, it did enable a developer to test such applications in the environments of all the floors, without going to each floor. I experimentally compared my system with one of the most conventional testing approach that runs the applications locally and allows them to access remote printing services through proxies for the services. Unlike my approach, the conventional approach can offer only particular services, but not all the services that are locally provided within the sub-network.

Development of a User Navigation System. This example illustrates the development of a location-dependent information system for assisting visitors to Ochanomizu University, like [1,3]. The current implementation of the system provides each visitor with a pen-based tablet PC, which can obtain various information from servers allocated on the sub-network of the current location through an HTTP-based protocol via IEEE802.11b wireless networks. Each building has one or more ranges of wireless networks. When moving from building to building, the tablet PC changes the displayed directory and map to match the user’s location. This framework could successfully test the system. That is, I constructed a mobile agent-based emulator for the tablet PC. The emulator can migrate a viewer application designed to run on the tablet PC to the sub-network of another building and enable the application to access the local database of the building and display suitable contents. Since the emulator can travel under its own control, it can exactly simulate the mobility of each visitor in testing such a user navigation system.
Fig. 6. Screenshot of the remote control server when the user navigation system runs on the mobile agent-based emulator.

By using the RAWT toolkit, this framework allows a content creator to view location-dependent information, which should be displayed on the tablet PC on the screen of his/her stationary computer as shown in Fig. 6. Fig. 6 (A) shows a window of the viewer application tested in the emulator. Fig. 6 (B) shows a user interface of the control server for monitoring several emulators and Fig. 6 (C) shows a window of an emulator for controlling itself and its applications. Fig. 7 shows the target tablet PC (Fujitsu PenNote Model T3 with Windows98) running the viewer application. As illustrated in Fig. 6 (A) and Fig. 7, both the application running on the emulator and the application running on the target device can have the same presentation of navigation information in the same location. That is, the tested application can be performed in the target device in the same way as if it were executed in the emulator. Furthermore, this example shows that the framework can provide a powerful methodology not only for testing applications for portable computers but also for creating location-dependent contents.

6 Conclusion

I have presented a framework for building and testing networked applications for mobile computing. It was inspired by the lack of methodologies for developing context-aware applications in mobile computing settings. IT aims to emulate the physical mobility of portable computing devices by the logical mobility of applications designed to run on the devices. I designed and implemented a mobile agent-based emulator of portable computing devices. Each emulator can perform an application-level emulation of its target device. Since it is provided as a mobile agent in the MobileSpaces system, it can carry and test applications designed to run on its target portable device in the same way as if they were moved with and executed on the device. My early experience with the prototype implementation of this framework strongly suggested that the framework can
Fig. 7. User navigation system running on a pen-based tablet PC.

greatly reduce the time needed to develop networked applications in mobile computing settings. I also believe that the framework is a novel and useful application area of mobile agents and thus provides a significant contribution to mobile agent technology.

Finally, I would like to point out further issues to be resolved. The current implementation of the framework relies on the JDK 1.1 security manager. Although my framework should be used just as a development tool, I plan to design another scheme to perform security and access control. This framework does not support any disconnection operation or addressing scheme for mobile devices. These issues are left open for future work. Also, the current implementation supports two kinds of portable computing devices: notebook PCs and pen-based tablet PCs. However, the framework can basically support mobile agent-based emulators of any devices having JDK 1.1 or a later version, including Personal Java. I plan to support other devices, including personal digital assistants and information appliances. I presented a mechanism for dynamically customizing routing schemes for mobile agents in another paper [17] and am interested in applying the mechanism to the routing of my mobile agent-based emulator.

Acknowledgments. I would like to thank the anonymous reviewers for their making a lot of significant comments on an earlier version of this paper.

References


Appendix: Application Programs

As mentioned previously, each application is constructed as a collection of mobile agent-based components. Each component and each emulator is defined as an instance of a subclass of the abstract class Agent, which consists of some fundamental methods for mobility and inter-agent communication.

1: public class Agent {
2:   // methods for registering a collection of
3:   // service methods for its inner agents
4:   void addChildContext(Context c){...
5:   void removeChildContext(Context c){...
6:   // methods for registering listener objects
7:   // to hook certain events
8:   void addListener(AgentEventListener l){...
9:   void removeListener
10:   (AgentEventListener l){...
11:   ....
12:   Service getService(Message msg)
13:      throws NoSuchServiceException ... { ... }
14:   ....
15:   void send(AgentURL url, Message msg)
16:      throws NoSuchAgentException ... { ... }
17:   Object call(AgentURL url, Message msg)
18:      throws NoSuchAgentException ... { ... }
19:   void go(AgentURL url1, AgentURL url2)
20:      throws NoSuchAgentException ... { ... }
21:   ....
22:   }

Each agent can call public methods defined in its emulator by calling the `getService()`
method with an instance of the `Message` class that can specify the message type, arbitrary
objects as arguments, and deadline time for timeout exceptions. The `send()` and
`call()` methods correspond to the asynchronous invocation and method invocation of
the agent specified as `url1` respectively. Hereafter, I will describe some extensions of
the agent program of the MobileSpaces system to run on portable computing devices. The
go(AgentURL url1, AgentURL url2) method instructs the agent specified as `url1`
to move to the destination specified as `url2`.

1:   interface MobilityListener
2:      extends AgentEventListener {
3:      void create(AgentURL url); // after creation at url
4:      void leave(URL src); // before migration from src
5:      void arrive(URL dst); // after arrived at dst
6:      void suspend(); // before suspending
7:      void resume(); // after resumed
8:      void destroy(); // before termination
9:      ....
10:   }

As mentioned previously, each emulator (and its current context servers) can issue
specified events to notify its applications of changes in their life-cycle states. Like
Aglets [9], to hook these events, each application can implement a listener interface,
`MobilityListener`, which defines callback methods invoked by the emulator and the
runtime system at certain times. For example, suppose that a mobile agent-based emulator
is just about to migrate from its current host to another host. An application contained
in the emulator is notified by the following process:
1. The `leave(URL src)` method of the application is invoked along with the name of the current network to handle the disconnection from the network, and then the application is prohibited from connecting to any servers.

2. Next, the `suspend()` method of the application is invoked to instruct it to do something doing the suspension, and then it is marshaled into a bitstream.

3. The emulator moves to the destination as a whole with all its inner applications.

4. After the application resumes, its `resume()` method is invoked to do something.

5. The `arrive(URL dst)` method is invoked along with the name of the new current network to handle the reconnection to the network.
Crawlets: Agents for High Performance Web Search Engines

Prasanna Thati, Po-Hao Chang, and Gul Agha

University of Illinois at Urbana-Champaign
{thati,pchang2,agha}@uiuc.edu

Abstract. Some of the reasons for unsatisfactory performance of today’s search engines are their centralized approach to web crawling and lack of explicit support from web servers. We propose a modification to conventional crawling in which a search engine uploads simple agents, called crawlets, to web sites. A crawlet crawls pages at a site locally and sends a compact summary back to the search engine. This not only reduces bandwidth requirements and network latencies, but also parallelizes crawling. Crawlets also provide an effective means for achieving the performance gains of personalized web servers, and can make up for the lack of cooperation from conventional web servers. The specialized nature of crawlets allows simple solutions to security and resource control problems, and reduces software requirements at participating web sites. In fact, we propose an implementation that requires no changes to web servers, but only the installation of a few (active) web pages at host sites.

1 Introduction

Today’s search engines cover only a small fraction of the web, and the mean time between the modification (or creation) of a page and the time it is re-indexed by a search engine is several weeks long [11]. This under-performance is largely due to the centralized nature of search engine design, and the lack of cooperation between search engines and web servers. Today’s search engines crawl hundreds of millions of web pages across the network, all from one place (or at most from a handful of sites), generating one network transaction per page. This approach does not scale well. Further, most of the current web servers do not distinguish crawler requests from regular requests, despite the fact that there is a variety of site-specific information that can be made available to significantly improve the performance of search engines [2]. Given the scale of the web, any solution to these problems that requires modifications to web servers or excessive support from web sites, such as significant software installations, would be impractical. As usual, by a web server we mean servers such as Apache and IIS (Internet Information Service) that serve HTTP requests, and by a web site we mean the system including the machine and the software platform on which a web server is executed.

© Springer-Verlag Berlin Heidelberg 2001
We propose a solution that uses mobile agent technology to modify the way conventional search engines crawl the web. Our approach is not only scalable but also achieves the performance gains of web servers that export site specific information. It requires minimal modification to the search engines, and just the installation of a few active web pages (e.g. CGI programs or ASPs) at participating web sites. The specialized nature of our agents offers simple solutions for securing both web sites and agents against malicious attacks. Our solution does not assume complete cooperation from all web sites, instead provides the flexibility of varying degrees of cooperation from web sites, and seamlessly integrates conventional crawling of non-cooperating sites. We will also argue that there are strong incentives for certain types of web sites to support our scheme.

2 A Critical Analysis of Conventional Search Engine Design

The architecture of conventional search engines roughly consists of three components - a crawler, an indexer, and a searcher. The crawler starts with a set of seed URLs and repeatedly downloads pages, extracts hyperlinks from the pages, and crawls the extracted links. Since web pages may change, the crawler revisits previously crawled pages periodically. The crawled pages are stored in a repository which is accessed by the other components. The indexer parses each downloaded page to generate compact summaries, and constructs indices that map individual words to the page summaries they occur in. The searcher uses these indices to respond to search queries from users.

We identify some problems with the crawler component that affect the performance of search engines.

Centralized Architecture: The crawling strategy is highly centralized and therefore does not scale well. Usually, an HTTP request and reply are generated for every page downloaded by the crawler. Each of these requests and replies requires a separate TCP connection. Given that the web today has well over 1 billion publicly indexable pages [11], the latency involved in establishing these connections quickly adds up. Further, the estimated amount of data in all the web pages is of the order of tens of terabytes and continues to grow. Since a search engine has to frequently re-crawl web pages to account for any changes, the network bandwidth required is tremendous. Moreover, all the downloaded pages are processed locally to generate page summaries, which requires a lot of storage and processing power.

It is possible to meet these formidable challenges by using extra hardware. In fact, Google uses multiple crawlers to open hundreds of concurrent connections with web servers and downloads hundreds of kilobytes of data per second [13]. It uses thousands of networked PCs to process the downloaded pages. The problem with this approach is the high cost of hardware and software maintenance, including failures, power consumption, and hardware and software upgrades.

Inaccurate scheduling policies: Different web pages have different rates of change, types and extent of change, and relative importance. Since a search
engine has only a finite amount of computational resources, different strategies to revisit pages may result in different levels of freshness and importance of results it returns for search queries. The reader is referred to [4] for a good discussion of this problem.

Several scheduling strategies for crawling have been proposed in the recent past. In [5], the authors propose URL ordering schemes based on certain importance metrics for pages, that can be used to refetch important pages more frequently. The scheduling algorithm presented in [3] partitions pages according to their popularity and rate of change, and allocates resources to each partition using a custom strategy. The main drawback of these approaches is that either they do not account for significant parameters such as frequency and type of page changes, or they make unreasonable approximations. For example, the algorithm in [3] approximates page changes as memoryless Poisson processes, but a large class of pages such as those of newspapers and magazines change on a periodic schedule which is not Poisson. Bad estimation of these additional parameters increases the probability that a crawler fetches a page that has not changed since its last visit. Indeed, experimental data [2] show that almost 50-60% of revisits by a naive crawler may be unnecessary. In summary, the parameters required for efficient scheduling policies have a very broad range and are highly dependent on the pages being crawled and the web sites that host them. Given the variety of pages and sites, it is difficult to predict these quantities accurately.

**Incomplete and Unnecessary Crawling:** Due to resource constraints search engines crawl most web sites only to a certain depth. This contributes to incompleteness of crawling. Moreover, not all pages fetched by a search engine may be relevant. This is especially true for special purpose search engines, such as the media-specific engines which only look for specific types of documents (e.g. GIF or MP3 files). The only way a conventional crawler can discover these files is by finding links to them from other HTML documents it crawls. Obviously, fetching these additional pages is a costly overhead.

### 3 Related Work

The lack of customized interaction between web servers and search engines is addressed in [2]. The authors suggest modifications to the interaction protocol so that web servers can export meta-data archives describing their content. This information can be used by search engines to improve their crawling efficiency. The main drawback of this approach is that it requires modifications to the web servers. The enormity of the number of web servers raises some serious concerns about the deployment of such changes. Further, all the search engines and web servers have to agree on the syntax and semantics of languages used to describe the meta-data.

In [9], as an alternative to the centralized architecture of search engines, a push model is proposed in which individual web sites monitor changes to their local pages and actively propagate them to search engines. A simple algorithm is used to batch multiple updates into cumulative pushes. This reduces the load on
search engines by distributing it amongst several web sites. However, the push model introduces a new challenge, namely the task of coordinating the pushes from millions of web sites. Bad coordination could result in too many simultaneous pushes to the search engine. Furthermore, mechanisms to control excessive pushes from over-zealous and non-cooperating web sites have to be devised. This would require extensive infrastructure, and co-ordination that involves too many principals.

The Harvest project [1] is another promising proposal for a distributed infrastructure for indexing and searching information on the web. The Harvest system provides an integrated set of tools to gather, index, search, cache and replicate information across the Internet. The distributed nature of information gathering, indexing and retrieval makes Harvest very efficient and scalable. However, as in the case of push model, the main drawback of this system is that it requires extensive software installations and agreed standards across the Internet. For instance, multiple sites have to agree to host Harvest system components that cooperate with each other. In our opinion, such schemes are up against an enormous inertia.

4 Our Approach: Dispatching Crawlets to Web Sites

The basic idea behind our approach is quite simple (see Fig.1). Instead of crawling pages at a web site across the network the search engine uploads an agent, called the crawlet, to the site. The crawlet crawls pages at the site locally and sends back the results in a custom format.

![Crawling Diagram](image)

**Fig. 1.** A distributed approach to web crawling.

Using crawlets has a number of advantages. Typically, only one network transaction is required for crawling an entire web site, thus avoiding the latency in establishing a separate connection per page. Moreover, downloading large collections of compressed pages per transaction would result in significant bandwidth savings. Furthermore, the crawlet may send back pre-processed page summaries instead of raw pages, thus reducing bandwidth requirements. This also helps shift computational load from search engines to web sites with sufficient resources. Given the fact that there are about 3 million publicly indexable web servers [11], shifting small loads to many servers would cumulatively reduce
significant computational load at the search engine. Moreover, since the search engine can upload crawlets in parallel to several web sites and receive the results asynchronously, the crawling time is reduced by several orders of magnitude. All these savings would translate to reduced hardware requirements, reduced hardware and software maintenance cost, increased web coverage, and increased freshness of the search results.

Crawlets can achieve the performance gains of customized web servers such as those that export meta-data with site specific information. For example, a crawlet can discover pages that have changed significantly and ship only them back to the search engine. For media specific crawlers, the crawlet can discover and ship only the relevant media files. In both these cases, although the total time spent on processing pages is the same, the savings in network transactions are significant. There is no need for web servers to export meta-data, and for search engines to use complex scheduling heuristics (which often rely on unknown parameters) for revisiting pages. Modifying web servers to export meta-data as in [2], is only a temporary solution and does not work well in practice. Any such change will have to wait for server updates before it can be used. In contrast, once there is an agreement on the execution environment for crawlets search engines can upload any clever or personalized foraging crawlets.

Crawlets are not general information retrieval agents such as those described in [14], which navigate through the network interacting and coordinating with other agents and services to find relevant information. Instead, they are very specialized agents which do not communicate with other agents and once uploaded to a site do not migrate any further. Moreover, due to the non-critical nature of crawling, crawlets typically do not require guarantees such as persistence or fault-tolerance. As a result they do not need a general purpose agent platform. We propose an implementation that requires no modifications to web servers, but only the installation of a few active web pages at participating sites.

There is a good incentive for certain types of web sites to support our scheme, especially the ones that are not regularly crawled and indexed by search engines, either because their contents change too frequently or they are considered unimportant. For instance, the search results returned by Google rarely contain pointers to (even the major) news sites. On the other hand, security is a major issue for both hosting web sites and crawlets. Hosts are to be protected from threats such as unauthorized disclosure or modification of information and denial of service attack. Dually, the crawlet has to be protected from unauthorized tampering of its code and data. Specifically, a host can modify the crawlet output in order to improve the number of hits to its pages in response to search queries.

The security measures required should be simple and computationally inexpensive. Otherwise they will neutralize the benefits of our scheme. As we will see in Sect.7, security can be enforced by simple sandboxing mechanisms. This is because crawlets require very limited and predictable access to system resources. Similarly, there are computationally inexpensive techniques to protect a crawlet from malicious hosts.
We recently discovered an independent work that proposes the idea of using mobile agents for efficiently crawling the web [10]. A stand alone application is presented that allows one to launch crawler agents to remote web sites. The application also provides powerful querying and archiving facilities for examining and storing the gathered information. In our opinion, using such applications to implement web search engines has several drawbacks. First, the approach does not integrate well with conventional search engine design. Search engines typically employ highly customized information storage and retrieval techniques specifically suited for their purposes, instead of those imposed by the application. A stand alone application is best viewed as a platform that supports small scale web search applications. In contrast, we present a tightly integrated implementation (see Sect.5) that involves a few simple modifications to the crawler component, and leaves ample room for search engine specific customizations. Second, security and fine-grained resource control issues that arise due to code migration, have not been addressed in [10]. In fact, these still are open problems for agents with multi-hop itineraries such as those used by their application. Third, their application imposes excessive software requirements on participating web sites, such as rule based inference engines and sophisticated communication mechanisms. These software installations are also heavy on the CPU, and may thus discourage web sites from participating.

5 Modifications to the Search Engine

Our scheme requires modifications to only the crawler component of a search engine. To simplify the discussion, we only show modification to a naïve crawler which does not employ sophisticated scheduling policies such as those mentioned in Sect.2. However, our discussions can easily be adopted to include them.

\begin{verbatim}
extract(Q,h) : Returns URLs in queue Q that are at h
dispatchCrawler(h,F,U) : Sends a crawler to h along with URL lists U and F. Returns the URLs crawled, and the outgoing links found by the crawler
crawl(u) : Downloads the page pointed to by u and returns all the links in the page

FQ = TQ = emptyQueue
enqueue(FQ,seedURLs)
while (true)
    TQ = FQ; FQ = emptyQueue
    while (TQ not empty)
        u = dequeue(TQ)
        if (site(u) supports crawlers)
            U, F = extract(TQ,site(u)), extract(FQ,site(u))
            (a,o) = dispatchCrawler(site(u),F,U)
            enqueue(FQ,a); enqueue(TQ,o-FQ)
\end{verbatim}
else
    l = crawl(u)
    enqueue(FQ,u); enqueue(TQ,l-FQ)
end if
end while

The modified crawler shown above maintains two URL queues - \( FQ \) which contains the URLs that have already been crawled, and \( TQ \) which contains URLs that are yet to be crawled. Before downloading the page at a URL \( u \), the crawler checks if the corresponding web site \( \text{site}(u) \) supports crawlets. Web sites that do, export an active page at a fixed relative URL, say \texttt{srch-eng/crvlt-ldr.cgi}. Thus, before downloading \texttt{http://osl.cs.uiuc.edu/foundry/index.html} the crawler contacts \texttt{http://osl.cs.uiuc.edu/srch-eng/crvlt-ldr.cgi}. If this fails it resorts to conventional crawling. Otherwise, it uploads a crawlet to the web site. Details of interactions with the active pages are described in Sect.6.

The crawlet carries two lists of URLs that belong to the remote site \( \text{U} \) which contains seed URLs, and \( \text{F} \) which contains URLs that have already been crawled. These lists are extracted from \( TQ \) and \( FQ \) the obvious way. The crawlet crawls pages at its host in the conventional style, except that it does not crawl a newly discovered link if it points to a remote site. The crawlet ships back a list of URL/page-summary pairs that correspond to crawled links and a list of outgoing links it did not crawl. Further details about crawlet implementation and related issues such as security are discussed in Sect.7.

The crawlet may not be able to discover all the pages at the web site in its first visit. For example, consider the graph of pages shown in Fig.2. An arrow between two pages means that the source page contains one or more links to the destination page. Suppose the crawler only knows the URL of \( w_{13} \). The crawler dispatches a crawlet with \( w_{13} \) as the seed URL. The crawlet replies back with \( w_{11} \) and \( w_{12} \) as the URLs crawled, and \( w_{21} \) and \( w_{22} \) as the outgoing URLs found. Note that the crawlet cannot reach pages \( w_{13} \) and \( w_{14} \) in this visit. But later when the crawler gets to crawl \( w_{22} \), it discovers the URL of \( w_{13} \). In the second visit the crawlet can crawl the remaining pages. In general, to avoid multiple visits to a site, the crawler may dispatch a crawlet with a larger set of seed URLs that were discovered in its previous iterations.

In the crawler algorithm shown above, once the crawlet is uploaded to a web site the crawler does not crawl any further until it hears back from the crawlet. Experimental results (see Sect.8) show that even in this simple case, the crawler is significantly faster than the conventional one. The crawling can be readily parallelized to obtain further speed ups by dispatching multiple crawlets to different sites and asynchronously waiting for their replies.

6 Uploading Crawlets

We now discuss the protocol between search engines and the (active pages at) web sites that is used to upload and execute crawlets. The security concerns that
Fig. 2. A graph of web pages residing at three web sites

arise include mutual authentication between the search engine and the web site to prevent malicious masquerading, and the integrity and confidentiality of the transaction data. It is reasonable to expect the web site to execute the crawler under certain resource constraints. The web site may adopt different resource allocation policies for different search engines, which may also vary with time depending on local system state. It is useful to convey these resource constraints to the search engine so that it may upload a customized crawler.

Following is a simple protocol built on top of HTTP (see Fig. 3) that implements these requirements. It uses public key cryptography for authentication, and shared key encryption for secure transmission of data.

Fig. 3. The protocol used to upload and execute crawlers.

**Request 1:** An HTTP request from the search engine to the web site. The request payload contains a randomly generated session key $K$ and is encrypted with the secret-key $S_e$ of the search-engine.

**Reply 1:** The crawler receives a reply containing a preamble confirming that the receiver understands the protocol, and an authentication key and a time nonce that should be used as a ticket for subsequent transactions. The reply also specifies the resource limits (such as CPU time, memory and disk space)
that would apply to the crawllet. The payload is encrypted twice, first with the secret key $S_e$ of the web site, and next by the session key $K$.

**Request 2:** An HTTP request from the search engine. The payload consists of the authentication key, nonce, and the crawllet along with its data. It is encrypted with the session key $K$.

**Reply 2:** An HTTP reply that contains the output that the crawllet writes to its standard output. The output stream is not interpreted by the web site. The stream can also be interrupted by an error message, which could be because of crawllet errors, or host-initiated crawllet abortion due to violation of security or resource constraints. The stream is encrypted with the session key $K$.

Due to the non-critical nature of the crawllets and the information they gather, typically there are no strict fault-tolerance or consistency requirements on the uploading and execution of crawllets. The protocol described above has been kept simple to minimize software requirements at web sites. However, it can be extended in several useful ways. The web site can furnish details about its local platform such as the type of its HTTP server, operating system, and the organization of web pages in the local file system. Such information can be used by the crawler to dispatch customized crawllets that crawl pages much more efficiently. There could also be a more elaborate negotiation between the crawler and the web site for resources. One possibility is to have the web sites 'sell' their computational resources in exchange for increased importance that the search engine associates with its pages while processing search queries.

7 Executing Crawlets

Once the crawllet is uploaded, it is executed by its host (active pages) in a controlled environment. Following is a simple algorithm for crawllets.

```plaintext
crawl(u) : Crawls the page pointed to by URL u and returns the list of URLs it points to and its pre-processed contents.
classify(l) : Classifies the URL list l into two lists: one containing URLs at the local site, and the other the remaining.

FQ, TQ, OQ = previouslyCrawledURLs, seedURLs, emptyQueue
while (TQ not empty)
  u = dequeue(TQ); (c,l) = crawl(u)
  enqueue(FQ,u)
  (o, i) = classify(l)
  o = o - OQ
  output((u,c),o)  // write to stdout
  enqueue(OQ, o), enqueue(TQ,i-FQ)
end while
```
The crawlet crawls local pages as usual by making HTTP requests. If the web server is not already optimized for such local requests the host can intercept these requests and directly access the file system. Of course, this only works for static HTML pages and not for dynamically generated ones, and it also requires the host to know the mapping between URLs and path names which may vary from web server to web server. Note that the crawlet periodically writes partial results to its standard output, which is continuously redirected back to the search engine by the host. This not only saves storage space for the crawlet, but also is handy if the crawlet does not get to complete its execution.

The security problem of protecting both hosts and crawlets against malicious attacks is considerably simplified since the crawlet visits only one host. Recall that, conventional security mechanisms break down for general mobile agent applications primarily because of their unrestricted mobility. The problem with unrestricted mobility is that hosts in the agent’s itinerary need not trust each other. Even if an agent is initially signed by its source, an intermediate host can alter its code or state in order to make it malicious. It is generally difficult for a receiving host to determine if the agent has been tampered with. Similarly it is difficult for an agent to determine if its execution environment at a host is un-tampered. The current approaches for securing agents that travel more than one hop include sophisticated approaches such as carrying proofs of code safety [12], maintaining agent state appraisal [6], and maintaining path (itinerary) histories [16].

A single hop itinerary and limited computational facilities required by the crawlets greatly simplify the problem of securing the hosts. Well known techniques used in conventional settings without migrating code are sufficient. The authentication and secure loading of crawlets was described in Sect.6. During its execution the crawlet only needs permission to make HTTP requests to the local web server, and use specific folders in the file system as scratch space. It neither needs to communicate with other agents nor access other system resources. The host is secure if it grants only these permissions to the crawlet and enforces the resource limits negotiated with the search engine. These security measures are easily realized using the sandboxing technique [7].

It is also important to protect crawlets from malicious hosts which may tamper its output. Although there isn't much incentive for a host to not forward crawlet outputs to the search engine, it may want to modify output such as the pre-processed page contents to improve its popularity. However, this problem is not unique to our scheme. Because web sites can distinguish search engine requests from others, they can forge replies with as much ease in the conventional setting. In any case, the tampered data is not critical enough to cause irreversible damage.

The possibility of hosts tampering with crawlet output is inconvenient enough to look for prevention mechanisms. A simple idea is to secure the crawlet’s output stream by encrypting data along with embedded keys. Shared key encryptions are computationally very cheap and a fresh key is generated every time a crawlet is uploaded. But this solution does not prevent the host from interfering with the
execution of the crawlet, in particular with the encryption process itself. A well
known solution to this problem is to encipher the encryption function itself [15].
Suppose the crawlet is to use the function \( f \) to encrypt its output, but \( f \) is to be
kept a secret. The search engine transforms \( f \) to some other encryption function
\( E(f) \) that hides \( f \). The program \( P(E(f)) \) that implements \( E(f) \) is embedded in
the crawlet. The host can therefore only learn about \( I(E(f)) \) which is applied
to produce the encrypted output. The search engine decrypts this output to
obtain \( f(x) \). Thus, once a suitable candidate for \( E(f) \) is known this strategy is
straightforward and is computationally inexpensive. In summary, the simplicity
of crawlets enables us to enforce security measures that are computationally
inexpensive and do not neutralize the benefits.

We end this section with a brief note on strategies the search engine may
adopt if the resources allocated by a web site are insufficient. The pages at a site
can be logically organized as a tree or a forest based on the path components
of their URLs. In the presence of insufficient resources the search engine can have
its crawlet crawl only a few subtrees in the forest. The crawlet would simply
treat any link not pointing to a page in the subtrees as an outgoing link. The
number of pages and their total size in a large subtree typically vary very little
with time and hence, based on previous experience, it is feasible to estimate the
resources required to crawl them. If the estimates are inaccurate, and the crawlet
is unexpectedly short of resources it can ship its state back to the search engine
so that crawling can continue either in the conventional style or through another
crawlet.

8 Performance Measurements

The primary focus of our experiments is to measure the performance gains of
crawling a single site with different types of crawlets in comparison with con-
ventional crawling. An important factor in the experiments is the choice of web
sites. According to web statistics, the average number of pages per web site is
about 500 [11]. Furthermore, most of the sites host very few pages and only a
small portion host a very large number of pages. So a web site with about a
few thousand pages is a good choice. We conducted experiments on three such
sites (see Table 1). However, the sites vary considerably in properties such as the
number and size of pages and their linkage structure. To control the experimen-
tal environment, we mirrored the sites onto a web server which does not host any
other pages. This is essential to isolate our measurements from irrelevant vari-
ations resulting from factors such as differences in hardware and unpredictable
load at web servers due to the regular request traffic (that is load not generated
by the experiments).

The active pages that receive and execute crawlets were implemented using
Microsoft ASP. These pages implement the protocol described in Sect. 6. They
assume that crawlets are implemented in Java. In general, such assumptions
and details of the execution environment can be conveyed while uploading the
crawlet. The pages use the built-in sandboxing facility of Java runtime environment for security control.

For our experiments, only the crawler component of the search engine is relevant. We implemented two versions of the crawler: one that crawls in the traditional style, and the other that uses crawlets. Both the crawlers were implemented in Java. The conventional one runs a few crawling threads (which download pages) and parsing threads (which process the pages to extract links) in parallel. The other crawler dispatches crawlets to web sites using the protocol described in Sect. 6. Java's object serialization and security features were used for dispatching the crawlets. To get a fair comparison, crawlets were implemented using the same crawling and parsing mechanisms as the conventional crawler. We experimented with 4 different types of crawlets which vary on how they crawl local pages - using HTTP requests or through the file system\(^1\), and how they transmit the results back - compressed or uncompressed. The compression is implemented using java.util.zip package. The crawlets not only ship back raw unprocessed pages but also the URLs that these pages point to. This means that the crawlets which do not use compression will be shipping more data than in conventional crawling. We did this to get a fair comparison because after crawling a site the conventional crawler will have extracted these URLs which may then be used by other components of the search engine.

The web server we used is Microsoft IIS version 5.0. It was executed on a Pentium III 450 with 192 MB RAM and the server version of Windows 2000. To get a fair comparison, the crawler was executed on a machine with exactly the same configuration. The experiments were conducted in two different settings: one with both the crawler and server executing on the same LAN, and the other with the two in different domains with an effective connection bandwidth of more than 1 Mbps. The available network bandwidth in both configurations was more than what was required by the crawler and crawlets. Thus, the difference is almost exclusively in the network latencies. However, when using crawlets there wasn't much difference in the measurements for the two configurations. This confirms the fact that network latencies are almost completely masked. So we present their results only for the WAN configuration.

\(^1\) As mentioned in Sect. 7, this is actually an optimization in the host execution environment. There is no change to the crawlet code.
We measured four parameters: total time taken to crawl a web site, number of bytes transferred between the crawler and the server, load on the crawler machine, and load on the web server machine. The total time taken for different configurations is shown in Fig. 4. The data confirms a number of our speculations in the previous sections, which were primarily about the WAN configuration.

- All the four types of crawlers outperform the conventional crawler.
- The crawlers which compress their results further reduce the crawling time.
- The optimization of redirecting HTTP requests of a crawler directly to the file system reduces the crawling time.

We observed an interesting phenomenon in the LAN configuration. The conventional crawler performs almost as good as crawlers (outperforming in some cases). This is because in a LAN network latencies are negligible and the extra time taken by crawlers for compression or shipping additional data is amplified. But this phenomenon is not very significant in real world because all the interactions while crawling the web are over the WAN. In summary, using crawlers reduces crawling time. This translates to reduced mean-time of revisiting pages, which in turn implies improved freshness of results for search queries.

![Graph showing total crawling time for different configurations.](image)

**Fig. 4.** Time taken to crawl a site.

Figure 5 shows the total number of bytes transferred between the crawler and the web site. As expected, for the crawlers that do not use compression the amount of data to be transferred is more than in the conventional case. This is
especially amplified for sites with several small pages, each pointing to a number of other pages, as for the ACM site (Table 1). But this is easily remedied using the compressing crawlet. Since the pages are mostly plain text, compression has a huge impact. For example, for the ACM site the data transferred is reduced by more than 70%. These numbers can be improved further if the crawlet ships back only page summaries, although this means more computational load on its host.

![Diagram](image.png)

**Fig. 5.** Total number of bytes transferred between the crawler and a web site.

Figure 6 shows the CPU load on the crawler and web server machines in different settings. In the interest of space we have shown the numbers for only the engineering site which hosts the largest amount of data amongst the three sites (Table 1). In all cases, most of the user time is spent on processing pages, while most of the kernel time is spent on I/O operations. The graph is composed of three parts one each for the crawler and web server machines, and another for the total load which is the sum of the first two parts.

It is obvious that there is significant shift in computational load from the crawler to the web server while using crawlets. However, the combined load at the crawler and the web server is less than in the conventional case. This saving is primarily because of the reduction in number of network transactions required. Another observation is the effect of compression of the crawlet output. The increase in user time because of compression is less than the decrease in the kernel time because of reduced I/O operations, thus reducing the total CPU load. This is because the compression rate is very good for plain text.

Thus, we have demonstrated substantial reduction in crawling time, computational load at the search engine, and network transactions on using crawlets.
9 Conclusion

We have proposed the use of mobile agents to improve the performance of web search engines. The performance gains translate to improved web coverage and freshness of search results. Implementations show that our scheme has minimal software requirements. In fact, it only requires the installation of a few web pages at participating sites, thus simplifying deployment. The experimental results clearly demonstrate the performance gains. Due to its simplicity our proposal does not introduce new security concerns. Security can be enforced by simple conventional techniques which are computationally inexpensive. We believe that there is a strong incentive for several web sites to support our scheme, especially the ones that are rarely indexed by search engines. Given that today’s search engines cover only 12% of the web [11], there could be a large number of such sites.

An interesting research in the context of our work is on integrating agent platforms into web servers, such as in the WASP project [8]. Although promising, this idea has not yet gained a wide acceptance due to several reasons, including elaborate software installations required, security concerns, and incentive barriers. However, if accepted, our scheme can be readily integrated into such platforms in the form of services.
References


An Efficient Mailbox-Based Algorithm for Message Delivery in Mobile Agent Systems

Xinyu Feng, Jiannong Cao, Jian Lü, and Henry Chan

1 State Key Laboratory for Novel Software Technology
Dept. of Computer Science, Nanjing Univ.
Nanjing, China
fxyn@softlab.nju.edu.cn, lj@nju.edu.cn

2 Internet Computing and E-Commerce Lab
Dept. of Computing, Hong Kong Polytechnic Univ.
Hung Hom, Kowloon, Hong Kong
{csjcao | chchan}@comp.polyu.edu.hk

Abstract. Agent mobility presents challenges to the design of efficient message transport protocols for mobile agent communications. A practical mobile agent communication protocol should provide location transparency to the programmer and thus need to keep track of the movement of an agent. In addition, because of the asynchronous nature of message passing and agent migration, how to guarantee the delivery of messages to highly mobile agents is still an active research topic in mobile agent systems. In this paper we propose an efficient mailbox-based algorithm for inter-mobile agent communications. The algorithm decentralizes the role of the origin (home) host in locating an agent. Furthermore, by separating the mailbox from its owner agent, the algorithm can be made adaptive and is efficient in terms of location updating and message delivery. In the cases that mobile agents migrate frequently but seldom communicate, our algorithm turns out to be preferable.

1. Introduction

In recent years, mobile agent computing has emerged as a new paradigm in developing applications in various areas including telecommunications, networking/distributed systems, and e-commerce. Mobile agents are active, autonomous objects or object clusters, which are able to move between locations in a so-called mobile agent system. A mobile agent system is a distributed abstraction layer that provides security of the underlying system on one hand and the concepts and mechanisms for mobility and communication on the other hand [1, 2].

Mobile agents used in various applications need to communicate with each other for different purposes such as exchanging information and/or co-operation [3, 4]. Although process communication has been a cliché in the research of distributed systems, agent mobility poses a number of problems in designing message delivery

* This research is partially supported by Hong Kong Polytechnic University Research Grant A-PC53.

© Springer-Verlag Berlin Heidelberg 2001
mechanisms for effective and efficient communications between mobile agents. Since a mobile agent has its autonomy to move from host to host, it is unreasonable, if not impossible, to require that agents have a priori knowledge about their communication partners’ locations before they send messages. Therefore, the first requirement of a practical mobile agent communication protocol is to allow mobile agents to communicate in a location transparent way, i.e., an agent can send messages to other agents without knowing where they reside. On the other hand, the asynchronous nature of message passing and agent migration may cause the loss of messages being sent to an agent on its move. Thus, a reliable agent communication mechanism should also guarantee the delivery of messages to highly mobile agents. Besides, the agent location tracking and message routing algorithm should not introduce too much overhead or offset any of the merits of mobile agent technology.

Many currently available mobile agent systems do not provide solutions to these problems and leave the hard nuts to agent programmers [1, 5]. Although there are several protocols proposed trying to provide location transparency and reliable inter-agent communication [7–10], they either handle it in a way too complicated to be efficient in practical systems, or use home-registration and rely too much on agent home, which is improper when a disconnected execution is needed.

The mailbox-based algorithm proposed in this paper adopts a hybrid approach combining the registration and forwarding schemes to locate mobile agents and deliver messages. Using the algorithm, messages can be delivered in a reliable and location transparent way. By forwarding the message at most once, the algorithm resolves the problem that messages keep chasing their highly mobile target agents. Unlike the home registration method used in mobile computing, e.g., Mobile IP [11], the algorithm decentralizes the role of the origin (home) host in locating an agent. This reduces the reliance on a single host, so that the agent’s ability to support disconnected operation, considered as an important advantage of mobile agent technology [12], can be achieved in a real sense. Furthermore, by separating the mailbox from its owner agent, the algorithm can be made adaptive and is efficient in terms of location updating and message delivery.

The remaining of this paper is organized as follows. Section 2 presents a brief review of related work. Section 3 describes our mailbox-based algorithm in detail and also presents a proof of its properties. In Section 4 we analyse the performance of the proposed algorithm. Section 5 describes the simulation results and discusses the relationship between the communication overhead and mailbox migration frequency. The final section provides some concluding remarks.

2. Related Work

To communicate with a remote mobile agent, we must find the location of the agent and route the message to it. A naming scheme is needed to identify agents in a unique fashion. The name should not change whenever the agent migrates to other hosts and it is up to the tracking algorithm to map the name to the agent’s current address. The routing process can be done either in parallel with agents tracking [9] or in a second phase after the address has been got [7].
The usual way to name an agent [7, 9] is to append the address of an agent’s origin host (i.e. agent home) with its title (a free form string used to refer to this agent). Thus it is impossible for agents born at different agent platforms to have the same name. For agents created at the same host, the origin host is responsible to manage the name space to ensure that each agent has a unique title. In this paper we adopt this naming scheme.

There are three basic schemes for locating agents, namely searching, logging and registration [5]. In the searching approach, we either send an agent to visit every host that the target agent might reside in or broadcast locating messages to these hosts [8]. The overhead is unaffordable when the network is large. The logging method locates the mobile agent by following the trail information indicating its next destination, left in every host the agent has ever visited [9]. If the trail information is lost or if one of the hosts is down, the target agent would no longer be found. With the registration scheme, an agent needs to update its location in a predefined directory server (e.g., its home host) that allows agent to be registered, deregistered or located. The directory server can be either a central node, which may become the bottleneck of the system performance and/or a single point of failure, or the agent’s home host, which follows the idea of Mobile IP [11].

Two common methods for message routing are forwarding and locate-and-transfer. Under the forwarding scheme (also called path-extension), locating a receiver agent and delivering a message to it are both done in a single phase. They are combined into one operation where an agent moved to a new host informs the previous resident host where it moves so that messages can be forwarded along the extended path. The disadvantage is that messages may take multi-hops before they reach the target agents. The performance is worsened when messages are large in size. On the other hand, locate-and-transfer locates the target agent first and then transfers the message directly to it. However, the message sender may get outdated address in cases that the receiver agent migrates frequently.

The Mobile IP [11] is the protocol designed for IP packets routing to mobile devices. A mobile host registers its care-of-address with its home host and it is the home host that forwards the IP packets to it. Although this home registration and forwarding method is easy to implement and has less location registration overhead, it is inappropriate in mobile agent systems. Since all the correspondents of an agent must find its address form its home host, the agent home host may be a performance bottleneck when a larger number of agents, each with many correspondents, are originated from that same host. Besides, the agent home host may sometimes break off from the network after the agent is dispatched. Disconnected computing cannot be supported under this scheme.

Among the mobile agent systems and programming environments that are currently available, few provides practical and efficient algorithms for mobile agent communications. In Mole [1] there is no solution to location transparent remote communication. An agent must give the current address of its correspondent explicitly in order to send a message. Aglets [5, 6] attempts to provide location-transparency via Aglet proxy, but the system does not provide APIs to support Aglet tracking. To
obtain the receiver’s proxy, Aglets programmers must implement tracking mechanism by themselves.¹

The Mogent system [7] proposed a reliable inter-agent communication algorithm. All the agents need to register their locations with their homes. Before sending a message to another agent, the sender agent queries the recipient’s current address from the target agent’s home host. If the target agent is currently moving across the network, the reply to the location query is pending until the target agent registers its new location. Before an agent can move, it needs to ask for permission from the home host. If there are messages on their way targeting at the agent, the agent need to wait until these messages arrive. It is the responsibility of the agent home to synchronize the migration of the agents and the message passing. In this way, reliable message delivery can be guaranteed and no message forwarding is needed. However, the algorithm depends so much on the agent home that the agent cannot move and communicate if their home is down or disconnected.

Murphy and Picco [8] present a broadcast-based mobile agent communication scheme which is similar to a distributed snapshot. The scheme guarantees transparent and reliable inter-agent communication, and can also provide multicast communication to a group of agents. However, to search for the message recipient, it requires to contact every node in the network that has been visited by at least one agent, and thus generates an amount of traffic that is comparable to a broadcast. Same as in the “searching” scheme mentioned above, the traffic overhead is unaffordable when there are a large number of hosts and agents in the network.

In [9] a hierarchical infrastructure is proposed to name agents and to route messages. All the hosts in the network are organized into a tree. The agent moves along the nodes of the tree and on every node leaves a pointer to the next one in the path. Messages are forwarded along the same path according to these pointers. However, the hierarchy cannot always be easily constructed, especially in the Internet environment. Instead of sending the messages or agents directly to their targets, unnecessary hops need be taken along the tree. Besides, under this scheme, messages may be missed by their recipient agents and need to keep chasing the recipient.

A resending-based TCP-like message delivery mechanism, called MStream, for mobile agent communications is introduced in [10]. The mechanism assumes that losses and failures are possible in the network. MStream is the communication endpoint that can be moved from host to host. When an MStream moves, a Location Manager will broadcast its new location to all other MStreams. If a message is sent to an outdated address of the target MStream, it will be retransmitted several times before the sender sends it to the Location Manager to be forwarded to the new destination. The paper does not mention how to avoid multiple forwarding for highly mobile agents.

Our mailbox-based algorithm adopts a hybrid approach combining the registration and forwarding schemes. It realizes location-transparency and ensures the message delivery. Under this communication scheme, most of the messages are sent to their recipients directly and others are forwarded at most once before they reach the

¹ In the latest release of Aglets system, namely ASDK V1.1 Beta3 [6], the MASIF interface MAFinder was implemented over Java RMI. With the cooperation of the finder and the aglet server, the proxy of a remote aglet can be obtained in a location transparent way. However, there is no guarantee for message delivery. If the target aglet moves away, the message sending procedure will fail and an AgletNotFoundException will be thrown.
receiver agents. Besides, the movement of agents can be separated from that of their mailboxes, thus, by deciding adaptively when to move the mailbox to its owner agent, we can reduce the traffic overhead greatly. The details of our algorithm will be discussed in the next section.

3. The Adaptive Mailbox-Based Routing Algorithm

As we have discussed in Section 2, the home registration and forwarding method adopted by Mobile IP cannot be borrowed blindly by mobile agent systems. One possible solution to reducing the dependence of agent communication on agents’ home hosts is to decentralize the role of the home host in locating a target agent. The responsibility of agent tracking is distributed to all the hosts (called “past hosts” hereafter) on the path traveled by the migrating agent. The location of the migrating agent is kept by all the past hosts. Once the agent has arrived at a new host, it multicasts its location to all the past hosts. This can reduce the agent tracking cost. However, the location updating cost can be too much to be acceptable if the agent visits a great number of hosts during its life cycle. As a matter of fact, in many applications, agents migrate from one host to another without communicating with others. In these cases, it is superfluous for agents to multicasting their locations to all past hosts. If we can find an adaptive way to avoid the superfluous address registration, the traffic overhead will be decreased considerably. As we will see, our mailbox-based algorithm can accomplish this goal by detaching the mailbox from its owner whenever possible.

3.1 System Model and Assumptions

In our system model, we assume that mobile agent communication is largely based on asynchronous messages. This is reasonable because, when mobile agents roaming the Internet, it is undesirable that two agents use synchronous communication to talk to each other [13], due to the large and unpredicted delays on the Internet, which can easily become several seconds.

A mailbox is a message buffer used to store incoming messages. Every mobile agent in the system is allocated a mailbox. Incoming messages sent to the agent are inserted into the mailbox first. Two modes of message delivery can be supported: Push and Pull. In the push mode, messages stored in the mailbox will be delivered to the mobile agent, while with the pull mode, the agent fetches messages from its mailbox any time it decides to do so. In this paper, we use the pull mode. A mobile agent is automatically initialized to check its mailbox whenever necessary. If the mailbox contains any messages, these messages are delivered. Otherwise, either a synchronous or an asynchronous receive operation can be implemented - the mobile agent can continue its execution or is suspended until a new message arrives. We assume that the send operation is always asynchronous (a synchronous send can always be simulated by letting the sending agent, after it has put the message in the message system, change to a receiver and wait for an acknowledgement).
As shown in Figure 1, in our algorithm, the mailbox can be detached from its owner agent in the sense that an agent and its mailbox can reside at different hosts. An agent can migrate to a new host while leaving its mailbox at a previous past host. When an agent $M_i$ sends a message to another agent $M_j$, $M_i$ sends the message to the host where $M_j$'s mailbox currently resides (Step (1) in Figure 1). The agent $M_i$ sends a request to its mailbox to fetch message (Steps (2) and (3) in Figure 1). Since the location of mailbox is unchanged, location updating is avoided and a considerable message passing cost can be saved. In location updating, the meaning of “past hosts” is also changed. It no longer refers to the hosts on the path of the migrating agent, but the hosts where the mailbox once resided, which may be much fewer in number. Thus the number of hosts that keep the agent’s location information is decreased and the overhead of location updating is further reduced.

An address table is maintained in every host to record current addresses of mailboxes that have ever resided at this host. A “valid” tag is associated with every entry of the table to show whether the corresponding mailbox address is outdated. Another field in an entry is a blocked message queue, which is used to temporarily keep the messages sent to the corresponding mailbox if the “valid” tag is false, i.e. the mailbox is moving on its way to a new host.

We assume that our algorithm is built on a set of low-level location-dependent communication mechanisms, which can be directly implemented above standard network protocols using asynchronous and point-to-point messages [14]. It is assumed that these mechanisms abstract away the network failure for our high level location-independent algorithm. They also maintain the FIFO order of message delivery, which is critical to the proper execution of our algorithm. As Murphy and Picco indicated in [8], their algorithm also requires the FIFO property, which can be implemented straightforwardly in a mobile agent server by associating a queue that contains messages that must be transmitted to a remote server.

3.2 The Algorithm

The algorithm works in two phases, location-updating and message-routing. In the location-updating phase, if the agent decides to migrate with its mailbox, it will first de-register its mailbox address and then re-register the new address with all the past hosts after it reaches the new destination host. In the message-routing phase,
messages are sent to the recipient’s address cached by the sender. If the recipient has been moved to another host, the messages will be forwarded to the current address. The algorithm is presented more formally in pseudo code in the rest of this section.

![Diagram of mailbox migration and registration](image)

**Fig. 2.** Mailbox migration and registration

**Location Updating.** Before moving, the agent determines whether to migrate its mailbox to the new host. It sends a “MVMB” message to its mailbox host if it decides to do so. The “MVMB” message contains the address of the destination host that the agent is to migrate to (Step (1) in Figure 2). The pseudo code of this operation is shown in the function `OnMigration_Agent()`.

```
OnMigration_Agent(){ //executed by agents before moving
    if(fetchMailbox()){ //the agent decides to go with its mailbox
        String nextAddr = itinerary.getNextHost();
        sendMsgToMailBox("MVMB", getMBAAddress(), nextAddr); //the underlying location-dependent primitive
    }
    migrateTo(nextAddr); //migrate to the target host, Step (2)’ in Figure 2
}
```

On receiving the “MVMB” message, the mailbox host executes the function `ProcessMVMBMsg_MB()`. It sends “Deregister” messages to all the hosts on its path, including the local host (Step (2) in Figure 2). After it has collected the “REPLY” messages from all the hosts, or when time out, it migrates the specified mailbox to the destination host (Step (4) in Figure 2).

```
ProcessMVMBMsg_MB(msg){ //executed by mailbox
    path = getPath();
    for(every host on the path) //including the local host
```
SendMsgToMAP("DEREGISTER", everyHost, localHost);
wait until all REPLY msgs from these hosts arrive or
time-out;
targetHost = msg.getContent();
  //get the address of target host
migrateTo(targetHost);
}

On arriving at the new host, the agent starts executing the function
OnArrival_Agent() by checking whether its mailbox has been moved to the
new host with it. If not, it does not need to register its new address. Otherwise it sends
a “REGISTER” message to every past host where its mailbox has resided (Step (3) in
Figure 2).

OnArrival_Agent() //executed by the agent
{
  if (migrated without mailbox)
    return;  //do nothing
  setMBAAddress(localAddress);
    //update the address of its mailbox
  append the localhost into its mailbox’s path;
  for (every host on the path of the mailbox)
    SendMsgToMAP("REGISTER", everyhost, localAddress);
}

The mobile agent platform (MAP) in a host is responsible of processing all the
count messages. Its operation is illustrated in MessageProcessing_MAP() shown below.

MessageProcessing_MAP(msg) //executed by MAP
{
  switch(msg.getKind()){
    case DEREGISTER:
      AddressEntry entry =
          addressTable.getAddr(msg.getSender());
      entry.VALID = false;
      sendMsgToMailBox("REPLY", msg.getContent(),
null);  //step (3) in

Figure 2.
  case REGISTER:
    AgentID sender = msg.getSender();
    AddressEntry entry =
        addressTable.getAddr(sender);
    if (entry == null){
      // REGISTER msg is from the local host, create a
      //new entry in address table for sender’s
      address.
        entry = new AddressEntry(sender);
        insert entry into the local address table;
    }
entry.VALID = true;
entry.address = msg.getContent();
while (there are messages in the block queue for
sender) {
    Message blockedMsg =
        entry.blockQueue.getNextMsg();
    sendMsgToMAP("AGENTMSG", entry.address,
        blockedMsg);
    sendMsgToMAP("UPDATE", sender_of_blockedMsg,
        entry.address);
    //update the address cached by the sender
} //end of while
entry.blockQueue.clear();
} //end of switch

\[ 
\text{Fig. 3. Message forwarding after mailbox leaves} 
\]

**Message Routing.** Figure 3 illustrates the message forwarding process. Suppose
agent \( M_s \) wants to send a message to agent \( M_r \). Referring to the function
SendMessage_Agents(), \( M_s \) first checks whether the address of \( M_r \)'s mailbox
has been cached locally. If so, it sends the message to the cached address. Otherwise
it sends the message to \( M_r \)'s home host (Step (1) in Figure 3).

\[
\text{SendMessage_Agents(Message msg) } \\
\quad \text{// executed by mobile agent} \\
\quad \text{if (the receiver's address is in cache)} \\
\quad \quad \text{sendMsgToMAP("AGENTMSG", address in cache, msg);} \\
\quad \text{else} \\
\quad \quad \text{String homeAddress = msg.getReceiver().getHome();} \\
\quad \quad \text{sendMsgToMAP("AGENTMSG", homeAddress, msg);} \\
\]

When a host receives a message destined to an agent \( M \), it checks whether \( M \)'s
mailbox is currently resided locally. If so, it inserts the message to \( M \)'s mailbox
directly. Otherwise the message is forwarded to the M’s current address recorded in the local address table. See the function `MessageRouting_MAP()` for details.

```c
MessageRouting_MAP(agentMsg) {
    AgentID receiver = the target agent of agentMsg;
    if (the receiver’s mailbox is local){
        insert agentMsg to the mailbox;
    }else{
        AddressEntry entry =
            addressTable.getAddress(receiver);
        if (entry.VALID) {
            sendMsgToMAP("AGENTMSG", entry.address,
                agentMsg);
              //Step (2) in Figure 3
            sendMsgToMAP("UPDATE",agentMsg.getSender(),
                entry.address); // Step (2)’ in figure3
        }else{ //valid tag is false: receiver is migrating
            entry.blockQueue.insert(agentMsg);
              //insert the message to the block queue;
        }
    }
}
```

Agent Mₖ caches the new address of agent Mₜ contained in the incoming “UPDATE” message. Next time when Mₖ sends messages to Mₜ, it will send the message to this new address.

### 3.3 Properties of the Algorithm

Before proving the correctness of our algorithm, we give a formal definition of the path of an agent mailbox.

**Definition. Path(mb)** is a sequence <hₙ hₜ ... hᵢ,..., h₁> of hosts where the mailbox mb has been inhabited. For all hᵢ, hⱼ in the path, hᵢ is visited by mb earlier than hⱼ if 0 ≤ i < j ≤ n. The host hₜ is the home of mb’s owner agent, and h₁ is the host where mb is currently located.

Theorem1 shows that our algorithm can provide location transparency from the point of view of a sender agent.

**Theorem1.** With the proposed algorithm, a sender agent can send its messages without knowing where the target agent is located.

**Proof.** According to the function “SendMessage_Agent()”, when the sender agent wants to send a message to another, it will check if it has cached the receiver’s address. If there is the receiver’s address in its cache, it will send the message to this address without caring about whether it is outdated. Otherwise it will get the receiver’s home address from its ID and send the message to its home. In both cases, the sender need not specify the current location of the receiver when it wants to send a message. QED
The following lemmas and theorem 2 show the effectiveness of our algorithm, i.e., it can guarantee the delivery of messages. Besides, the message will be forwarded at most once so that it will not chase its recipient.

**Lemma 1.** Suppose a mailbox $mb$ is currently located at host $h$, and Path($mb$) is $< h, h, \ldots, h, h >$. For all $h$ in Path($mb$), if $h$ was not down, $mb$ must have received the REPLY message from $h$ before it leaves $h$.

**Proof.** If $mb$ wants to leave $h$, it must send Deregister messages to all the hosts in Path($mb$). Since $h$ is not down and it is assumed that the underlying location dependent communication mechanisms can shield the network failure, $h$ will at last receive the Deregister message and send a REPLY message to $mb$. According to the function ProcessMVBMsg($MB$), $mb$ cannot leave $h$ until it collects all the REPLY messages from all the hosts in Path($mb$) or when time out. Since we need not worry about network failure, we can conclude that the REPLY message from $h$, will arrive at $mb$ before $mb$ leaves. QED

**Lemma 2.** For all $h$ in Path($mb$), the valid tag of $mb$’s address in the address table is true only if the address reflects exactly the current location of $mb$, i.e., the address kept in the address table is not outdated.

**Proof.** Suppose the address of $mb$ kept in $h$’s address table is $h$. If the valid tag is true, from the function MessageProcessing_MAP() we can conclude that $h$ must have received $mb$’s REGISTER message from $h$, and the Deregister message has not arrived yet. So the REPLY message has not been sent out from $h$. From lemma 1 we know that $mb$ is still at $h$ and cannot leave until it has collected all the REPLY messages from hosts in Path($mb$), including $h$. So the address $h$ kept in the address table reflects the current location of $mb$. QED

**Theorem 2.** All the messages can be delivered to their recipients’ mailboxes by being forwarded at most once.

**Proof.** Suppose a sender agent $S$ is sending a message $m$ to a receiver agent $R$, and $R$’s mailbox $MB_r$ is located at host $h$. Let Path($MB_r$) be $< h, h, \ldots, h, h >$. Without loss of generality, we assume that $R$’s address kept in the cache of $S$ is $h$ and $0 \leq i \leq j$ (if there is no record of $R$’s address in the cache of $S$, the message will be sent to $R$’s home, which is $h$ in Path($MB_r$)).

$S$ will obtain $R$’s address, namely $h$, from its address cache and send the message $m$ directly to $h$. When $m$ arrives at $h$, 3 cases could happen:

**Case 1:** $i = j$. The message $m$ will be directly inserted into $MB_r$ without being forwarded. No matter where $R$ resides, it can get $m$ from its mailbox later.

**Case 2:** $i < j$ and $h$ has not received $MB_r$’s Deregister message from $h$. In this case, $m$ will be processed before $R$’s Deregister message. To deliver $m$ to $R$, $h$ will check its address table and find $R$’s address is $h$ and the valid tag is “true”. So $m$ is forwarded to $h$. Since $m$ is processed earlier than $R$’s Deregister message, $m$ is forwarded to $h$ earlier than the REPLY to the Deregister message. The FIFO property can guarantee that $m$ arrives at $h$ earlier than the REPLY message. From Lemma 1 we can conclude that $MB_r$ cannot migrate to other hosts during the transmission of $m$. After $m$ arrives at $h$, it will be inserted into $MB_r$. So $R$ can receive $m$ later from $MB_r$ and $m$ is forwarded only once.

**Case 3:** $i < j$ and $h$ has received $MB_r$’s Deregister message from $h$, i.e., $MB_r$ has left for $h$, The host $h$ checks the valid tag of $MB_r$’s address. If it is “true”, the address of $MB_r$ kept in the address table must be $h$, (warranted by Lemma 2) and $m$ is
forwarded to MB
in the same way discussed in the second case. If the valid tag is "false", we can conclude that MB
is on its way to h
. The message m will be put into the blocked message queue. It won’t be forwarded until MB
reaches h
, and its REGISTER message arrives at h. After the REGISTER message arrives, m will be forwarded to h
. As discussed in the second case, MB
will not leave during the transmission of m, since m will arrive at h
, earlier than the REPLY message. Therefore, in this case, m is also forwarded only once and R can get m later from MB
.

From the above discussion of all the three cases, we can conclude that all the messages can be delivered to their recipients’ mailboxes by being forwarded at most once. \textbf{QED}

4. Performance Analysis

In this section we formulate the traffic cost of the location updating and message delivery of the proposed algorithm in terms of the number of messages required. To simplify the problem, we ignore the differences in the distances between hosts. Here we introduce 3 decision variables: \( x \), \( x_a \), and \( x_i \), which are defined as follows:

\[
\begin{align*}
x &= \begin{cases} 
1 & \text{The agent has left and messages should be forwarded to its new location.} \\
0 & \text{Otherwise} 
\end{cases} \\
x_a &= \begin{cases} 
1 & \text{The agent will move with its mailbox.} \\
0 & \text{Otherwise} 
\end{cases} \\
x_i &= \begin{cases} 
1 & \text{The agent and its mailbox reside at different hosts.} \\
0 & \text{Otherwise} 
\end{cases}
\end{align*}
\]

The location updating cost and message delivery cost can be formulated as follows:

\[
C_{update} = x_0 \left( x_1 C_{ctrl} + (3 N_{mb} + 1) C_{ctrl} \right) \quad (1)
\]

\[
C_{delivery} = C_{msg} + x \left( C_{msg} + C_{ctrl} \right) + x_1 \left( C_{msg} + C_{ctrl} \right) \quad (2)
\]

where \( C_{ctrl} \) and \( C_{msg} \) denote communication traffic of a control message and an agent message, respectively. Since control messages, such as "\texttt{MVMB}”, “REGISTER” and “UPDATE” messages may be much shorter in length than agent messages, they should not be counted in the same way. \( N_{mb} \) denotes the number of hosts in Path(\( mb \)). As discussed in section 3, when an agent is leaving and decides to take its mailbox along with it to the new host (\( x_a \) is 1), it sends “\texttt{MVMB}” message to its mailbox if the mailbox does not reside at the same host (\( x_i \) is 1). The cost is denoted by the first term in parentheses of Formulation (1). Then it sends “\texttt{DEREGISTER}” messages to the \( N_{ab} \) hosts in Path(\( mb \)), collects \( N_{ab} \) “\texttt{REPLY}” messages and sends \( N_{ab}+1 \) “\texttt{REGISTER}” messages on arriving at the next destination (the second term in the parentheses of Formulation (1)).
When an agent sends a message to another agent, it sends the message to the receiver’s location cached in its address table (first term in Formulation (2)). If the sender’s knowledge about the receiver’s location is out of date, the message has to be forwarded to the new location and the “UPDATE” message is returned to the sender. The cost is denoted by the second term of Formulation (2). If the receiver wants a message from its mailbox and it resides at a different host with its mailbox, it sends a control message to its mailbox and the mailbox returns the corresponding message to it (the third term of Formulation (2)).

From these two formulae, we can see that if an agent migrates without taking its mailbox ($x_i$ is 0), the location updating cost is 0. By deciding the value of $x_i$ to adjust location updating cost, our algorithm works in an adaptive way. There are two extreme cases.

1. The first one is that the mailbox never moves during the life cycle of its owner agent. In this condition, the mailbox always resides at the home of its owner. Messages are sent to the receiver’s home and the receiver gets the messages from its home. It is similar to the home registration and forwarding method. In this condition, the location updating cost is 0. But the message delivery cost is expensive ($2C_{msg} + C_{mov}$) and the home must be kept linked during agents’ life cycle.

2. The other extreme case is that the mailbox is bound to its owner and they are always migrating together. Under this condition $x_j$ is always 0 and the message delivery is less expensive, but the location updating cost $C_{update}$ as shown in Formulation (1), is $(3N_{tot} + 1)C_{mov}$ since $x_i$ is always 1.

To save the total traffic cost, which includes the location updating and message delivery cost, compromise must be made between the two extremes according to specific applications. To determine whether moving with its mailbox or not, an agent can consider factors such as the number of messages it will receive in the next host and the distance between its next destination host and the current location of its mailbox. If an agent seldom receives messages from others in the next host, it doesn’t need to take its mailbox to the new host. On the other hand, if an agent will receive messages frequently from others and the next host is far away from the host its mailbox currently resides at, it will be expensive to leave the mailbox unmoved and to fetch messages from the remote host. In this case the agent should migrate to the new host together with its mailbox.

5. Simulations and Observations

To evaluate the performance of the algorithm as formulated in Section 4 under various conditions, our algorithm is implemented in a simulated mobile agent environment. In our simulations we assume that the traffic cost for every agent message ($C_{msg}$) is 1 unit and the control message cost ($C_{mov}$) is one fourth that of $C_{msg}$. The cost is recorded automatically each time a message is sent out. We also assume that whenever an agent migrates, it will hop to a host different to all the hosts it has ever visited.

The first scenario of our simulation involves one agent only. It migrates from one host to another without communication. The cost of the register, deregister and reply messages is recorded. We use the term “migration ratio” to denote the ratio of the mailbox migration number to the agents migration number. Figure 4 shows the
average traffic costs per agent migration under different migration ratio and total hops numbers. We can see that as the migration ratio increases, the average traffic cost increases quickly. Since \( N_{\text{stop}} \) increases as the migration ratio rising, this result can be predicted from Formula (1) in Section 4.

As we have discussed in Section 4, message delivery is expensive if the mailbox stays at the agent’s home host. Then what’s the relation between the cost of the message delivery and the mailbox migration ratio? Result shown in Figure 5 can answer this question. This time a sender keeps sending messages and the interval of two messages is randomly set. The total number of messages is 600. The receiver receives several messages on every host and migrates to another. The migration intervals are set to 30, 10 and 6 messages respectively and the corresponding traffic costs under each interval and each migration ratio are recorded. As we can see, the average delivery cost per agent message is the highest when the migration ratio is 0, i.e., the mailbox stays at its owner’s home all the time during the agent’s life cycle. The cost decreases as the migration ratio increases. It reaches the lowest point when the migration ratio is 1. The mailbox is bound with its owner under this condition and the agent can get the message directly from its mailbox. We can also observe that under the same migration ratio, the average delivery cost is a little higher when the move interval becomes shorter. The result is reasonable because the more frequently the mailbox migrates, the more messages must be forwarded.

![Fig. 4. Effect of Migration Ratio on the Updating Cost](image1)

![Fig. 5. Effect of Migration Ratio on the Message Delivery Cost](image2)
From figure 4 and 5 we can observe that the average location updating cost and message delivery cost varies in opposite directions as the migration ratio rises. There must be an optimal point on which the total traffic cost is the lowest. We introduce a sender agent and a receiver agent in our third simulation scenario. As in the second one, the sender keeps sending messages at random intervals. The moving intervals of the receiver are also randomly set. They vary from 0 to 19 messages (inclusive). Whether the receiver migrates with its mailbox or not is determined by the moving interval and a pre-set threshold value. Before moving, the receiver estimates the number of messages it will receive in the next host (the number is generated by a random number generator in our simulation). If the number is less than the threshold value, the receiver will migrate without its mailbox. Otherwise the mailbox will be taken along. The total number of messages are set to 100, 300 and 500 respectively. The average total cost is shown in figure 6. We can see that the costs are higher in two extreme conditions. Since the moving intervals distribute evenly between 0 and 19, it reaches the lowest point when the threshold value is almost half of the highest interval, i.e. 8 or 12 messages. From this example we can conclude that by determining properly whether the agent will take its mailbox along, the communication overhead can be decreased considerably.

![Figure 6](image.png)

**Fig. 6.** There is an optimal point if the threshold value is properly set.

6. Conclusions and Future Work

We have proposed a mailbox-based approach to designing mobile agent communication protocols. In our design, a mobile agent and its mailbox can be separated in the sense that they can reside in different hosts. An agent can migrate to a new host while leaving its mailbox in a previously located host. This helps overcome the high location updating cost. An agent can decide whether to take with its mailbox along with it according to the number of messages in the network and its movement area. One of the two extreme cases of our algorithm is similar to the home forwarding scheme. If the decision is properly made, as shown by our simulation results, the lowest total traffic cost which is less than that of both extremes can be obtained.

A mailbox-based protocol still follows the registration-and-forwarding scheme but can be made to overcome many of the drawbacks. It can route the messages in a
reliable and location transparent way. By forwarding the message at most once, the protocol avoids the problem that messages may chase forever its target agent that migrates frequently. Unlike the home registration method used in mobile computing, e.g., Mobile IP, the mailbox-based protocol decentralize the role of the home host and reduce the reliance on it, so that mobile agent’s capability of supporting disconnected operations can be realized in real. Furthermore, the protocols are adaptive and can decrease the overhead of location registration by deciding whether a mobile agent will migrate with its mailbox.

Although in our algorithm the dependence and workload of the agent home have been distributed to all the hosts on the agent migration path, the agent home still has to work as a location server, especially when it’s the first time that an agent sends a message to another born on it. To let the algorithm work even when some hosts on the agent migration path including the agent home are down or disconnected, one dedicated location server can be introduced in our framework. Since it is queried only when disconnection or system failure occurs, the dedicated location server will not be the performance bottleneck as a central one. Security issues should also be considered in our future work. Because the sender agent may accept “UPDATE” message from any host on the receiver migration path as discussed in Section 3, it is vulnerable to the address spoofing attack. Specifically a Bad Guy could simply send a bogus “UPDATE” message to the sender and cause all the messages to be sent to the Bad Buy instead of the receiver. To prevent such attacks, authentication schemes must be adopted.

References


Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent Environment

Klaus Haller and Heiko Schuldt
Swiss Federal Institute of Technology (ETH)
Institute of Information Systems
ETH Zentrum
CH-8092 Zürich, Switzerland
{haller|schuldt}@inf.ethz.ch,
http://www-dbs.inf.ethz.ch

Abstract. Mobile agent systems are a powerful approach to develop distributed applications since they migrate to hosts on which they have the resources to execute individual tasks. Existing mobile agent systems require detailed knowledge about these hosts at the time of coding. This assumption is not acceptable in a dynamic environment like a peer-to-peer network, where hosts and, as a consequence, also agents become repeatedly connected and disconnected. To this end, we propose a predicate-based approach allowing the specification of hosts an agent has to migrate to. With this highly flexible approach, termed \textit{P2P Mobile Agents}, we combine the benefits of execution location transparency with those of code mobility. Similarly, also the recipients of messages can be specified by predicates, e.g. for synchronisation purposes. For providing meta information about agents and hosts we use XML documents.

1 Introduction

Mobile agents are a programming paradigm for distributed systems. In particular, this approach tries to reduce the communication costs and to evade the problem of network latency. Mobile agents consisting of code and an execution state are transferred from host to host to achieve these goals. They move in order to process the data available on hosts they reside on instead of sending the data to the host which is processing them. Hence, mobile agents are individual software entities performing tasks autonomously while hopping from host to host, which know on which host they find the data they need. With this point of view in mind, mobile agents require only simple mechanisms for messaging among themselves or choosing a host at run-time – since the latter already has to be defined when a mobile agent is coded.

On the other side, mobile agents are proposed for executing workflows since their early days [5]. Semantically corresponding steps are coded within a single mobile agent, so each agent can be considered as an independent transaction.
Table 1.

<table>
<thead>
<tr>
<th>Execution Location</th>
<th>Code and Data Mobility</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>&quot;simple&quot; program</td>
</tr>
<tr>
<td>yes</td>
<td>RPC</td>
</tr>
</tbody>
</table>

However, when processing takes place on shared resources or data, also synchronisation of different, originally independent mobile agents is required. Under this perspective, sophisticated methods for specifying targets of messages, especially for synchronizing agents accessing shared resources, and for migration are essential.

Existing mobile agent systems usually deal with the problem of implementing basic technologies, such as strong migration, which are however still subject to research. Instead, we focus on extending an existing mobile agent framework with sophisticated methods for specifying message targets and hosts an agent should migrate to. To be generic, we use predicates for specifying agents and hosts in a declarative way.

Also, agents are commonly seen as autonomous entities being able to cooperate in a bilateral way. Thus, there is no particular need for centralized services. Because of this, a peer to peer approach for agent cooperation is obvious.

Consider the following example, which we will use throughout the paper: An agent interested in buying stocks wants to figure out a place with a stock exchange agent in order to watch the stock prices and eventually to buy stocks. Moreover, it is interested to move to the place with the lowest load.

To point out in which way this adds new ideas to the mobile agents world, we want to compare our approach, called \textit{P2PMobileAgents}, with other mobile agents and with remote procedure calls as illustrated in Table 1. We can use two criteria for classification: Code and data mobility on one side and execution location transparency, meaning that the programmer does not have to know on which host (sub)tasks are going to be executed in future, on the other side.

Mobile agents belong to the class that represents code and data mobility, but no execution location transparency is provided, since a programmer has to specify explicitly where he wants the agent to migrate to. Conversely, remote procedure calls (RPCs) hide the fact that a code fragment is executed on a foreign host. But in contrast to mobile agents, code is not passed through the network, only parameters. So execution location transparency is supported, but neither code nor data mobility.

Our \textit{P2PMobileAgents} as an extension to the mobile agent approach certainly support code and data mobility, but they additionally offer execution location transparency. Places, to which an agent has to migrate to continue its execution, are specified using predicates. The \textit{P2PMobileAgents} framework evaluates such predicates at run time, and so combines the advantages of mobile agents and RPCs. The remainder of this paper is structured as follows: We present an overview of the messaging and migration mechanisms of exist-
ing mobile agent systems and the possibilities for specifying message recipients. Therefore, we introduce our system architecture in Section 2. In Section 3, we discuss the query language which is used for describing the features of entities (places and agents), before we explain how queries are evaluated (Section 4). We conclude this paper with a short summary in Section 6, after discussing related work in Section 5.

2 System Architecture

Within the P2P Mobile Agent project of the Database Research Group at ETH, we are implementing a mobile agent platform providing the possibility to choose and specify agents and hosts in a peer-to-peer environment by using predicates. In what follows, we present our system architecture.

Our system consists of places on which static and dynamic agents are executed. Resources available on places are encapsulated and can only be used via static agents bound to particular places. A peer-to-peer network supports communication between different places. Additionally, meta information for all agents and places is provided.

An example is shown in Figure 1: There are places connected by a peer-to-peer network and also both static and a mobile agents running within these places. The static agent existing at the place identified by "atp://infl3.ethz.ch:4435" represents a stock exchange and supports the buying and selling of stocks. In addition, there is a mobile agent at the place "atp://infl7.ethz.ch:4435", a broker offering the service "buy stocks".

We assume that the broker agent’s task is to buy stocks of the "ACompany". This requires him to monitor the trend of this stock and to be able to react immediately. Especially the last requirement demands that the agent is executed on the same place as the stock exchange agent.

For this reason, the broker agent has to migrate to a place with such a static stock exchange. Therefore, the agent has to find out via the peer-to-peer network which place is appropriate for its particular demands. The information about places and agents, called meta information in the remainder of this paper, has to be provided by each place and agent, respectively.

The task of managing meta data is realized by an additional type of agent, one per place, called AgentsManagementAgent, AMA (see Figure 1). The concept of an AMA is essential for the implementation of our new features: Together, all AMAs form a peer-to-peer network. Additionally, they are responsible for the meta data management. It is subject to the next two subsections to explain how these AMAs realize this task, before certain system issues are discussed in the last subsection.

Already at this place, we want to point out that no programmer using the P2P Mobile Agent framework notices the AMAs: They are started with the places. Also, communication with these AMA is transparent for mobile agent programmers, since the latter use an interface providing the extended capabilities of our system.
2.1 Meta Data Management and Querying

The P2PMobileAgent system allows to specify destinations for migration or messages by predicates. Therefore, meta data describing agents and places is needed. This subsection describes how this meta data is managed, focusing on the aspect of both structure and storage.

**Structuring the Meta Data.** A key factor for allowing sophisticated queries to select places and agents is to structure the meta data needed for that purpose appropriately. To find the most suitable solution, the characteristics of this kind of information have first to be classified.

Typically, meta data comprises the description of services an agent provides or the actual load of a place. For this kind of information, a global schema can be defined. However, if groups of agents cooperate to solve a problem, it is sometimes important to be able to make information about their internal state available to the public. Yet, no universal schema can be found for this kind of information, since this particular state information differs from agent to agent not only with respect to the individual instances but also with respect to the granularity.

Nevertheless, to be able to process queries efficiently, we cannot rely on unstructured data. Hence, storing the meta data in a semistuctured way is sensible. In particular, we use a subset of XML (eXtended Markup Language) [13] for this purpose, which doesn’t contain the concept of parameters.
We assume that agents have to have a shared knowledge about the structure of the internal information (and in particular using a common vocabulary) available from other agents they wish to interact and cooperate with. Alternatively, ontologies would be needed, but this is definitely not the objective of our work.

Although the amount of information encapsulated may vary from agent to agent and from place to place, there is a minimal set of information mandatory for agents (i.e., their type, that is whether or not they are mobile) and for places (e.g., their address), respectively.

Figure 1 also shows such XML documents storing semistructured data. They consist of entities enclosed in tags which also can be used in a nested way. We want to illustrate this concept by looking at the static agent representing a stock market. The entity <agent/stocks> with its subentities <stock> names the stocks which are subject to trade. Knowing this structure and the tags exploited, every other agent can access and use this information.

Meta Data Storage and Processing. Information about the place itself is managed by the AMA while information about agents are provided by the agent. Hence, this illustrates that there are two possibilities for query processing: it could be done by the agents or the AMAs.

If the individual agents evaluate the queries, agents become more complex—and a lot of code is redundant for all agents. Currently being present at some place, they would need information about all other agents of the same place. Additionally, the load of the agents would rise, because they cannot simply concentrate on their normal task. Concentrating the query evaluation on the AMAs allows us to benefit from the fact that they can aggregate information from all local agents making query evaluation faster. This applies especially if a query is not restricted to the information about either a single place or a single agent but also includes information on several agents. “Give me the place where an agent of the type ‘stock exchange’ is executing?” is a good example for this type of query.

To evaluate queries, the required information has to be made available to the AMAs. Therefore, we have to distinguish two different kinds of information: static and dynamic.

A piece of information of an agent or place is called static if it cannot change during its whole life cycle, e.g. the services an agent provides or its type (see for instance the entities <agent/services/service> or <agent/type> in Figure 1). This kind of information can be cached by the AMA to improve the efficiency of the system.

On the other side, there is dynamic information which cannot be cached by the AMA. Instead, every time an agent has to be asked by the AMA to deliver the current value when the latter needs this information for query evaluation purposes. An example of this dynamic meta data is the number of brokers connected to a stock exchange broker (see <agent/status/brokers> in Figure 1).

Dynamic information is always part of the element <status>, so that the system can determine whether data is dynamic or static. If there are different
kinds of agents, some may store the same information as static whereas others store them as dynamic information. Because the same kind of information should always be addressable by the same tag name, the tag level "$status$" is masked for queries. So there is no need to know (for queries) whether the data is static or dynamic.

2.2 Peer-to-Peer Network

The goal of mobile agents issuing queries over places and other agents is to support their migration within the network. To this end, we want so support an open environment, since we are not interested in network topologies with any kind of central management. So, a peer to peer network fits to our requirements. All nodes are equal in this topology and have to provide the same functionality. Usually, not every node is aware of all the other nodes, but sees only a fraction of the whole system. Hence, communication between two nodes might involve several intermediate nodes.

This network is used for querying on agents and places and not for direct communication. This service is provided solely by the AMAs, so there is no reason that every "normal", i.e., mobile agent participates. Thus, the network is spawned by the universe of all AMAs.

Therefore, our system has two layers as shown in Figure 2: There is a client/server-like communication between agents of one place and their AMA and a peer-to-peer network between the AMAs of the different places.

Communication between an agent and its AMA is set up when the agent is started on the place, independently whether it is a new agent or whether the agent has just migrated to this place. The reason is that the AMA caches static information about the agent as it is described in Section 4. After the termination of an agent or after its migration to another place, the connection to the AMA is closed.
2.3 Implementation Details

Our system of \textit{P2P Mobile Agents} is based on Aglets [9], the mobile agent framework developed by IBM which evolved to an open source project. Using this framework makes it possible to concentrate on implementing new concepts for specifying communication and migration targets instead of dealing with basic problems of mobile agents.

The Aglets framework provides mechanisms for weak migration and for sending messages. Each Aglet has a unique ID and by knowing this ID of an agent, communication mechanisms of the framework can be used for sending messages directly to it. In addition, the Aglet system allows to receive events when something important happens, e.g., when a new agent is launched or arrived at a certain place. These places are usually Tahiti servers, a program also being part of the Aglet system.

In this subsection, we concentrate on how the functionality for sending messages and migration based on the specification of targets using predicates can be added to the existing framework by using its basic features. The most important question is how to integrate the functionality into the existing Aglet framework. For this reason, it is either possible to extend and modify existing classes or to build a new component on top of the existing one.

In order to facilitate maintenance and since it is more convenient for programming, we have chosen the latter possibility with the restriction that the extended functionality for agents is realized by adding two new methods to the Aglet class. These methods implement mechanisms to send messages, and to migrate to targets specified by predicates.

In short, we realize the \textit{P2P Mobile Agent} system by adding a new layer to the Aglet system. This layer provides support for querying on agents and places.

Figure 3 illustrates this for the case of an agent query. In the Aglet system, an agent inherits from the class Aglet. In the \textit{P2P Mobile Agent} system, a new subclass is introduced, which provides the functionality for sending messages to agents and for migrating to places both specified by predicates. All mechanisms of the Aglet framework are not hidden and still available. In Figure 4, we illustrate the layered architecture for places. The context class of the Aglets
system is the basis, on top of which the Tahiti server is running. Similar to the agent case, there is still the possibility to address these tiers by agents. In the $P2PMobileAgent$ approach, we added a new tier, the $AgentsManagementAgent$ level.

AMAs are also Aglet agents that use the services provided by the lower level, especially the event handling and communication primitives and that integrate them with the new XML meta data to provide the ability to query about agents and places. This functionality is used by the new layer of the agents that implement the new features for sending and migration.

In a peer-to-network, the Aglet system provides the possibility to send messages to agents if their IDs and addresses are known, but linking the places together to share knowledge about places or agents is not provided by the Aglet system framework.

So this peer-to-peer network formed by the AMAs is used for communication during the processing and evaluation of queries on hosts and agents, but not for communication between agents. If a message is to be sent to a particular agent fulfilling the query conditions, the agent is determined and its proxy is returned to the agent having initiated the query. Having a proxy, it is possible to send messages directly to the agent, although it might migrate in the future without indication via the AMA peer-to-peer network.

3 Query Language

In this section, we address how to access the meta data. Therefore, we define a query language. The task of this query language is to find agents or places with particular properties. The result set of such queries consists of places an agent should migrate to or of a set of agents which are the recipients of a message.

In the context of querying XML documents, various query languages have been proposed, [8] is a compilation and comparison of the most important ones. Rather than relying on a fully-fledged XML query language, we follow a slightly simplified own approach. The reason is, that existing query languages provide a
set of sophisticated features which considerably exceed the requirements of our 

P2P Mobile Agent approach.

The features of the existing query languages are:

1. Queries consists of a pattern clause, a filter clause and a constructor clause, 
   including the possibility for information passing between different clauses. 
   Also, nesting, grouping, indexing, and sorting is supported.
2. A join operator.
3. Tag variables and path expressions.
4. Handling of alternatives.
5. External functions like aggregation, etc.

In contrast, these are the requirements of the P2P Mobile Agent system:

1. Specifying which documents of which places are of interest.
2. How to deal with the problems of a peer-to-peer environment.
3. Only flat result sets and no need to build up complex types.
4. Simple expressions like predicates describing, e.g., agent tag values.

These requirements reflect that in our case, the information itself is stored in 

a distributed way in a peer-to-peer network. The existing query languages are 

well suited for complex queries to be evaluated on top of a single database. In 

our case, the queries itself are less sophisticated. Instead, we need to deal with 

a complex environment. 

So, of course, a full fledged query language could also be used – after extension 

– for this purpose but with the drawback of carrying on an considerable amount 

of unneeded overhead.

For these reasons, we have constructed a query language tailored to the 

P2P Mobile Agent requirements. This query language supports two levels in 

the definition of a query. These levels are discussed in the two following subsections: 

The meta level deals with the environment, describing what documents are of 

interest under which circumstances. In the second subsection, we present how 

to specify the criteria driving the selection of entities that form the result set 

and by this, representing the target agents or places for messages or migration 

purposes. Then, we present an example of such a query.

3.1 Formulating a Query: The Meta Level

Prior to the evaluation of a query against an XML document, the appropriate 

place or agent, respectively, managing this document, has to be found under 

certain constraints. These aspects are subject to the meta level part of a query.

The four following criteria describe the meta level of a query and are mandatory 

for each query:

1. result type: The result set of a query is either formed by hosts or by agents, 
   depending in which context the query is used; recipients of a message have 
   to be agents, whereas places are expected in case of migration.

RESULT_TYPE ::= AGENT | PLACE
2. **cardinality of result set**: Depending on the kind of problem, an agent issuing a query might only be interested in receiving at most one element in the result set, e.g., if it is booking a flight, it wants to send the message to only one travel agency agent. Instead, if the agent wants to receive offers before booking a flight, it is interested in contacting all travel agencies. So we also have to consider the possibility to retrieve all entities complying with the specified condition. To be able to restrict the costs for executing a query, an agent can even restrict the number of entities in the result set to some individual maximum value.

   \[
   \text{CARDINALITY ::= ONE | ALL | MAX(int)}
   \]

3. **search space**: The concept of mobile agents is based on the assumption that it is cheaper under certain conditions, to transfer code instead of data. So it is useful to be able to restrict communication to the local place, although it might be required to search the whole peer-to-peer network in other cases. Considering peer-to-peer networks with the dimension of the Internet, it is reasonable to allow a more differentiated granularity then "local" or "global". For this purpose, we also introduce the possibility to restrict the number of tiers the message should be forwarded to. The number of tiers thereby represents the number of intermediate AMAs having forwarded a request.

   \[
   \text{SEARCHSPACE ::= LOCAL | GLOBAL | TIERS(int)}
   \]

4. **time-out interval**: In a large and dynamic network, it is not reasonable to introduce a protocol that allows a book-keeping mechanism to decide whether or not all places have evaluated a query. Instead, it is better to define a time-out interval that restricts how long an agent having initiated a query accepts answers that shall be added to the result set before it proceeds its execution. The parameter in this case is an integer value that specifies how many milliseconds the agent waits for new results at most. Depending on the values of the "cardinality" and "order criteria" parameters, it is even possible that the agent is able to proceed its program execution earlier.

   \[
   \text{TIMEOUT ::= MS(int)}
   \]

### 3.2 Formulating a Query: Specifying the Selection Process

Additional criteria for carrying out a query are conditions and order criteria. The latter ones describe which entities shall be chosen if too many entities fulfill the conditions.

1. **conditions**: A condition consists of one or more predicates. Such predicates can be combined by using boolean operators. A predicate specifies expected element values for the agent (only if the result is an agent) or for places, thereby making it possible to specify requirements of places or to restrict the set of agents that are evaluated to the agents on the same place. Additionally, it can be required that only entities (places resp. agents) are of interested, if other agents reside on the same place.

   \[
   \text{CONDITION ::= PREDICATE \{(PREDICATE LOG\_OP PREDICATE)\}}
   \]

   \[
   \text{NOT \{PREDICATE\}}
   \]
PREDICATE ::= PLACE(EXPRESSION) | AGENT(EXPRESSION)* | EXISTS_ALSO_AGENT(EXPRESSION)
EXPRESSION ::= EXPRESSION | NOT (EXPRESSION) | (EXPRESSION LOGOP EXPRESSION)
LOGOP ::= AND | OR
EXPRESSION ::= VALUE COMPAOP VALUE
VALUE ::= CONSTANT | TAG
COMPAOP ::= > | < | =
*only if RESULT_TYPE is AGENT

2. order criteria: If the cardinality of the result set is not "all", it is possible
that more entities fulfill the requirements specified in the field "condition".
To this end, we introduce order criteria, like the ORDER-BY-clause in SQL
queries [6]. This allows us to find the best fitting entities. Tags are used as
sorting criteria for the result set. If the result set consists of places, only tags
of the place are allowed, if the result set consists of agents, tags of the agent
itself and also tags of the place can be used. In the latter case, these tags
are dressed by "place."", e.g. "place.os".
ORDER,CRITERIA ::= Tag ( ASC | DESC ) [, Tag ....]

3.3 Example of Place Query

In this subsection, we continue the discussion of the example started in Section 2: there is a mobile agent representing a broker, which should migrate to a place
where a static agent exists that implements a stock exchange. In Figure 5, a
query is presented that formulates this requirement in a query.

Because the agent wants to migrate, it is interested in a host and thus specifies
"PLACE" in the field "RESULT_TYPE" (1). It is only desired to move to one
place and not to duplicate itself to migrate to different places, so the value of
the field "CARDINALITY" is set to "ONE" (2).

There is no restriction on how many places should be involved in this query, so
the whole peer-to-peer network is specified as the search space (3). Nevertheless,
the agent wants to consider only places that are found within the 15 ms timeout
interval (4).

Then the requirements for the acceptable places have to be specified: our
mobile broker agent needs a particular agent on the destination place, so it
uses a predicate of the kind "EXISTS_ALSO_AGENT" (5). The tag "TYPE" is
required to be "StockExchange" (6). Additionally, this agent has to be static (7)
so that it is guaranteed that it is still there after the mobile broker agent has
migrated to the new place. Moreover, the stock exchange has to be in operation
(8).

It is possible that different places fulfill these conditions. If the agent issuing
the query does not specify any order criteria, the first place fulfilling these con-
ditions would be chosen as migration target. But in case it aims at migrating to
the host with the lowest load, this has to be added as an order criterion (9).
Fig. 5. Sample Query

4 Query Execution

This section concentrates on the evaluation of queries. Especially, it deals with the interaction between agents and their AMA.

First, we discuss in detail how searching for agents is performed. Since searching for places does not significantly differ, we only give a brief summary of these differences. In both cases, the query execution is embedded in the execution of a migration resp. messaging procedure invocation, which takes place synchronously. We illustrate these algorithms by continuing the example started in the last section.

4.1 Searching for Agents

Searching for agents takes place in the context of sending messages. There is no specification of a particular target agent but rather a description of the kind of agent the message should be delivered to. By using the presented query language (see Section 3), the matching agents have to be found. Therefore, the peer-to-peer agent system starts to search for such agents in the network.

First, the query is submitted to the local AMA. There, it is distributed to all AMAs on other places the local AMA knows (i.e., has direct links to). These AMAs also forward the query to all AMAs they know, until the upper bound of tiers is reached – if one is specified in the field "SEARCHSPACE".

Aside of forwarding the query, each AMA also evaluates which of the agents at its place fulfil the specified requirements. To this end, it caches all static information it gets during the communication with newly launched agents, either static or mobile agents. But if dynamic information is required for evaluating a query, meaning that information appears within the tag <status> in the describing XML document, the local agent has to be asked by the AMA to deliver this information.

If an agent matches the requirements of the query, the information to contact it are delivered directly to the agent that initiated the query.

After the time-out interval is exceeded, the initiating agent does not expect to be informed about any further agents matching the criteria of its query meaning that they are ignored. It finally evaluates the query and sends the message to the selected agent(s).
A global unique ID is attached to each query, such that each AMA can store
the last queries it evaluated. By this, communication and execution costs are
reduced since it can be avoided that a query is executed and forwarded several
times at/by the same place, because a place can be mostly reached via different
paths in the network. But by means of the ID of a query which is evaluated
against the IDs of executed queries, this phenomenon can be avoided.

The approach followed here has a side-effect: queries can only be evaluated
on a snapshot of the overall system. Hence, agents currently migrating while a
query is launched are not found, because at this moment they are not registered
at any AMA implying that the snapshots need not necessarily be consistent. The
AMA of the place they are leaving does not know them and even if it would
know them, it would not be able to communicate. The new AMA the agent is
migrating to does not know them because they have not started to communicate
with each other.

4.2 Searching for Places
Searching for places is initiated whenever the host or the hosts an agent should
migrate to are specified by a query. As a consequence, the starting point is the
XML document describing the place, even though also references to agents are
possible.

4.3 Example
In this section, we discuss the evaluation of our sample query presented in Sec-
tion 3.3. Therefore, we first have to present a network configuration: As shown
in Figure 6, there are four places with agents. On place 1, there is the agent
initiating the query discussed above. On place 2, there is an agent running that
encapsulates a database. On the places 3 and 4, there are different stock exchange
agents running. Certainly, there is an AMA on every place.

At the time 0, the query is launched by the initiator agent by transferring it
to its AMA. The AMA forwards this query to every AMA it knows and adds a
globally unique query ID to this query. The AMAs on place 2 and 3 receive the
query and both forward it to the other AMAs they know (except for the sender).
Since all AMAs keep a log of the IDs of arriving queries, the new query can be
discarded by places 2 and 3 when it arrives the second time, such that (only)
place 4 gets newly involved into this query.

Now, each AMA first checks whether it has processed this query before by
comparing the query ID with the last queries it has processed. After having
evaluated the first condition of the query, the AMA on place 2 knows that it
cannot fulfil the requirements because there is no stock exchange agent running
on its place. The AMAs on place 3 and 4 notice that they have such a static agent,
but they have to evaluate the third condition that requires that these agents are
currently in operation. Since this is a dynamic information, the AMAs have to
ask the individual agents. Both stock exchange agents are active so the AMAs
on place 3 and 4 send their place ID to the AMA of the first place because these
places fulfil the requirements of the initiating agent. They also send their actual load information because it is needed as an order criterion.

After the time-out interval has passed, the AMA on place 1 evaluates the answers it has received. There are two possible places, but only one is desired, so it has to choose one out of them. Because place 4 has a lower load than place 3, the AMA of place 1 informs the initiator agent that it should migrate to place 4.

5 Related Work

Our work contributes to two aspects: migration and communication. Both of them are vital to a mobile agent system, so nearly every mobile agent system published addresses these issues. Related work on these two topics is discussed in the following two subsections separately.

5.1 Communication

For the purpose of inter-agent-communication, several different approaches have been published under the term "coordination" or "coordination language".

Basic algorithms for finding mobile agents, e.g., using logging, registration, and advertisement for relocating are discussed in [1]. All together, they assume that the agent which is to be found is known – a different focus then that we have.

Besides sending messages to an agent with a known identifier (including multicast messages), also some concepts with a higher level of abstraction have been proposed.

The concept of events implies that an agent has to register for a special kind of event. After this registration, it is informed every time such an event occurs. That kind of event-based interaction is commonly referred as "publish and subscribe" [10]. This approach can be found, e.g., in Concordia [12], Mole [2], and – in a limited way – also in the Aglet system.

Sessions are an approach proposed in [2]. It supports 1:1 communication of agents that may stay on different places, but which are not allowed to migrate
during a session is established. The most interesting idea related to our approach is the idea of badges: a set of strings is attached to every agent. It can be used to restrict the agents that are allowed to establish a session. As the most important difference to our approach, there is no structure within these badges and especially there are no tag value pairs like in XML which complicates sophisticated queries or even prohibits them. Also, in our case we support 1:n communication and there is no synchronisation needed between the communicating agents. This makes it much more easier to establish a connection if it is not known where the partners are.

The black board approach allows agent interaction with the help of a shared local data space. As a severe disadvantage, agents have to know the name of a certain blackboard in order to access the relevant information. A system supporting this approach is described, e.g., in [7].

An interesting special case of the black board approach is linda-like coordination as it is used for instance in the MARS [3] project. Here, associative methods are used for accessing information of the black board. Recently, it was decided to use XML for description purposes [4].

In order to summarize the above discussion, to our best knowledge, there is no other system offering such sophisticated methods for specifying agents on a high level of abstraction and without the requirement that a communication partner synchronizes or moves to the same place.

5.2 Migration

Taking a look at the second aspect, namely migration, we have not found any other system in the literature that allows for a declarative specification of the place a mobile agent should migrate to. Hence, the approach followed in our work where agents dynamically choose the place they should migrate to by using predicates is novel and considerably exceeds existing approaches in terms of flexibility.

6 Summary and Outlook

In this paper, we have presented a new approach for specifying both places to migrate to and agents being the recipients of messages. By using predicates, this specification can be done in a declarative and thus very flexible way.

The architecture is based on Aglets and extends this framework with new features like a peer-to-peer network, the possibility to describe agents and places using XML documents, and by allowing agents to query over these documents. Hence, it brings mobile agent systems and peer-to-peer configurations together.

Fundamental to our architecture is the concept of AgentsManagementAgents (AMAs), implementing the services mentioned above. AMAs are responsible for forming the peer-to-peer network, for the management of the meta data and also for the evaluation of queries.
In our context, queries consist of two parts: Firstly, queries comprise a meta-
level that describes how the evaluation process is driven. Secondly, predicates
are used for driving the actual selection process.
Such queries are sent to the local AMA from agents interested in migrating
to other hosts or in sending messages to other agents. This AMA pushes the
query over the network to the other AMAs which also forward the query and
evaluate whether or not this place or agents on this place fulfil the require-
ments of the query. Information about matching agents or places are sent to the AMA
of the initiating agent (this is also important for maintaining the peer-to-peer
network, i.e., for increasing the number of direct links to other places/AMAs).
This AMA chooses the fitting entities.
With this approach, we support location transparency as well as code and
data mobility and so combine the advantages of mobile agents and RPCs.
In our future work, we aim at adding additional guarantees known from
data bases to mobile agents. In particular, we are looking at these agents as a
special transaction. Hence, agents accessing shared data need to be synchronized
but also need support for failure handling strategies within the same framework.
To this end, our goal is to apply ideas of transactional processes [11] to mobile
agent systems.
The intended result will be a mobile agent framework allowing an easier way
to program mobile agent groups. Combined with execution guarantees, it will
extend the basic framework and is supposed to allow for new kinds of application
in industrial strength.

References

2. J. Baumann, F. Hoh, N. Radosunklis, R. Rothermel, M. Strasser: Communication
3. G. Cabri, L. Leonardi, F. Zambonelli: Reactive Tuple Spaces for Mobile Agent
   Coordination, 2nd Int. Workshop on Mobile Agents, Stuttgart, Germany, 1998
4. G. Cabri, L. Leonardi, F. Zambonelli: XML Dataspaces for Mobile Agent Coordi-
   nation, Symposium on Applied Computing, Como, Italy, 2000
   Publishing Company, Reading, MA, 1992
7. P. Dömel, A. Lingnau, O. Drobnik: Mobile Agent Interaction in Heterogeneous
   Environments, 1st Int. Workshop on Mobile Agents, Berlin, Germany, 1997
8. M. Fernandez, J. Siméon, P. Wadler: XML Query Languages: Experiences and
   Exemplars, 1999, available from
   http://www-db.research.belllabs.com/user/simeon/xquery.ps
9. D. Lange: Programming and Deploying Java Mobile Agents with Aglets, Addison
   Wesley Logman, Reading, MA, 1998
    Wiley, New York, 1999
A Scalable and Secure Global Tracking Service for Mobile Agents

Volker Roth and Jan Peters

Fraunhofer Institut für Graphische Datenverarbeitung
Rundeturmstraße 6, 64283 Darmstadt, Germany
{volker.peters}@igd.fhg.de

Abstract. In this paper, we propose a global tracking service for mobile agents, which is scalable to the Internet and accounts for security issues as well as the particularities of mobile agents (frequent changes in locations). The protocols we propose address agent impersonation, malicious location updates, as well as security issues that arise from profiling location servers, and threaten the privacy of agent owners. We also describe the general framework of our tracking service, and some evaluation results of the reference implementation we made.

Keywords: mobile agents, tracking, agent name, security.

1 Introduction

Early research on mobile agent systems concentrated on how to migrate mobile agents. Many agent systems exist today and there is a notable shift towards research in how to best support transparent communication between mobile agents [20,16,10,11]. Transparent means that agents need not be aware of the actual location of agents with which they wish to communicate.

Two general problems must be solved in order to achieve transparent communication: first, the peer agent must be located, for instance by means of a tracking service that maps an agent’s location invariant name onto its current whereabouts. Second, messages must be routed to the peer. This can become difficult since mobile agents might “run away” from messages. Guaranteed delivery is addressed for instance by Murphy, Picco, and Moreau [11,10].

In this article we address the problem of establishing a public global tracking service for mobile agents, which scales to the Internet and accounts for the particularities of mobile agents (frequent changes in locations). Public means that lookups of agent locations are not restricted in principle, yet we would like to account for security and privacy issues arising for agent owners in using such a tracking service. More precisely, the following problems must be addressed:

– tracking service updates and lookups must be fast. Since mobile agents can migrate at any time a huge rate of updates must be expected.
– The load must be distributed between a sufficient number of tracking servers. A suitable unambiguous mapping between agents and tracking servers must be established.
– The number of tracking servers must be gradually scalable to increasing demand.
tracking services bring security problems that must be addressed properly. Yet heavy-weight cryptography (e.g. mutual authentication, public key infrastructures) has an overhead that most probably counters the demand for fast lookups and updates.

In this article we describe our approach to solving these problems. Section 2.1 introduces the notation and conventions we use in this paper. Section 2.2 motivates and describes the architecture and the basic protocols that we developed. Our principle goal is to reduce application of costly public key cryptography to the bare minimum while achieving a good tradeoff between performance and security. We also made a reference implementation of these protocols. Section 3 presents the setup and the results of our evaluation of the reference implementation’s performance. Related work is discussed in Sect. 4, followed by our brief conclusions in Sect. 5.

2 Tracking Agent Locations

Several models of tracking agents are conceivable. Aridor and Oshima [1] already gave an initial discussion of agent tracking services and suggested three methods of locating agents: brute force, logging, and redirection. Milojićević et al. distinguish four models [9] which incorporate those of Oshima and Aridor: updating at home node, registering, searching, and forwarding. We discussed these models in greater depth already in [12], and came to the conclusion that registering is our mechanism of choice. Registering is a classic server-based approach: one or more dedicated servers provide associative mappings from agent names to agent locations. The interesting part is the structure of the name space, which server is responsible for which part of it, the way updates and lookups are handled, and security mechanisms, of course.

2.1 Notation and Conventions

The description of our protocols uses the notation given below. We will write encryption of some plaintext into a ciphertext symbolically as $c := \{m\}_K$, where $K$ is the key being used. A digital signature will be written as an encryption with a private signing key $S^{-1}$. We will write $S^{-1}(m)$ when we refer to the bare signature rather than the union of the signature and the signed data. We assume that the identity of the signer can be extracted from her signature. A cryptographic hash of some input will be written $h(m)$. When $A$ sends some message $m$ to $B$ we will write $A \rightarrow B \backslash m$. For ease of reading, we refer to some entities by their nicknames, e.g. Alice and Nick. In general, Alice plays the role of an agent’s owner, and Nick plays the tracking service. For simplicity, we do not distinguish between an entity and its identity, this should become clear from the protocol context. The itinerary of Alice’s agent is written as $i_{\alpha}, \ldots, i_{\gamma}$, where $i_{\alpha} = \text{Alice}$ and $i_{\gamma}$ is the host currently visited by the agent.

2.2 Protocols

We make a number of assumptions on the structure of mobile agents. These assumptions are not required for the tracking service per se, but are important for some of its security properties. Alice prepares her agent $I_{\alpha}$ as follows:
\[ \Pi_\phi = \{ (P, S) \mid S_A^{-1}, V_0 \} \]
\[ \phi = h(S_A^{-1}(P, S)) \]

where \( P \) is the agent’s program, \( S \) is its static part, and \( V_0 \) is its initial variable (or mutable) part. Alice signs her agent’s program along with its static part for the purpose of authentication and integrity protection. For the sake of security, \( S \) should be unique for each agent instance that uses \( P \) (see [13]). We denote \( \{P, S\} \) as the agent’s kernel, and define the agent’s name \( \phi \) as a cryptographic one-way hash of the kernel’s signature.

We refer to \( \phi \) as the agent’s implicit name, because it is not a given name but derived from the agent instance itself. The hash function \( h \) must be preimage and 2nd preimage resistant (see e.g. [8]). For practical purposes, the SHA (Secure Hash Algorithm) [6] can be used.

Implicit names have a number of useful security properties. Agents cannot impersonate other agents, and the chance to create another agent that maps to the same implicit name (either accidently or on purpose) is negligible. Two agents of the same owner cannot be linked by means of their implicit names. It is virtually impossible to guess agent names. Furthermore, implicit names can be used to bind data, which is acquired by a mobile agent at runtime, securely to that particular agent instance [13].

It may be argued that a random session key could be used rather than an implicit name. However, such a random key would be chosen explicitly. This defeats the purpose of the implicit name because a malicious host could provoke a name collision (e.g., it may replace an agent with another of its own that can be controlled conveniently, yet receives all messages directed to the original agent).

Depending on the number of mobile agents that must be served globally, a scaling factor \( l \) is chosen. The range of hash function \( h \) is divided by this scaling factor into \( 2^l \) subranges. Each subrange is identified by a number in the range \([0, 2^{l} - 1]\). A tracking server is set up for each subrange, and the mapping from subrange numbers to tracking servers is distributed to all agent servers that use the tracking service (see also Fig. 1). As long as few tracking servers are required, this can be done by means of a master list, comparable to the beginning of DNS (Domain Name Service) use. Later on, a special DNS domain can be set up, where the number of the subrange serves as the host name of the tracking server that is responsible for this subrange.

**Fig. 1.** The implicit name \( \phi \) of length \( n_l \) bits is split into \( l \) bits that identify the tracking server, and \( n_l - l \) bits that are used by the tracking server to distinguish between different agents.
What we propose is in fact a global hashable where each of its slots is managed by a dedicated tracking server. Agents are assigned to slots by means of a cryptographic hash function.

When Alice initializes her agent, she chooses a random initial cookie $C_{i_0}$, and sends her agent on its way with this cookie. The cookie must be big enough to make any chance of being found by guessing or exhaustion attacks negligible. Each hop runs the same protocol when the agent is received:

$\begin{align*}
(1.1) & \quad i_{n-1} \rightarrow i_n : \Pi_{i_0} C_{i_{n-1}} \\
(1.2) & \quad i_n \rightarrow N : \phi, i_n, C_n, C_{i_{n-1}} \\
(1.3) & \quad N \rightarrow i_{n+1} : m \in \{\text{ok, error}\} \\
(1.4) & \quad i_{n+1} : \text{executes } \Pi_{i_0} \text{ until } \Pi_{i_0}, \text{ migrates} \\
(1.5) & \quad i_n \rightarrow i_{n+1} : \Pi_{i_0}, C_{i_0}
\end{align*}$

When $i_n$ receives the agent from $i_{n-1}$ it updates the agent’s position in the location register of Nick (the tracking server) with its own name. Nick’s name is derived directly from the agent’s implicit name and the mapping function, based on the $l$ bits subrange identifier. The update operation is authenticated by means of the cookie received with the agent. Each host generates and sends a new cookie with its update message. The new cookie is passed together with the agent to the agent’s next hop. Hence, each host hands over authority to make location updates to the next hop.

Once an update operation is completed, previous hosts have no access to the location register any more. A host cannot hand off an agent and keep control of the update register at the same time because the next hop would get an error in return of its own update request, thus uncovering the attempt. The host of an agent may nevertheless update the agent’s location register with (a series of) bogus locations, thus creating the impression that the agent visited hosts it never visited actually (using IP spoofing if necessary). On the other hand, the host cannot intercept messages that are routed to the agent, without revealing the information sink. We clearly had to make a compromise here, and decided against heavy-weight cryptography in favor of efficiency.

Nick updates its entry for the given implicit name if (1) no mapping of the given implicit name yet exists, or (2) the old cookie $C_{i_{n-1}}$ matches the one stored with the location entry. In that case, Nick updates the stored cookie to the new cookie $C_{i_n}$ that came with the update request. The location entry is deleted if the host name portion of an authenticated update operation is empty. We refer to this as a clear request. Alice sends a clear request to Nick when her agent returns, and hosts send clear requests when the agent terminates without migrating.

However, if a tracking server is malicious then it can uncover Alice’s identity by observing the host from which either the initial update or the clear request was sent (agents likely return to their owners). Alice can avoid this in two ways: firstly, she never sends the initial update or final clear message – the initial update is then done implicitly by the agent’s first hop. Secondly she routes the message through a relay service, for instance another agent of hers, which she sent to a neutral host that allows her agent network access. The important point is that the relay host and the tracking server do not collude.

The lookup protocol is also quite simple. Anybody who knows the implicit name of the agent (for instance Alice) can look up its current location by querying the tracking server.
After a fixed amount of time, Nick expires and garbage collects entries unless they are refreshed. The refresh protocol, which is given below, supports bulk refreshes of multiple entries in order to save bandwidth and to reduce the number of connections that must be opened.

\[
i_{n} \rightarrow N : \phi_{1}, \phi_{2}, \ldots, \phi_{k}
\]

Please note that, although \(i_{0}\) is given as the origin of refresh requests, they can be sent principally by all entities that know an agent’s implicit name. This feature comes handy when attackers attempt to force expiration of a particular entry by launching a denial of service attack against the respective agent’s current host.

The protocols use few and simple cryptographic operations. In particular public key operations are avoided, which commonly require a public key infrastructure, and carry a heavy burden. On the downside, our protocol does not account for the confidentiality and integrity of the cookies that are transmitted. If Alice is willing to disclose her agent’s origin,\(^1\) then she can use a more secure variant of the protocol, whose differences to protocol (1) are the following:

\[
A \rightarrow N : \phi, i_{n}, C_{n}, 0\]
\[
i_{n} \rightarrow N : \phi, i_{n}, \{C_{n}\}_{C_{n-1}}, h(\phi, i_{n}, C_{n}, C_{n-1})
\]

Alice encrypts the initial update request with the public key of the tracking server. Subsequent updates from agent servers can be protected by means of the chain of cookies, without the use of public key cryptography. This is done by encrypting\(^2\) the new cookie with the old one. The integrity of the data is assured by a message authentication code computed on the implicit name, the new location, and the cookies. Please note that a single leaked cookie allows decryption of all subsequent cookies (given that the adversary intercepts subsequent update requests sent to the tracking server). This is the price that must be paid for not using public key cryptography for all but the first update. Please also note that the current cookie must be transported with the agent. Hence, agents must also be transported over confidential channels for maximum security. Confidentiality is achieved, in the face of network eavesdroppers, only if both conditions are fulfilled.

### 2.3 Agent Naming and Tracking Framework

The general framework distinguishes between agent name services and agent tracking services as is illustrated in Fig. 2. This distinction is conceptually comparable to the way file services are implemented in Amoeba [18, §14.6]. Amoeba’s bullet server serves files based on a flat name space, while the directory server maps a hierarchical name space onto the flat name space managed by the bullet server.

---

\(^1\) Again, Alice can disguise her identity by using a relay agent.

\(^2\) In our reference implementation we use a simple XOR for encryption, which should be sufficient given that the new cookie is generated randomly.
Figure 2. The framework distinguishes between a name service that maps symbolic agent names onto their implicit names, and a tracking service that maps implicit names onto locations.

We did not yet design the agent name service, but we anticipate that it resolves symbolic, function specific, and user friendly names onto the implicit names required by the tracking service. For instance, a local name service can map the function specific name “printer” onto the implicit name of the nearest printer agent. If a particular printer is to be used then that printer agent’s implicit name can be used to address it unambiguously. The aforementioned distinction also makes sense from a practical and security point of view. Symbolic name mappings are probably more stable than mappings of agent names to agent locations, and they have different security requirements as well, as the given example suggests. However, we have to defer further discussion of the name service, and concentrate henceforth on the design of the tracking service.

Figure 3 gives an overview over the general configuration of the tracking service components. The tracking server provides lookup and update operations as described in Sect. 2.2. Additionally, each tracking server supports a refresh operation, and timeouts for its mappings. Periodically, clients have to refresh mappings of agents that don’t migrate, in order to keep the mappings valid. Multiple mappings can be refreshed with a single request. The validity period must be well-defined and sufficiently large so that no unreasonable network load incurs. This prevents stale entries from clogging up a tracking server’s database forever.

A proxy acts as a write-through cache, and is meant to be an optimization for looking up agents in the administrative domain covered by that proxy, e.g. an organization’s local area network. The use of proxies requires an extension of the protocol: whenever a mobile agent leaves the administrative domain of a proxy, the proxy’s mapping for that agent must be invalidated. Lookups that cannot be resolved by the proxy are forwarded to the appropriate global tracking server. The proxy also assures that lookups for agents in a local area network succeed in the face of a broken link to the external network.

The purpose of the administration server is to provide up-to-date information of the scaling factor (the number of bits of each implicit name that identify the subrange of the name space to which the implicit name belongs), and the mapping from subranges to tracking servers. This can be thought of as an alternative to the DNS-based scheme that we mentioned in Sect. 2.2. Since information provided by administration servers is expectantly rather static, regular caching strategies can be used to achieve scalability of this service.
The framework also accounts for a diagnostic component that can be used to query servers and proxys for statistics and excerpts of current mappings (subject to access control). Finally, Fig. 3 shows the relay agent whose relevance for the privacy of agent owners is described in Sect. 2.2.

We made a reference implementation of the tracking server, proxy, client, and a diagnostic tool in the Java programming language. All components were integrated and tested in our mobile agent server SeMoA [14]. The tracking server and proxy run as daemons, which listen on network ports and dispatch requests to a pool of handler threads. Both proxys and tracking servers are backed by a balanced binary search tree [5], which is kept in memory and is accessed by means of a generic interface. Hence, it is straightforward to provide adaptors that interface to a powerful database backend. A detailed description of the reference implementation is beyond the scope of this paper.

2.4 Discussion

Systems such as Globe [19] base their scalability on the assumption that some kind of coherence in the movement of mobile objects can be used to optimize the operation of the distributed tracking system. We believe that this assumption is overly optimistic for mobile agents, which operate on a global scale, and whose movement is not bound by physical constraints. With our approach, tracking servers may have to manage agents on the opposite side of the globe (no pun intended). This is the starting point of our discussion below.

Consider $m = 2^k$ name servers. If the probability of a random break down of a name server is equally distributed and Alice would be able to make a random pick among the name servers then her chances of picking one of $k$ broken ones among $m$ are $k/m$. If she takes our approach then her chances are also $k/m$ given $h$ has a reasonably equal distribution. If Alice knows that a particular name server is down and $\phi$ is mapped to
it then Alice can simply add a random nonce to $S$ and try again. The chances that she
does not succeed after $t$ tries is $(k/mt)^t$. If half of the name servers are broken and Alice
makes 4 tries then her chance of failing to generate an agent that is mapped to a working
name server are 0.0025.

If Alice knows where $\Pi_o$ is about to go then Alice might want to choose a tracking
server with good connectivity to the destinations of $\Pi_o$. In our approach, the packets
between Bob and Nick might circle the globe in the worst case. However, if $\Pi_o$ travels
around the globe and Alice makes any pick among the name servers then the effects are
the same. If Alice still wants to pick one of a particular set of name servers then her
chances of succeeding heavily depend on the actual number of name servers and the
number of name servers she is willing to take. More precisely, her chances to pick one of
$k$ name servers among a total of $m$ servers after $t$ tries is $1 - (m-k)/m$. If 1% of the
name servers are acceptable for Alice then her chances to pick one of those after 4 tries
are less than 0.04, and less than 0.1 after 10 tries. Alice will succeed with a probability
of approximately 0.9 after 230 tries. On a Pentium II 400MHz we measured less than 4
seconds for computing 230 SHA-1 hashes on about 10K of data (Java implementation).

If a tracking server goes down then entries are certainly lost. However, once the server
is available again, agents will be registered as a consequence of regular update request
as soon as they migrate. This leaves a window that can be used by malicious servers to
“hijack” location entries of agents managed by that tracking server. Nevertheless, we
believe that our approach strikes a good compromise between security, scalability, and
flexibility.

3 Evaluation Results

Our tests took place in a 100 MBit/s switched LAN that connects a couple of hundreds of
workstations and personal computers, and is used by about two hundred researchers and
students. We run our software on several Sun Ultra 10 workstations (UltraSPARC-IIi
333 MHz, Solaris 8). The client and proxy machines were equipped with 256 MB main
memory, while the tracking server did have 512 MB. We used the HotSpot VM of Java
Version 1.3.1 Beta with native thread support and sunjit enabled.

First, we tested the capacity and performance of our storage backend. The tracking
server was able to hold up to $2 \cdot 10^9$ entries before the system ran out of memory. This
means that, given an extreme of $5 \cdot 10^9$ Internet users each running 100 mobile agents
simultaneously, about $25,000$ tracking servers would be required to keep all entries.
This is less than 0.025% of the hosts in the Internet, according to ISC estimates$^3$ at the
time of writing.

Next, we let up to eight clients send requests concurrently. Table 1 gives the response
rates we measured in tests with a single client, sorted by request type. Encrypted reg-
istering was slowest, as could be expected. However, this type of request is required
only once per agent. In this test the tracking server handled about 200 agent lookup re-
dquests per second, which includes processing overhead at the client (clients start requests

$^3$ NUA estimates there were more than 400 million users online in the Internet on December 2000,
source: http://www.nua.com/surveys/how-many_online

$^4$ ISC estimates there were more than 100 million hosts in the Internet on January 2001, source:
http://www.isc.org/ds
Table 1. This figure shows the size of request packets, and average processing time of the tracking service with one client, by request type. The lengths marked with † may differ depending on the length of the stored location reference.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Mean time</th>
<th>Requests/s</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup</td>
<td>30 bytes</td>
<td>4.7 ms</td>
<td>213</td>
<td>tracking</td>
</tr>
<tr>
<td>register encrypted</td>
<td>421 bytes†</td>
<td>201 ms</td>
<td>5</td>
<td>init</td>
</tr>
<tr>
<td>update</td>
<td>103 bytes†</td>
<td>8 ms</td>
<td>125</td>
<td>update</td>
</tr>
<tr>
<td>register plain</td>
<td>103 bytes†</td>
<td>5 ms</td>
<td>200</td>
<td>proxy</td>
</tr>
</tbody>
</table>

sequentially). Figure 4 shows the response rates we measured for concurrent lookup requests with one to eight clients. With two or more clients, the response rate jumps from about 200 requests per second to roughly 325, and remains more or less stable at this mark (with one client, the server has idle time, with two or more it becomes congested). Table 4 shows how response times develop with an increasing number of clients. With about 80 clients, requests take longer than 15 seconds to process, which causes network connections to time out.

We also measured the impact of the tracking service integration on the migration time of mobile agents in the SeMoA server. Without tracking service integration, we measured an average of 1.178 seconds per migration of a simple benchmark agent, compared to 1.18 with location tracking, which we consider tolerable.

4 Related Work

The Globe [19] system is a distributed directory designed to support billions of references to mobile objects. However, the authors acknowledge that their hierarchical approach is not scalable enough to fulfill this goal due to the enormous storage demands and relatively large number of requests that must be handled by higher-level directory nodes. In order to overcome these problems, they propose to use the first 12 bits of an object’s globally unique handle as the identifier of directory subnodes, which share the load on their directoy level. This approach equals the one we chose in order to provide scalability.

The notion of put-ports and get-ports in Amoeba [18, pp. 607] can be regarded as an analogy to the implicit naming scheme we propose for agents in this article. An Amoeba server process registers with the Amoeba kernel using a (private) get-port, and the according put-port is computed by the kernel by means of a one-way function. Processes that wish to communicate with the server process address packets to the put-port. This prevents intruders from impersonating server processes. The server process can be regarded as a mobile agent, and the put-port as its name. In contrast to Amoeba we do not allow free choice of the private get-port, but compute the put-port directly from the agent’s unique kernel, which is required to prevent one agent from impersonating another with the help of a malicious host that leaks the equivalent of get-ports.

Several other schemes for locating mobile agents, and routing messages among them were proposed in the past, e.g. [3,11,4,20,17,7]. Some of these approaches assume that there is a logical network of connected agent servers [3,11,7], and routing of agents or messages is done along the edges of this graph. In the case of [7], the graph must actually
be a balanced tree. However, any approach that builds on a particular network topology makes sense only if mobile agent systems are implemented on the network layer as part of routers. Most of the contemporary mobile agent systems are implemented on the application layer, though. From the perspective of the application layer, the Internet is a fully connected graph. Hence, a logical topology that is layered on top of the physical structure of the Internet creates undesired and unnecessary routing overhead. The logical routing may even run counter to the actual physical routing. Additionally, the approaches described in [3,7] put the burden of setting up and maintaining the logical structure on administrators; a job that, in our opinion, quickly spirals out of control.

In particular, the approach described in [3] is not scalable. Each node in the tree has storage requirements proportional to the number of mobile agents managed by it, and update rates proportional to the rate of migrations that start or end in its subtree. In particular, the root node has to cope with all of the traffic.

Strategies based on forwarding pointers and dynamic shortening of pointer chains are proposed, e.g. in [16,10]; they are also used in Mole for the purpose of orphan detection [2]. The disadvantage of this approach is its lack of robustness, a single broken or timed-out link makes the agent unreachable.

The Mobile Object Workbench [4] supports a hierarchical directory service for locating objects that moved. Wojciechowski et al. [20] use a combination of registering and forward references. Forward references act as a cache. In case of a miss, the central server is asked to forward the message, and the invalid forward reference is updated. Di
Stefano et al. [17] propose the use of location servers, where each server is responsible for all agents in its domain. Each agent has a home server that can be derived from a location-specific part of the agent’s name. Whenever the agent enters a new domain, the servers responsible for the old and new domain, as well as the home server are updated. Lookups for agents not in the local domain start at the home server.

A detailed discussion of all these approaches is beyond the scope of this paper, and is well worth a paper of its own. To the best of our knowledge, none of the approaches described above address security issues, and few seriously address Internet-wide scalability.

5 Conclusions

In our paper, we propose a framework and protocols for a secure and scalable global tracking service for mobile agents. We do not presuppose that a mobile agent’s migration pattern is governed by a coherence principle that could be used to achieve scalability. We must anticipate a high rate of update requests of agent locations, and thus our approach was designed to scale without caching mechanisms. Our approach resembles a global hash table, where the hash function fulfills some security requirements.

The protocols we devise have a number of advantageous security properties. In particular, malicious location updates by unauthorized hosts are prevented. Each host
must hand off authority to update an agent’s location register when the agent migrates. Agents cannot impersonate other agents with regard to their names. Furthermore, the implicit naming scheme for agents prevents malicious tracking servers from profiling agent owners by means of their agents’ movements, given that an agent’s host does not collude with the adversary. This protects the privacy of agent owners in the face of omnipresent tracking servers. All this is achieved with a minimum of cryptographic overhead, which is an important requirement for scalability.

Furthermore, we made a reference implementation of our protocols, and present the results of its evaluation. The outcome is as good as one could expect from a Java implementation, and can be further improved by implementations that are optimized for the processor architecture of the machines on which the tracking server shall run. We are well aware that laboratory settings hardly give significant evidence for a system’s applicability when used in the field. Therefore, we would very much like to tests our implementation in a larger scale, and welcome interested parties that would like to take part.

Acknowledgements. This paper elaborates on initial ideas presented in [12]. In particular, it contributes to the proxy approach, the encrypted update protocol, as well as a report on the evaluation of our reference implementation. We’d like to thank the anonymous reviewers for their detailed and constructive comments which helped us to improve the paper.

References


Translating Strong Mobility into Weak Mobility*

Lorenzo Bettini and Rocco De Nicola
Dipartimento di Sistemi e Informatica, Università di Firenze
{bettini,denicola}@dsi.unifi.it

Abstract. Mobile agents are software objects that can be transmitted over the net together with data and code, or can autonomously migrate to a remote computer and execute automatically on arrival. However, many frameworks and languages for mobile agents only provide weak mobility; agents do not resume their execution from the instruction following the migration action, instead they are always restarted from a given point. In this paper we present a purely syntactic translation process for transforming programs that use strong mobility into programs that rely only on weak mobility, while preserving the original semantics. This transformation applies to programs written in a procedural language and can be adapted to other languages, like Java, that provide means to send data and code, but not the execution state. It has actually been exploited for implementing our language for mobile agents X-KLAIM, that has linguistic constructs for strong mobility.

1 Introduction

The diffusion of Wide Area Networks (WAN) has stimulated the introduction of new programming paradigms and languages to model interactions among hosts by means of mobile code [25], a key concept in distributed programming. By this it is intended software that can be sent to remote sites and executed on arrival. A particular example of mobile code is represented by mobile agents [13, 31]: these are software objects, with data and code, that can be transmitted over the net, or can autonomously migrate to a remote computer and execute automatically on arrival. Mobile agents have been advertised as an emerging technology/paradigm that provides means to design and maintain distributed systems more easily [16].

Three kinds of mobility have been identified [8,14]:

- weak mobility: the dynamic linking of code arriving from a different site;
- strong mobility: the movement of the code and of the execution state of a thread to a different site and the resumption of its execution on arrival;
- full mobility: the movement of the whole state of the running program including all threads' stacks, namespaces and other resources. This is a generalization of strong mobility that makes the migration completely transparent.

* This work has been partly supported by MURST Projects SALADIN and TOSCA.

© Springer-Verlag Berlin Heidelberg 2001
Java [1] is often used to develop distributed applications with mobile code and mobile agents; indeed Java provides machine-independent byte-code interpretation and dynamic linking which make the development of this kind of applications quite easy. Unfortunately, Java only provides weak mobility since threads’ execution state (stack and program counter) cannot be saved and restored; this implies that once an agent arrives to a remote site, it has to be restarted from the beginning. Since many tools and systems for mobile agents are almost completely implemented in Java, they all rely on weak mobility. Some examples of such systems are Mole [24], Odyssey [11], the successor of Telescript, and Aglets [15].

Systems such as Telescript [30], Agent Tel [12] and ARA [19] provide strong mobility by using a dedicated language interpreter to capture and resume the process’ execution state.

Full mobility is provided by LOCUS distributed operating system [29]. Full mobility is necessary if process migration is used, for instance, for load balancing: the migration has to be completely transparent.

We would say that the notion of mobility at the heart of the classical concept of mobile agent is strong mobility: the execution state of a migrating agent is suspended, and its stack and program counter are sent to the destination site, together with the relevant data. At the destination site, the stack of the agent is reconstructed and the program counter is set appropriately, i.e. to the first instruction after the migration action.

Weak mobility is in contrast with standard definitions of mobile agent, because automatic resumption of execution threads is one of the main features of mobile agents (it exalts their autonomy). However, as we said, many systems provide only this kind of mobility. Because of this, there have been different attempts to “simulate” strong mobility on top of weak mobility: it is possible to implement almost the same program functionality by explicitly coding the agent code on top of a weak mobility environment by using a notification event based model. In Aglets framework, when an agent arrives to a remote site, its callback method onArrival will be called and the programmer will be able to execute some actions according to information that is stored in the agent’s state and to the parameters that are passed to this method. Before leaving, the method onDispatching is invoked; the programmer can use this method to save information about the state of the agent.

However, the known approaches for obtaining strong mobility on top of weak mobility require programmers to explicitly write code to capture the relevant execution state information into some variables and transfer these values together with the agent (typically in the shape of agent class data members). This is error prone and demands additional coding.

Moreover if the code of a migrating agent is split into several callback procedures, then some execution flow analysis (and consequent checks and optimizations) cannot be performed by the compiler. For instance if a compiler is analysing this code:

\[
x := 10; f(); migrate; print x;
\]
when processing the print $x$ instruction it can safely assume that $x$ is initialized
(and, if useful, also that $f(x)$ has been called). If strong mobility is not provided
by the language, then such code should be rewritten, by splitting it in many parts, possibly across procedures, and the compiler could hardly perform the
same accurate analysis.

In this paper we present a purely syntactic translation process for transfor-
mimg programs that use strong mobility into programs that rely only on weak
mobility, while preserving the original semantics. This transformation works on
programs written in a procedural language and can be adapted to other langua-
ges, e.g., Java, that provide means to send data and code, but not the execution
state. To illustrate this transformation we will use a simple prototype Pascal-like
language.

We shall not consider full mobility, since it can be considered orthogonal to
mobile agents, and moreover it would require a strong support from the operating
system layer; if implemented with a language that provides weak mobility it can
be very hard to synchronize all threads of a migrating agent, and would imply
performance penalties [10,14].

The rest of the paper is organized as follows: Section 2 informally introduces
our translation procedure, which is described in Section 3; Section 4 gives a
sketch on how this technique is exploited in our language for mobile agents,
X-KLAIM, that provides linguistic constructs for strong mobility. A final section
about future works describes some similar approaches.

2 The Basic Idea of the Translation

The idea at the heart of our transformation is that an agent, that migrates,
stores in its own state also information about the point from where to restart
after migration; by using this information, stored in a variable called mark, at
the beginning of the execution the agent will perform a jump to a certain point
of the code. A different mark will be stored for every procedure instance; this
means that if procedure $P$ recursively calls itself, every called instance will refer
to its own mark: these marks may indeed be different.

Moreover we must detect at run time whether a procedure call involves a
procedure that actually executes a migration command: for instance the proced-
ure can be turned by the translation into a function which returns the state of
migration (the constant MIGRATED will indicate that the procedure has migra-
ted). This way the translated code, after a procedure call, will check the return
state, and in case the procedure has migrated, it will, in turn, terminate the
execution and return to the caller the value MIGRATED; the caller stack will be
cleaned, and the agent will terminate the execution in the departure site. For
every procedure instance $P$ the current caller, i.e. the procedure that called $P$,
is also stored ($P.caller = Q$ means that $Q$ has called $P$) and the variable state
represents the state of its execution: during migration state will contain the con-
stant MIGRATED and upon arrival it will contain ARRIVED. The variable state can
be set to ARRIVED at the destination site, before the agent execution is resumed.
So when \textit{state} is accessed within a transformed procedure, it will refer to the agent that executes the procedure.

We will use the following notation:

- $P, Q, R, \ldots$ represent procedures, and $P()$ is the instruction that calls procedure $P$;
- $CB1, CB2, \ldots$ represent blocks of instructions (code blocks);
- $P = CB$ represents the definition of procedure $P$ with body $CB$;
- $[P], [CB], \ldots$ represent translated code, and $[X] \triangleq X'$ means that the application of the translation to $X$ results in $X'$, where $X$ can be either $P$ or $CB$;
- $P \rightarrow Q$ indicates that procedure $P$ calls procedure $Q$, in this case $P$ will be expressed as $P = CB1; Q(); CB2$, where $CB1$ and $CB2$ are possibly empty;
- \texttt{goal} is the action that makes the agent migrate to the site $l$.

Basically the call stack will be simulated backwards; thus if an agent executes the following procedure calls $P \rightarrow Q \rightarrow R$, and $R$ causes the agent to migrate to $l$, on $l$ the sequence $R \rightarrow Q \rightarrow P$ will be executed. The call stack is reconstructed by starting from the last called procedure (the one that migrated); before the procedure returns, in case it has migrated, it will explicitly call the \textit{caller}. Thus the stack is not reconstructed at the destination site, instead calls and returns are simulated, to obtain, anyway, the same result of the original stack.

For example, given the procedure $P = CB1; Q(); CB2; R(); CB3$, where both $Q$ and $R$ can migrate, and $CBi$'s do not execute migrations, the translated code will be something like the following, where \texttt{afterQ} and \texttt{afterR} are the labels inserted to permit jumps to the appropriate continuation:

\[
\begin{align*}
[P] \triangleq \\
& \text{if } \text{mark} == \text{afterQ} \text{ then goto afterQ} \\
& \text{if } \text{mark} == \text{afterR} \text{ then goto afterR} \\
& \text{CB1; } \\
& \text{mark} := \text{afterQ; } \\
& \text{if call}(Q()) == \text{MIGRATED} \text{ then return MIGRATED; } \\
& \text{afterQ: } \\
& \text{CB2; } \\
& \text{mark} := \text{afterR; } \\
& \text{if call}(R()) == \text{MIGRATED} \text{ then return MIGRATED; } \\
& \text{afterR: } \\
& \text{CB3; } \\
& \text{if status == ARRIVED then } \\
& \quad \text{caller(); } \\
& \text{else } \\
& \quad \text{return NOT.MIGRATED; } \\
& \text{end if}
\end{align*}
\]

\footnote{We are not explicitly representing formal parameters, that however will be part of the state, together with local variables, that is transmitted during migration.}
where the auxiliary function call(Q) sets caller for Q to P, executes Q, possibly sets state to MIGRATED (if the migration succeeds) and returns the migration status of Q. Thus when the procedure is executed for the first time the execution starts from the first code block (CB1), since the two if tests will both fail (mark is unset at that time), while if it is executed after the migration, caused for instance, by the procedure Q, the execution starts from the label afterQ, since, before calling Q, the mark had been set to that label. The last part of the transformed program detects whether the agent is executed after the migration to a remote site; in that case the previous procedure is called, otherwise it simply returns to the caller procedure, specifying that the agent has not migrated. This way we can reconstruct the stack in case of migration.

However, many languages, including Java, do not have goto instructions. Indeed goto statements are considered harmful in many respects. They make harder program analysis that is essential, not only for verification, but also for optimization. Thus we will use if-blocks to simulate gotos. mark will be an integer variable and we will assume that at the first execution mark will be 0 and it will be increased during the execution; thus the code presented above becomes:

\[
\begin{align*}
[P] & \triangleq \\
& \text{if mark} \leq 0 \text{ then} \\
& \quad \text{CB1;} \\
& \quad \text{mark} \leftarrow 1; \\
& \quad \text{if call(Q())} = \text{MIGRATED} \text{ then return MIGRATED;} \\
& \text{end if} \\
& \text{if mark} \leq 1 \text{ then} \\
& \quad \text{CB2;} \\
& \quad \text{mark} \leftarrow 2; \\
& \quad \text{if call(R())} = \text{MIGRATED} \text{ then return MIGRATED;} \\
& \text{end if} \\
& \text{if mark} \leq 2 \text{ then} \\
& \quad \text{CB3;} \\
& \text{end if} \\
& \text{if status} = \text{ARRIVED} \text{ then} \\
& \quad \text{caller();} \\
& \text{else} \\
& \quad \text{return NOT.MIGRATED;} \\
& \text{end if}
\end{align*}
\]

Basically mark plays the role of the program counter that is exploited to know, in each procedure, the statements that have already been executed. The if statements, used to test the value of mark do enclose code blocks containing a migration action. Before every action that can lead to migration, mark is updated, and the if code block is closed after that action. By testing such value, we can skip code blocks that have already been executed, and “jump” to the actions that have to be executed upon arriving to a new site.

We make a basic assumption about agent mobility from one host to another: it is subjective and not objective, thus the migration is always started by the
agent itself and not by an external control. Moreover we do not address explicitly
the saving of the state (including local variables and actual parameters) of an
agent: we assume that this is handled within the target language because we
assume that it supports weak mobility, and thus it provides means to access and
save the state of a process. We are not explicitly handling migration failures;
the details would depend on the actual implementation and are not important
for the presentation of the transformation patterns.

3 The Translation

In this section we will show, in details, the transformation of the typical con-
structs of a prototype procedural language, including if and while statements.
We assume that migration is triggered by:

– the execution of a successful go@l action;
– the call of a procedure that migrates (in this case the call returns the constant
  MIGRATED).

The translations are given as translation patterns that are to be applied by
pattern matching to portions of code that can trigger migrations. if and while
statements will be translated only if their code blocks migrate (in the case of
if, if at least one of the two branches can execute a migration). We are not
describing the procedure, that should be executed before the actual translation,
to detect the code blocks that can migrate, since this can be easily implemented
by structural induction.

During the translation, for every procedure we will keep a variable, CurrentMark
where we store the current value for the variable mark that will be
inserted in the transformed code. In the translation patterns we have also a few
instructions enclosed by \{\}. These will be used by the algorithm during the ac-
tual translation process for updating auxiliary variables and will not be part of
the target code. The value of a variable that is maintained by the transformation
process, such as CurrentMark, will be inserted in the generated code by prefixing
$: such value will be inserted as an integer constant: the value of the variable
at that moment. The translation of a piece of code that executes a migration
action is depicted in Figure 1 (where CBl is assumed not to migrate); There
the \[\] on the right are used to indicate the recursive application of a translation
pattern.

It is important to notice that matching is performed by starting from the first
instructions of a code block and proceeding sequentially to the end. An example
of matching is shown in Figure 2.

Moreover if no pattern is applicable to a code block CBx, then that code is
enclosed in an if-block:

    if mark <= $CurrentMark then CBx endif

This happens for the last part of a procedure, or of a code block, whenever the
last instruction is not a migration action.
\[ [\text{CB1; migration action; CB2}] = \begin{cases} & \text{if } \text{mark } \leq \$\text{CurrentMark} \text{ then} \\ & \text{CB1; } \\ & \{\text{increment CurrentMark}\} ; \\ & \text{mark } := \$\text{CurrentMark} ; \\ & \text{if migration action succeeds then} \\ & \text{return NIGERATE;} \\ & \text{end if} \end{cases} \text{end if} \]

Fig. 1: Translation pattern for CB1; migration action; CB2 (where CB1 cannot migrate).

An example of matching for the pattern in Figure 1 is shown in Figure 2: we assume that at the beginning of a procedure (thus CurrentMark is 0) we encounter the code CB1 ; Q(); CB2 where the instructions of CB1 do not execute a migration, and the procedure Q can migrate. In the figure a bullet (●) indicates the current instruction of the translation that is to be executed according to the translation pattern, and in the box at the upper right corner the value of the auxiliary variable CurrentMark is shown.

Fig. 2: An example of matching for the pattern in Figure 1.

The translation is started by applying the translation process to all the procedures that can migrate, according to the pattern in Figure 4.
3.1 Translation of if

The translation of an if \( b \) then ... else statement requires some more operations. The main problem is due to the boolean expression \( b \) tested by the if: this expression could test the value of a variable, and we must make sure that it is not evaluated another time when resuming execution after a migration, since that value could make the test fail (it could have been changed before the migration action); even worse that expression may have side effects that would take place another time, if the expression was evaluated again. So we must ensure that the translated code tests that expression only the first time that if statement is executed.

We cannot insert the if \( mark \leqslant \ldots \) block inside the if \( b \ldots \) block, because, upon resuming from migration, \( b \) would be evaluated again; nor we can insert it before, because after migration the execution would not resume inside the if \( b \ldots \) block, in case the migration took place in the then or else branch.

To handle an if statement we introduce a local variable in the transformed code, named \( if\_exp_x \) (where \( x \) is an incremental number) that will store the result of the evaluation of \( b \) by checking the value of \( mark \) we ensure that this evaluation takes place only before the migration; moreover by using two other values during the translation IfMaxMark and ThenMaxMark, we will redirect the execution so that either the whole if block will be skipped (\( mark > IfMaxMark \)) or the then block will (\( mark > ThenMaxMark \)). Once again these values will be updated during the translation, and their values will be inserted in the translated code.

Notice that the values of these variables must be replaced in the generated code only after the if statement is processed, since the correct values are known only then. In order to specify this in the translation patterns, instead of prefixing $\&$, we will use \# (in this case the substitution is postponed); the code blocks that are involved are delimited by \{begin #block\} and \{end #block\}. When \{end #block\} is encountered in the transformation process, all the identifiers prefixed with \# in the scope of the nearest \{begin #block\} will be replaced with their current values. The translation of an if instruction is illustrated in Figure 5.

If \( mark > ThenMaxMark \) the then branch can be skipped, since it means that it had already been executed completely, or the else branch had been chosen, before the migration; if on the contrary \( mark \leqslant ThenMaxMark \) then the expression is evaluated only if the condition \( mark \leqslant CurrentMark \) is true,
otherwise the then branch had already been chosen, and it will be chosen again, since if.exp_r is set to true.

Notice that an if statement could be contained in another one, and this is consistent with the fact that the value of ThenMaxMark is replaced in the transformed code only after the then branch was processed (the inner {begin #block} and the value of IfMaxMark when also the else branch was processed (outer {begin #block}).

The last instruction in the enclosing generated if block increments the value of the mark, so that when resuming execution after a migration the original if block will be completely skipped, if either the then or the else branch had already been completely executed before migration. The last instruction, mark := #IfMaxMark, could have also been written as mark := $IfMaxMark or mark := $CurrentMark. In Figure 6 a more complex example is shown.

3.2 Translation of while

The translation of a while b do statement is similar to that for an if, but it requires some extra generated code. One of the problem again concerns the correct evaluation of expression b; also in this case the result of the evaluation of b is stored in a local variable while.exp_r in the transformed code and while will use the value of this variable as its expression. At the end of the body of while, b will be evaluated once again and while.exp_r will be updated with the new result.

Moreover the value of mark has to be reset to the value it had at the beginning of the while's body, otherwise it would be executed only once. During the translation we assume to have a stack data structure where we can push and pop values for CurrentMark. The translation is in Figure 7 (WhileMaxMark has the same meaning of IfMaxMark in the translation of if).

Also in this case, mark is assigned WhileMaxMark + 1 after exiting the while block, so that, upon resuming, that block will be totally skipped in case
Fig. 6: A translation of a procedure with some nested if’s. We assume that CB’s do not need translation.
the condition is not true anymore. Notice that the value of $CurrentMark$ is popped from the stack but it is only assigned to $mark$, not to $CurrentMark$. In Figure 8 an example with two nested while’s is presented.

4 Strong Mobility and X-KLAIM

The code transformation presented in this paper is exploited to obtain strong mobility in the X-KLAIM programming language for mobile code [3]. Some prototype applications of X-KLAIM exploiting strong mobility have already been described in [4]. The implementation of X-KLAIM can be found on-line at http://music.dei.unifi.it/xklaim.

The xklaim compiler traverses the syntax tree many times, and also transforms the tree in order to handle strong mobility. In particular it first detects the procedures that can migrate (possibly handling also mutual recursive procedures). These procedures are then examined to detect possible migration points, and to insert marker tags in points where the migration can take place ($Set$Label marker) and where the execution may restart ($Jump$Label marker). Then the code is transformed inserting if-blocks testing $mark$. An example of these phases is in Figure 9. Only at this point, after the code has been transformed, the target code is generated.

The target code of the current implementation is Java, but the transformation does not act on Java code, but only on X-KLAIM code. This is done because X-KLAIM is not thought for Java implementation only: indeed an implementation in Ada was also done [27]. In that case only the code generation part of the compiler has to be changed: the transformation patterns can be reused. Moreover if the target language supported goto statement, the last phase of the transformation could be changed to exploit the markers inserted by the previous phase to generate code with gotos, instead of if-blocks. An implementation in C# [17] of the run time system is under development, and the same transformation phases can be reused as well.
In our implementation weak mobility is completely handled in the Java package, KLAVA, for the run-time system of our language [2]; every procedure, called process in KLAVA, is abstracted in a class containing instance variables for procedure parameters, local variables and mark field. Saving and restoring the state of a procedure is done automatically by the system through serialization; moreover, through reflection the code for the agent and its procedures are sent to the remote site [3].

5 Concluding Remarks

In this paper we presented a purely syntactic translation process for translating strong mobility into weak mobility, while preserving the original semantics. This technique is currently exploited in our language for mobile agents, X-KLAIM, that provides linguistic constructs for strong mobility. However we would like to remark that our transformation is independent of the language itself. We plan to extend our patterns by taking into consideration exception handling. Moreover
Fig. 9: The X-KLAIM code transformed according to the translation presented: (a) the original code, (b) code after the marking, (c) transformed code.

we think that the patterns can be used to implement also other typical operations of mobile agents [15], such as clone (an agent clones itself and the two agents continue their execution from the same point, but each of them with their own data) and suspend (the agent suspends its execution, possibly saving its state on secondary memory, and upon resumption continues its execution). We shall experiment on this. We would like to conclude by comparing our work with other similar approaches.

The idea of migrating processes was already exploited in the 1980s, in the LOCUS distributed operating system [29], for load balancing. Some other systems [9,18] use state saving techniques to provide transparent process migration or persistence functionalities (a survey of such techniques can be found in [23]). However, in these systems, the migration mechanisms are part of the operating system or of the run time system itself.

Telescript [30] was one of the first language specifically designed for mobile agents, and provided strong mobility; this was achieved inside the language interpreter; however Telescript does not provide migration of multiple threads. The successor of Telescript, Odyssey [11], which is implemented in Java, does not provide strong mobility transparently.

There are systems providing strong mobility in Java, such as [5,19,20], that modify the Java Virtual Machine, to access, save and restore the execution state of threads; however this solution puts at risk one of the most desirable advantages of Java: portability across platforms. Indeed one needs to run the modified version of the JVM in order to use such agents.
Other systems instead follow our approach of syntactic code transformation, but are specific to Java. In [14] Java methods are transformed in order to possibly return, upon migration, the local continuation; upon resuming the agent on the remote site, this continuation can be used to walk through the stack of the agent, in order to reach the correct point of execution. Method signatures are also transformed in order to accept such continuations. Groups of possibly migrating threads have to explicitly synchronize in order not to generate deadlocks and inconsistent states. A similar approach is used in [10,22].

In [26] the transformation is applied at byte-code level: the original code is preprocessed and some code is inserted in the generated byte-code that saves the runtime information when the program requests state saving and reestablishes the program’s runtime state on restart. Even if byte-code is directly modified, the execution state is still inaccessible, thus the inserted code simulates the resumption of the execution on the remote site. The problem of agents made up by several threads are solved by inserting remote references (like in Obliq [6]) in the threads when one of them migrates, implementing Distributed Tasks [7].

All these approaches are targeted to Java and are heavily based on its features (indeed, the translations of [10,22] make use of Java run-time exception to traverse the method call stack). Instead we have proposed a more general translation that abstracts from the language features; we only assume that the target language provides means to implement weak mobility. Java would then be just one of the languages to which our method can be applied.

In [32] a similar transformation is applied within the context of Messengers, a system for distributed applications and mobile agents. While the translation is not targeted to Java (Messengers’s mobile agents can be written by using C mixed with navigational and synchronization statements), yet it is still dependent on the language. Indeed, an executing agent can be preempted only at known points. Moreover the code is transformed so that a navigational statement can be executed only as the last statement of a function. All functions are numbered and for each of them, at compile time, the possible successors are computed, that will be selected at run time. This successor is indeed hard-coded in the transformed code, and this renders the code not modular; on the contrary the field caller in our translation is updated by the run time system (i.e. by the weak mobile language) and so it is not static. Another drawback of the transformation of [32] is that, in order to handle the continuations after navigational instructions, additional functions are introduced: if this happens within a conditional statement or a loop, it leads to code explosion.

Another major difference is that all these systems adopt the approach of reconstructing the call stack starting from the first method on the stack until the method that caused the migration is reached; we follow the opposite approach (i.e. the call stack is reconstructed by starting from the last called procedure), and this is more efficient, since not all the procedures have to be called again. This is also more coherent to the idea of resuming execution. Indeed we are not aiming at reconstructing the entire stack of the agent: we want to mimic the sequential execution, and make sure that the call stack is simulated.
Moreover we aim at minimizing the size of the generated program: compared to [22], a while instruction is transformed without duplicating possible nested while, and no additional if-blocks are used just to restore the values of local variables as in [10], since this can be done in the target language in a more efficient ways. Our transformation does not suffer of code explosion as in [32] either. While the approaches at byte code level may lead to improving execution performances, we prefer to promote generality; the transformation can be smoothly modified in order to use goto statements, in case the target language provide them. Finally, we would like to stress once again that we are not interested in full mobility, thus we are not concerned with synchronizing mobile agents made up of several threads (as in [7,10,14]): for us, each mobile agent is a single thread.

References

Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture

Torsten Illmann, Tilman Krueger, Frank Kargl, and Michael Weber

University of Ulm, Dep. of Media Computer Science, 89069 Ulm, Germany,
torsten.illmann@informatik.uni-ulm.de,
http://cia.informatik.uni-ulm.de/

Abstract. In this paper we describe a transparent migration of mobile agents in Java using the Java Platform Debugger Architecture (JPDA). The JPDA allows debuggers to access and modify runtime information of running Java applications. In the context of mobile agents, the JPDA can be used to capture and restore the state of a running program. Since JPDA does not support to set the program counter, we introduce two different solutions to solve this. We either slightly modify the virtual machine or instrument some byte code instructions. Finally we measure the produced overhead in code and time compared to normal execution and other approaches addressing this problem. Altogether, we show that developing Java-based mobile agents with a transparent migration can be performed nearly without changing the source code, the byte code or the interpreter.

1 Introduction

Agent technology is increasing more and more. Since society is moving steadily towards an information society, the need of personal assistants for searching, supporting in e-commerce transactions and communicating with others is increasing in the same way. Personal software agents are a paradigm that promises to support these needs. Nevertheless, establishing and spreading agent technology in the real world still requires some important aspects to be solved.

Mobile agents are a suitable paradigm especially for mobile and distributed computing. Considering this in combination with the desire of sophisticated and easy-to-use agent systems, the need for development of transparent mobile agents and, in consequence, transparent migration techniques is there.

By migration we mean the movement of an agent to another location in the network (e.g. computer) and transparent continuation at the point before the migration occurred. That means, code and state of the agent must be captured, transferred to and restored at the destination location.

Modern agent systems are mainly implemented in Java because of its features platform independency, dynamic class loading, security issues and object-orientation. Unfortunately, standard Java does not allow to access all the internal runtime information structures of an agent. There has already been done some research to establish transparent migration by modifying source code, byte
code and the interpreter. Since these changes are expensive and/or highly complicated, we introduce another solution where complicated transformations or modifications are not necessary. Using the Java Platform Debugger Architecture (JPDA), which is part of the virtual machine specification, runtime information may be accessed in debug mode. This can be used to perform a transparent migration.

In the following paper, we first introduce the project where this work was developed in. Next, we describe problems of transparent migration and classify our approach. In chapter 4 and 5, we describe in detail how transparent migration is realized using the JPDA. Chapter 6 describes other approaches of transparent migration in Java. In chapter 7 we measure the produced growth in code and execution time of our approach in comparison to others. Finally we discuss our approach in a conclusion and name future work in this area.

2 The CIA Project

The CIA project deals with the development of an infrastructure for personal software agents. The system is called Collaboration and Coordination Infrastructure for Personal Agents [16] and is totally implemented in Java. It combines technologies like Java Messaging (JMS), Jini, Java Enterprise Beans, RMI and applets.

Basically the CIA System consists of three layers: The agent layer, the directory, broker and trading layer (DBT) and the service layer.

The agent layer defines programming models for static and mobile agents as well as topic-based inter-agent communication primitives. As communication models, one may choose among synchronous or asynchronous and uni- or multicast communication. Agents belonging to one user (or organization) are clustered in a so called agent cluster [18]. Clusters may be easily spread on workstations, portable computers or PDAs. Permanent connections and temporary connections may be handled in one cluster.

The DBT uses the Jini technology [15] to automatically and spontaneously combine different clusters and services without configuration. It assists agent clusters to find and trade with other clusters or external services.

The service layer allows to integrate external services of different kinds of technologies and from different locations into one service platform. For example it is possible to integrate RMI, EJB or CORBA services. With introducing the service layer, services are easily and transparently accessible for agents.

Whereas the design is modular and open, the current development mainly focuses the following topics:

- Communication transparency by using a software bus
- Network transparency with ad-hoc networking mechanisms
- Device- and location transparency of agents’ user interfaces
- Integration of external services in the agents’ infrastructure
- Migration transparency of mobile agents

In this paper, we concentrate on transparent migration of Java-based mobile agents.
3 Problems of Migrating Java Programs

To perform a migration of a mobile agent, one has to consider many aspects. Transparent migration in the view of an agent programmer means that he does not have to care about how to migrate. He specifies a migration statement (often called goor move) anywhere in his agent’s code and expects that the agent system manages the whole migration. After migration, code, private data and execution state are the same as before. The only difference should be that the agent resides on a different host.

Figure 1 shows a classification of problems that have to be taken into account when realizing a transparent (or strong) migration [17].

![Diagram of migration types](image)

**Fig. 1.** Problems of transparent migration

At the top level, the classification consists of two aspects: code and state migration. These refer to code transfer and state transfer of the agent. The state migration is composed of more aspects. On the second level, it is execution and data migration which means that the state of an agent is generally made up of the current execution point and the current data of the agent. Execution and data migrations are again made up of further parts. The execution migration is composed of the program counter and multi-threaded migration, Stack, member, resource and user interface migration define the data migration.

Whereas code migration and member migration can be implemented using the standard Java mechanisms dynamic classloading [14] and serialization [10], the others (program counter, multi-threaded, stack, resource and user interface migration) are not supported in the standard.

**Program counter migration** means that the execution at the destination location continues at the same point where it was interrupted before. We assign the re-establishment of the correct calling order of all nested executed methods and their inner code locations to this kind of migration. Generally, there are two peculiarities of this kind of migration: self-migration and forced
migration. To develop autonomous and self-responsible mobile agents, self-migration is required. Forced migration is rather needed in load balancing systems than in mobile agent ones.

**Multi-threaded migration** means the migration of all threads of a multi-threaded agent. That means, threads that the agent has created during this lifetime have to migrate as well. The currently running thread which requests the migration is in a well-known execution state (namely at the point of calling the migrate statement). When interrupting the other threads belonging to the agent, they may either be blocked or at any unknown point of their execution. We call the problem of capturing and restoring the state of all dedicated threads of an agent in the right order, the multi-threaded migration.

**Stack migration** implies the migration of all local data in every method on the call stack. That data consists of all values of local variables (variable stack) and operands of computations being on the stack (operand stack) up to the point of interruption. Stack migration depends on the program counter migration. It is not achievable without.

**Resource migration** addresses the problem of migrating open connections to system resources the agent is connected to. System resources are for example network connections, files or databases. In most reasonable mobile agent scenarios, an agent does not have open connections any more. To achieve total transparency, this problem has to be considered as well.

**User interface migration** means the problem of migrating the state of the user interface of agents. Because agents are mainly working invisible or the user interface is not bound to the agents location like in our system, this problem does not arise often. Nevertheless, this task also belongs to a transparent migration.

In this paper we focus on the problem to migrate the program counter and the stack. So we assume that the agent is single-threaded, is not connected to open system resources and has no open user interface.

4 Stack Migration Using Java Platform Debugger Architecture

Whereas other projects [1] [2] [3] [4] [5] solve the problem of transparently migrating a thread by modifying source code, byte code or virtual machine, we use the Java Platform Debugger Architecture (JPDA) [13] to perform this task.

JPDA is part of the virtual machine specification [11] (that means implemented by every standard virtual machine) and normally used to develop (remote) debuggers for Java applications. JPDA gives access to runtime information like running threads, their call stack and their program counter. It is possible to suspend and resume execution, execute single byte code instructions and set/unset breakpoints at arbitrary points. Furthermore, it enables programs to define event handlers that are called when methods are being entered or exited.

The JPDA is made up of two parts: The Java Virtual Machine Debugger Interface (JVMDI) and the Java Debugger Interface (JDI). JVMDI is a native
implementation and JDI a Java API built on top of it in order to access the
debugger functionality. Since JDI is implemented in Java, all code to access
runtime information can be implemented in pure Java. For example, reading
stack frames (their local variables) of all currently invoked methods – starting
from the deepest one – shows up in JDI as follows:

**Code example 1:**

```java
1 import com.sun.jdi.*;
2 // The program can connect to the virtual machine of the
3 // running thread via JDI
4 VirtualMachine vm = ...
5 List threads = vm.allThreads();
6
7 // index (or name) of current thread must be known
8 int current = ...
9 ThreadReference thread = (ThreadReference) list.get(current);
10
11 // read all stack frames (starting from last one)
12 List frames = thread.frames();
13 for (Iterator i = frames.iterator(); i.hasNext(); ) {
14   StackFrame frame = (StackFrame) i.next();
15
16   // read all local variables and output their values
17   List locals = frame.visibleVariables();
18   for (Iterator j = locals.iterator(); j.hasNext(); ) {
19     LocalVariable var = (LocalVariable) j.next();
20     System.out.println(var.typeName() + " \\
21         var.name() = " + var.getValue());
22   }
23 }
```

Using the above functionality, it is simple to store all stack frames in the case
an agent requests to migrate. The agent activates the debugger who suspends
the agent thread, stores all values of local variables of all stack frames in a
serializable variable called `stack`. This variable is for example implemented as
member field in the mobile agent’s base class. Consequently, stack frames are
automatically transmitted with the serialized agent. To achieve this behavior,
the above algorithm has to be changed from line 22 to 24.

**Code example 2:**

```java
22   Value value = frame.getValue(var);
23   stack.push(value);
```

The restoration of stack frames is nearly as simple. The `StackFrame` class
allows to set local variables using the `setValue` method. Since the values of all
local variables are popped from the serialized stack in opposite order, the local
variables have to be iterated in reverse order as shown below (change line 20 to
25 of code example 1):
Code example 3:

```java
for (int i=locals.size(); i <= 0; i--) {
    LocalVariable var = (LocalVariable) locals.get(i);
    Value value = (Value) stack.pop();
    frame.setValue(var, value);
}
```

Unfortunately, JDI does not provide interfaces to access the operand stack of a Java thread. We requested this feature to be integrated within future JDI specifications. Nevertheless, one may achieve a migration without transmitting the operand stack by either defining a simple programming convention for agent programmers. To avoid this problem, one may not specify nested calls of methods that request to migrate without storing intermediate results in local variables. The following example illustrates this convention.

Code example 4:

```java
int result = method1() + method2();
int temp  = method1();
int result = temp + method2();
```

In line one of the upper sample, the method `method1` is called first and its result is being pushed on the stack as operand for the later addition. If method `method2` requests a migration, this operand still has to be on the stack after restoration to perform a correct addition. Using the lower modification (line 01 and 02) where the intermediate result are stored in a temporary local variable, this problem does not arise any more. Certainly, this convention can be automated by slightly instrumenting the agent’s byte code.

The restoration of local variable values must be performed for all methods being called in the moment the migration is being initiated. This method invocation list is restored using the program counter migration described in the following chapter.

5 Program Counter Migration

The previous chapter describes the possibility to migrate the stack of a mobile agent’s thread. To perform a transparent migration, we further need to migrate the program counter. In more detail, that means that the sequence of nested invoked methods that are in execution right before migration and the locations within each method have to be captured, transferred and re-established.

Standard Java does not allow to access the program counter of the currently executing methods. Again in debug mode, the JPDA allows to access the location within a stack frame of a method. The value of the location identifies the byte code index relative to the start instruction of the method. Storing locations of all methods of the agent’s thread, makes it possible to capture the program counter of this thread. Code example 5 illustrates this:
Code example 5:

```java
for(Iterator i=frames.iterator(); i.hasNext(); ) {
    StackFrame frame = (StackFrame) i.next();
    ...
    // store local variables' values
    // store program counter
    long pc = frame.location().codeIndex();
    stack.add(pc);
}
```

The methods' program counters are also pushed on the stack to be transmitted. Since the program counter of a method is required earlier than values of local variables during the restoration process (see below), we push it on the stack after pushing local variables.

Re-establishing the program counter could have been in analogy to capturing them. An event handler is registered at the JPDA framework to be called whenever a method in the agent's thread is entered. The agent's thread is suspended if an entry event occurs. In this handler, we execute the code to restore the program counter and the previous mentioned local variables of this method. At last, we continue the execution. The following code fragment shows the simplicity of this event handler:

Code example 6:

```java
public void methodEntered(MethodEntryEvent e) {
    e.disable();
    StackFrame frame = e.thread().frame(0);
    long pc = stack.pop();
    Method method = frame.location().method();
    Location newLocation = method.locationOfCodeIndex(pc);
    frame.setLocation(newLocation); // does not exist!
    // set local variables ...
    e.enable();
    e.thread().resume();
}
```

The algorithm terminates when the migrate method of the agent framework is being entered. Unfortunately, the StackFrame class does not support a method to set the location (line 7). If there was such a method, the total migration functionality could be done in JPDA. No modification or transformation of source code, byte code or virtual machine would be needed.

Because of the lack of this method, we realize two small alternative extensions to support this functionality: we sightly change the virtual machine or the byte code of the agent.

5.1 Changing the Hotspot Virtual Machine

One alternative to set the program counter is to modify the virtual machine. Since the standard virtual machine of Sun Microsystems [12] is open-source, we
examined the source code for adding this functionality. Within the implementation of the Java Virtual Machine Debugger Interface (JVMDI), which is the native part of JPDA, it is possible to access runtime information of current executing threads on C++ basis. The JDI described earlier is built on top this native implementation. In there, the following method headers are declared:

```c
jvmdiiError GetCurrentFrame(jthread *thread, jFrameID *frame);
jvmdiiError SetCurrentFrame(jthread *thread, jFrameID *frame,
    jlocation *location);
```

The implementation of `GetCurrentFrame` enables to access the frame location within a running thread. The function `SetCurrentFrame` should allow to set the frame location. Surprisingly, this function is declared but has never been implemented. It seems to us that Sun planned to implement it and either forgot to do it or, since debuggers normally do not need it, forgot to remove this declaration.

In addition, there is an intern method `frame:interpreter:frame_setCurrentLocation` which sets the location of a stack frame by specifying the byte code instruction index (BCI). We add the implementation of `SetFrameLocation` using this method (about 20 lines of code) and finally make it accessible using a JNI interface. Whenever a program counter restoration is requested, this method is called with the byte code index being de-serialized from the stack.

We propose to integrate this patch in the virtual machine specification and reference implementation. The current release version of Sun’s reference implementation included in the jdk1.3.1 is Hotspot 2.0 (client and server). We further suggest to integrate the function to set frame locations in the Java Debugger API (JDI). If this patch gets accepted, a transparent migration is feasible in pure Java without modification of any line of code.

The added code is located in the platform-independent `share` part of the Hotspot’s source code. That means that all changes are portable for other operating systems. We already re-compiled the Hotspot 2.0 for Linux and Win32.

### 5.2 Instrumenting Byte Code

Another alternative to re-establish the program counter is the instrumentation of byte code instructions. Only few instructions enable the functionality of setting the program counter. First, the minimal set of byte code instructions to be instrumented have to be identified. Studying the virtual machine specification [11] in detail, we found out three small modifications that achieve this goal:

1. A program counter must be specified as local variable in each method.
2. A branch statement must be instrumented at the beginning of each method.
   It branches in dependency on the program counter to all locations where the execution may continue after a migration has been occurred.
3. Before invoking a method possibly performing a migration, the program counter has to be incremented.

In order to determine which methods are to be instrumented and which locations needs to be branched to, the following code example is examined:
Code example 7:

```java
public void calculate() {
    System.out.println("Do migration ...");
    migrate();
    System.out.println("Do migration recursively ...");
    calculate();
}
```

The migration is initiated directly in line 3. In line 5 the method is called recursively and thus initiates further migrations (for simplicity, this method never ends). To determine the methods being instrumented we first have to find out all methods that call `migrate` directly. Then, all parent methods calling them have to be retrieved iteratively using a bottom up search algorithm. We call these methods migratory methods. All migratory methods have to be instrumented by a program counter and a branch instruction that branches to all invocations of other migratory methods within this method. Looking at the byte code of method `calculate` of the previous code example, is transformation is explained:

Code example 8:

```java
getstatic java.lang.System.out Ljava/io/PrintStream;
ldc "Do migration ..."
invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
aload_0
invokevirtual Agent.migrate ()V
getstatic java.lang.System.out Ljava/io/PrintStream;
ldc "Do migration recursively ...
```

The migratory methods are `migrate` and `calculate`. They are invoked using the `invokevirtual` instruction at locations 9 and 21. The instrumentation of byte code as described above produces the following result:

Code example 9:

```java
iconst_0
istore_1
iload_1	
tableswitch default = 24, low = 1, high = 2(34, 47)
getstatic java.lang.System.out Ljava/io/PrintStream;
ldc "Do migration ...
invokevirtual java.io.PrintStream.println (Ljava/lang/String;)
iconst_1
istore_1
invokevirtual Agent.migrate ()V
getstatic java.lang.System.out Ljava/io/PrintStream;
```
40: ldc "Migration done."
42: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
45: iconst_2
46: istore_1
47: invokevirtual Agent.calculate ()V
50: return

We inserted four byte code instructions at the beginning of the method. The original start instruction moved to location 24. From location 0 to 1, we defined the program counter as local variable and initialized it to 0 for normal execution. From location 2 to 3, we branch according to its value to either the original start location (24) or all invocations of migratory methods (34 and 47). Whenever a migration method is invoked, an incrementation of the program counter is inserted before (32-33 and 45-46).

During normal or initial execution the same byte code than before transformation is executed, since the table switch statement (3) defaults to the original start instruction (24). In case of re-establishing the program counter after migration, the JPDA event handler of code example 6 is used. Instead of invoking the method setLocation in line 5, the following steps have to be performed:

1. Single step two byte code instructions until program counter is initialized.
   The execution is before location 2 now.
2. Set the program counter to the value popped from the stack via JDI.
3. Continue execution.

To implement these byte code modifications we use the Byte Code Engineering Library (BCEL) [8] developed at the University of Berlin by M. Damm. BCEL is open source [9] and provides a comfortable Java API in order to load, inspect, modify and save back byte code. Every byte code instruction and entity like classes, methods, fields and the constant pool is represented by a separate class. Creating instances of these instruction classes and adding them to the appropriate method instance easily modifies existing byte code. To instrument the functionality of initializing the program counter and branching to different locations, the following calls of BCEL have been used:

**Code example 10:**

```java
1  instructions.insert(new TABLESWITCH(pcValues, destinations,  
2    instructions.getStart()));
3  instructions.insert(new ILOAD(pc));
4  instructions.insert(new ISTORE(pc));
5  instructions.insert(new ICONST(0));
```

*Instructions* is the list of all instructions of a certain method. Instructions are inserted in opposite order (line 5 to 1) using the *insert* method. The branch instruction (line 1) needs all possible values of the program counter *pcValues*, the default branch location *instructions.getStart()* and all destination locations *destinations* where calls to migratory methods occur.

This byte code transformation is executed in a self-developed classloader [14] which transforms mobile agents’ code once after loading their original byte code. Therefore, the modification never is persistently visible.
6 Related Work

There are several projects that implemented transparent strong migration as well. In the following, we describe these projects and state differences compared to our approach.

**Wasp** [1]. The WASP project being developed at the University of Darmstadt aims at agents being integrated in web servers. The system implements transparent migration using a source code transformation (pre-compiler). The transformation inserts code to to store and restore the state of the agents. The program counter is implemented by adding conditional branches to the migratory methods. The capturing of the agent’s stack is done within exception handlers. When the agent requests to migrate, an exception is thrown and in every called method an exception handler is instrumented with stores all local variables within a stack variable. Re-establishing the stack is done by setting the local variables after each invocation of methods. Unfortunately, the source code is modified in many more ways to cover all possible cases. After the transformation the source code is unreadable and fully blown-up. The agent class has to be recompiled before execution.

**Sirac** [5]. In Sirac project, the standard Java virtual machine was modified for thread mobility and persistence. It also extends the standard Java API by introducing lower level methods to capture and restore threads. A higher level API provides primitives to perform thread mobility or thread persistence. In addition to self-initiated thread mobility, they also allow to force the migration by other threads. This functionality is for example needed to develop load-balancing or automatic activation/deactivation systems. Its application context is not focused on mobile agent systems. So far, the modified virtual machine is only available for JDK1.2.2.

**Nomads** [2]. The Nomad system implements their own virtual machine for performing transparent migration. They further realize special functionalities within this machine called fine-grained resource control. Connections to resources (e.g., CPU, network connections, file connections) can be restricted by passing parameters to the virtual machine.

**Brakes** [4]. The Brakes approach realize migration transparency by instrumenting all necessary functionality, i.e., stack and program counter migration, in the byte code via a post-compiler. For multi-threaded environments they define their own threading framework: *Tasks* have to be used instead of threads and a separate scheduler has been implemented. The framework supports cooperative multitasking only. After every task change (which is initiated by the task itself), the task is being serialized and the next activated. If a migration is requested, all tasks instead of the running one are already in a serialized form and may be transferred to the other location.

**JavaGoX** [3]. The approach of the JavaGoX system is quite similar to the one of Brakes. They instrument the byte code also by using a post-compiler. As far as we know, it does not implement support for multi-threading environments so far.
Many concepts and systems concerning thread mobility and persistence of native (non Java-based) applications for homogenous and heterogeneous platforms have already been developed and implemented. Information about these may be taken from [6] or [7].

7 Measurements

To underline our results and rank them in contrast to other approaches, we made two kinds of measurements:

1. The growth of byte code after instrumentation.
2. The execution time overhead when executing highly recursive methods.

7.1 Growth of Byte Code after Instrumentation

We measured the growth in byte code of three Java programs:

- A simple agent with only one migration call and one normal statement
- the Fibonacci algorithm
- a complex agent performing 10 migration statements and 50 normal statements

<table>
<thead>
<tr>
<th>Approach</th>
<th>Simple Agent</th>
<th>Fibonacci Program</th>
<th>Complex Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sirac</td>
<td>+0%</td>
<td>+0%</td>
<td>+0%</td>
</tr>
<tr>
<td>JPDA &amp; Hotspot-Modification</td>
<td>+4%</td>
<td>+4%</td>
<td>+4%</td>
</tr>
<tr>
<td>JPDA &amp; Byte Code Transformation</td>
<td>+10%</td>
<td>+5%</td>
<td>+3%</td>
</tr>
<tr>
<td>Brakes</td>
<td>+46%</td>
<td>+42%</td>
<td>+25%</td>
</tr>
<tr>
<td>JavaGoX</td>
<td>+114%</td>
<td>+61%</td>
<td>+102%</td>
</tr>
</tbody>
</table>

Table 1. Comparison of relative growth of byte code size

Whereas other projects that modify the byte code produce a high overhead (between 25% and 100%), our byte code instrumentation approach produces only between 3% and 10%. The more normal code (not specifying migrations) the agent contains the less overhead the approach produces. The modification of the Hotspot machine (Sirac and our approach) certainly produces no overhead in byte code size because no instructions are inserted.

7.2 Execution Time Overhead

We use a highly recursive function (the Fibonacci algorithm) to measure execution time and compare results to other approaches. We executed the tests on a PII-660 workstation with WinME operating system and JDK 1.3.1. Here are the results:
Table 2. Comparison of execution efficiency

<table>
<thead>
<tr>
<th>Approach</th>
<th>rf(35)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brakes parallel</td>
<td>+3700%</td>
</tr>
<tr>
<td>JPDA &amp; Hotspot-Modification</td>
<td>+810%</td>
</tr>
<tr>
<td>JPDA &amp; Byte Code Transformation</td>
<td>+814%</td>
</tr>
<tr>
<td>JPDA only</td>
<td>+758%</td>
</tr>
<tr>
<td>JavaGoX</td>
<td>+56%</td>
</tr>
<tr>
<td>Brakes serial</td>
<td>+28%</td>
</tr>
<tr>
<td>Java only</td>
<td>+0%</td>
</tr>
</tbody>
</table>

As we can see from the results, we get one total outlier. The brakes implementation with multi-threaded migration slows immensely down, probably because thread serialization is invoked very often. Since the other tested implementations only consider single-threaded agents, we do not further rate this outlier.

The other results show that the execution in debug mode already produces an overhead of more than 750% even if no migration functionality is used. Our approaches including migration functionality are about 6% slower than the one in debug mode without. In consequence, the main overhead is the requirement that the agent has to be executed in debug mode. In debug mode, the JIT compiler is always disabled. The other results of Brakes serial and JavaGoX use the JIT compiler. The SIRAC approach is not included in this evaluation since it is only accessible for JDK 1.2.2.

Looking at these results, a useful feature of the JPDA would be to partly switch on/off the debug mode. Then, the normal execution of an agent where no debug access is needed, could be executed using the JIT as well and produce better results. Furthermore, the discussion to integrate debugger functionality in the JIT has already started in the mailing lists. This clearly would improve our results immensely as our measurements show.

8 Conclusion

In this paper we presented two mechanisms for transparent stack and program counter migration of a Java thread using the Java Debugger Interface (JDI). Since JDI provides access to runtime information like stack frames, local variables and the program counter, the state of a Java program can be captured without modifying source, byte or virtual machine code. Nevertheless, there are two drawbacks to perform a transparent strong migration. On the one hand, the operand stack is not accessible from JPDA yet and, on the other hand, the functionality of setting the program counter is unexplainably missing. For avoiding the problem of the operand stack we propose either a simple programming convention for migratory methods or the use of a pre- or post processor which eliminates these problems [1]. To re-establish the program counter, we propose two alternatives to solve this problem. On the one hand, we slightly modify the standard Hotspot virtual machine to perform this task. On the other hand, we slightly instrument the byte code of the agent to set the program counter.
Algorithms and code examples showing the use of JDI and our suggestion to set the program counter are presented.

Since JDPA is part of the Virtual Machine Specification and our changes to set the program counter are either at byte code level or at the portable parts of the virtual machine’s source, this approach is totally portable to other operating systems and different versions of JDK.

Measurements of the growth in byte code show that our approaches add – depending on the considered alternative – no or only little instructions to the agent’s code. But they also show that the execution in debug mode is much slower than without.

9 Future Work

In the nearby future, we will try to convince Sun to integrate our small extension to set the program counter in the virtual machine and the JPDA specification. A transparent migration of stack and program counter would then completely be possible using the JPDA.

We will think about an extension of the virtual machine to temporarily suspend the debug mode to activate the JIT compiler and resume it later. Only the process of capturing and restoration the agent’s thread would then be slowed down. As long as no migration is requested, the agent could run with usual performance.

Furthermore, we will concentrate our research in other, often disregarded migration aspects, like multi-threaded migration, resource migration and user interface migration.

Our long-term goal is to develop an open source Migration API. Agent system developers or other developers requiring thread mobility or persistence in Java should be able use it in their own systems in order to enable migration techniques. Like a construction kit, it could be possible to choose the desired level of migration transparency and the desired implementation.

References

Portable Resource Reification in Java-Based Mobile Agent Systems

Alex Villazon\textsuperscript{1} and Walter Binder\textsuperscript{2}

\textsuperscript{1} University of Geneva, Switzerland
villazon@uii.unige.ch
\textsuperscript{2} CoCo Software Engineering GmbH, Austria
w.binder@coco.co.at

Abstract. Resource awareness is an important step towards the realization of adaptable software, something which is particularly desirable in the context of mobile code and mobile agent environments. Since resources (CPU, memory, network bandwidth, etc.) are not available and manipulable as first-class entities in standard programming models, such as in the Java language, some kind of reification seems indispensable. This is however difficult to achieve, especially if portability is a requirement. In this paper we describe a mobile agent execution environment that reifies several aspects of both the execution environment itself and of the mobile agents it hosts. We explain how resources consumed by an agent are reified directly from the agent code. Performance measurements show that our approach incurs only moderate overhead.

1 Introduction

Resource awareness in the context of mobile agents has been identified as an important concept for agent adaptability. If a mobile agent is aware of its resource consumption, it may use this information e.g. to optimize its migration decisions. Furthermore, a mobile agent platform that executes unknown foreign code has to control resource allocation, i.e., the system has to account the resources consumed by an agent and to prohibit allocations exceeding the agent’s resource limits, in order to prevent denial-of-service attacks caused by malicious (or buggy) agents, which may even crash the agent execution platform [21, 4]. Information about resource consumption may be used to implement different control algorithms (e.g., market-based [20,6], energy-based [2], applying different scheduling policies [4], etc.). Moreover, resource accounting and control may be targeted towards provision of quality of service or of usage-based billing, in order to amortize investments in hardware and software set at customers’ disposal.

These considerations raise questions concerning the manipulation of resource-related information and the programmability of agents, since resource-related aspects are clearly non-functional, i.e., frequently these aspects are not directly related to the basic task of the agent, and therefore it is important to separate them from the base-level code of the agent. Another important issue is how resource consumption can be reified, i.e., how it can be made accessible for
manipulation. Unfortunately, most mobile agent execution environments do not provide any means for obtaining information regarding the resource consumption of different agents. As a remedy, we suggest to integrate reflective capabilities into the execution environment (EE). Reflection and reification are closely related concepts, since a reflective system requires the reification of some of its internals. That is, reflection is the capability of a system to reason about and act upon itself [15]. A reflective system is composed of a base-level, which is the part of the system reasoned about, and a meta-level, which has access to the reified information about the base-level.

Even though Java [12] is the predominant implementation language for mobile agent systems, it does not support resource accounting. Proposed solutions for resource control in Java are either incomplete, or rely on native code, on low-level resource control mechanisms offered by the underlying operating system, or on a modified Java Virtual Machine (JVM) [14]. Consequently, these systems are not well suited to be deployed in heterogeneous environments, such as the Internet, where a wide variety of different hardware platforms and operating systems has to be supported. Because portability is of paramount importance for the success of a mobile agent system, resource control facilities have to be provided on top of standard Java runtime systems.

Resource control has to cover physical resources, such as CPU, memory, and network bandwidth, as well as logical resources, such as threads, the number of agents, etc. Moreover, communication with agents or services is also subject to resource control policies, which may e.g. limit the communication bandwidth and the size of exchanged messages. The reification of physical resources poses some serious difficulties, as it should be based only on the information that can be obtained from the agent code itself, without resorting to any external functionalities, such as those provided by the operating system. Thus, one of our goals is to provide an abstract and portable representation of the physical resources mentioned above, as well as mechanisms allowing to manipulate them without relying on functionalities specific to a particular operating system.

In addition to this, our approach allows to fully exploit all advantages of mobile code, since the reification itself may be performed by mobile code. That is, special code is injected into the mobile agent platform in order to customize the reification process. This is achieved by allowing agents to interact with reflective components inside the EE, rather than only with an external interface of the reflective system.

This paper is organized as follows: In section 2 we discuss related work on reflection and resource control in mobile agent environments. In section 3 we describe the generic architecture of a mobile agent platform, which enables the reification of physical and logical resources, as well as of communication structures. In section 4 we explain some basic ideas for resource reification in Java. We focus on physical resources and give an overview of our techniques to transpar-

---

1 For an (incomplete) list of different mobile agent platforms see The Mobile Agent List at http://www.informatik.uni-stuttgart.de/ipvr/vis/projekte/mole/mal/mal.html. Most of the systems presented there are based on Java.
ently reify memory and CPU resources by directly inspecting the code of mobile agents. In section 5 we present benchmarking results of our fully portable techniques for resource reification in Java. The last section concludes this paper.

2 Related Work

We distinguish two broad categories of related work: Proposals which apply reflective techniques to mobile agents, and systems which support resource control and may be used to implement mobile agent platforms. Even though resource control is beneficial for all kinds of programming environments, we focus on Java technology, because it is the common implementation language for mobile agent systems.

2.1 Reflection in Mobile Agent Environments

Some ideas about applying reflection in the context of mobile agents have been sketched by Ledoux et al. [13] and by Watanabe et al. [23].

Ledoux et al. suggest to use reflection in order to reason about and act upon the agent’s transfer mechanism. They point out that reification of resources, as well as of relationships between agents and resources, is crucial for the realization of an open architecture. In other words, mobile agents should be aware of the underlying infrastructure. In their approach reflection is exploited to allow different granularities in the migration process, and therefore to elaborate a fine-tuned model for code mobility. Reflection is thus placed inside the execution environment and not at the level of mobile agents.

Watanabe et al. take a different approach, placing reflection at the mobile agent level. This approach focuses on a fault-handling mechanism by defining meta-agents that can customize the base-agents’ fault-handling strategies (e.g., by introducing user-defined before- and after-fault handling methods at the meta-level of each agent).

Applying reflection to mobile code technology is therefore recognized as an interesting approach to improve openness. Regarding the combination of reflection and mobile code, there are two complementary issues:

- Tuning of internal aspects of the EE, such as mobility and communication, through reflection and reification of internals of the EE.
- Introducing new strategies to handle specific aspects of agent execution, through reflection and reification of agents.

The approach presented in this paper takes advantage of these two aspects: Reflection is used at both the execution environment and mobile agent levels. This requires a particular infrastructure, which we call a reflective EE (REE), supporting reification of internals of the EE, as well as reflection of the agents. The REE enables the execution of mobile agents as reflective entities that interact with the reified internals of the environment. The REE allows mobile agents
to influence their own execution or the execution of other agents, and to interact with internals of the REE.

Kava [24] is a ‘reflective Java’, which is based on load-time bytecode rewriting and supports the adaptation of applications. Kava has been used to specify and implement security policies for mobile code [25]. Reflection is used to insert security checks into the compiled application, avoiding the need to re-compile the application when different security checks are required. The security mechanism is implemented by *meta-objects*, i.e., special objects containing reflective information [15] that act as reference monitors and enforce security policies. Meta-objects are part of the trusted computing base and can be securely loaded from a remote source. Kava also supports some limited forms of resource accounting, it enforces a limit on the number of threads an application may create. A meta-object acts as a resource monitor and throws an exception, if the thread limit is exceeded.

2.2 Resource Control in Java Environments

JRes [10] is a resource control library for Java, which takes CPU, memory, and network resource consumption into account. Accounting for CPU relies on native code and on the underlying operating system\(^2\). Memory accounting in JRes is closely related to the reification of memory resources presented in this paper, although JRes still needs the support of a native method (to account for memory occupied by array objects). To achieve accounting of network bandwidth, the authors of JRes also resort to native code, since they swapped the standard java.net package with their own version of it. Consequently, JRes does not meet our requirements regarding portability.

KaffeOS [1] is a Java runtime system based on a modified JVM. It supports the operating system abstraction of *process* to isolate applications or mobile agents from each other, as if they were run on their own JVM. Thanks to KaffeOS, it is possible to achieve resource control with a higher precision than what is possible with our portable techniques for resource reification. The KaffeOS approach should result in better performance by design, but is however inherently non-portable.

NOMADS [17] is a mobile agent system which has the ability to control resources used by agents, including protection against denial-of-service attacks. The NOMADS execution environment is based on a modified JVM, the Aroma VM, a copy of which is instantiated for each agent. There is no resource control model or API in NOMADS; resources are managed manually (on a per-agent basis) and the resource related information is not accessible to agent. Since NOMADS is based on a modified JVM, its portability is limited.

\(^2\) More precisely, CPU accounting in JRes is based on native threads, a feature not supported by every JVM.
3 Reflective Execution Environment (REE)

Reflection is the capability of a computation system to reason about and act upon itself and to adjust itself to changing conditions [15]. Reflective systems provide more openness than traditional systems, since they allow inspection and modification of internal functionalities. Reflective systems require the reification of some aspects of the base-level computational system. In other words, reification makes something accessible, which normally is not available in the programming model.

Mobile agents offer high flexibility for application deployment, dynamic application extensibility, and configurability. However, frequently the corresponding EE provides insufficient means for manipulating internal functionalities, and hence limits the resulting software adaptability. The application of reflection to mobile agents aims at providing enhanced adaptability and flexibility for the implementation of agent applications.

Fig. 1 shows how it is possible to enhance adaptability using reflection and mobile code. The application of mobile code usually improves adaptability w.r.t. a classical application, since parts of the service can be implemented as mobile components, which are pushed into the system dynamically. However, the adaptation of mobile components is normally performed based on external information requiring to stop the application, to modify the code, and to deploy it again (see (a) and (b)). The reflective approach enables mobile components to perform the adaptation exploiting internal information about the EE and their own execution (see (c)), without any external configuration.

![Diagram](image)

**Fig. 1.** Adaptation through mobile code and reflection.

Thus, the design of a REE for mobile agents requires the following considerations: (1) which aspects of the EE shall be reified, and (2) which parts of incoming mobile agents have to be reified. These considerations help to define the architecture of the different elements inside the EE and their dependencies. In the case of reification of mobile agents, the role of the reflective part of the agents is also established. Thus, a reflective mobile agent EE has to:
- Reify internal aspects of the EE and of agents;
- Allow the manipulation of the reified information;
- Allow separation of concerns of orthogonal aspect in mobile agents.

The architecture of such a REE consists of two levels: the **base-level** and the **meta-level**. The first level is composed of components that handle the execution of *base-level agents*, whereas the second level supports the execution of *meta-agents* and the manipulation of the reified information. The base-level can be seen as a conventional mobile agent EE without any support for reflection, which acts as a *black-box* (i.e., a closed environment that gives minimal and ad-hoc access to internal details). The meta-level includes components that are related to the reified aspects of both agents and the EE itself.

Fig. 2 illustrates the conceptual architecture of the REE. The different components in both base- and meta-level are described in the following subsections.

![Diagram](image)

**Fig. 2.** The architecture of the Reflective Execution Environment.

### 3.1 Base-Level of the REE

The base-level provides the basic functionalities of the EE. It consists of the following elements:

**Communication:** The communication element handles communications between base-level agents and local services. It also supports communication with remote EEs or applications, i.e., it provides external connectivity.
Loader: This component is responsible for retrieving the code of mobile agents. The loader is connected to the external communication channel that receives agent code from remote EEs. The loader is logically divided into two parts: the loader at the base-level and the reflective loader at the meta-level. Both components are complementary, since the base-level loader is responsible for loading mobile agents and new services, while the reflective one performs the reification of the agent and associates the corresponding meta-agent (the reified information is made accessible through meta-objects). The combination of both loading components allows the creation of reflective mobile agents.

Agent context: This element is used to create the context in which the agent is executing and to trigger the agent’s initialization. The context provides access to the different services available in the EE.

Service repository: Local services, which may be accessed by mobile agents running in the EE, are stored in the service repository. Agents may trigger the installation of new services, which are loaded dynamically. The loader handles the loading and linking of services.

Security: The security component has to mediate the external communication with remote EEs or applications, as well as to enforce security policies during the loading of agent code. For example, this component may ensure that agents are loaded from a trusted remote EE or application, and verify that agents do not refer to forbidden objects or services (such as internals of the EE).

3.2 Meta-level of the REE

At the meta-level internal aspects of the EE are reified, i.e., the EE exhibits internal components to be manipulated at the meta-level. The meta-level is populated by meta-agents, which form the reflective part of mobile agents executing in the base-level. From a logical point of view, the combination of a base-level agent and its associated meta-agent may be considered a reflective mobile agent, which is able to think and act upon itself, to access its internal representation (code and state), and to change its behavior. Meta-agents are located at the meta-level and manipulate the reified information (about the EE internals and about the base-level agent) in order to adapt the base-agent. Meta-agents are located in the Agents’ meta-level and the reified internals of the EE reside in the EE’s meta-level (see Fig. 2).

Reification of EE Internals. The REE is a white-box, which allows its internals to be reified. This architecture is based on multi-model reflection, which allows the separation of concerns at the meta-level itself [16]. Considering the specificity of the mobile agent paradigm, three aspects have been identified that are necessary to provide openness in the REE: composition, communication, and resources:
Composition: The composition of different elements in the system is reified in order to expose the interconnections of base-level elements. For instance, this allows to inspect the binding of communication facilities to network interfaces, or to discover all services that employ the communication component.

Communication: The communications between different elements of the EE are reified in order to provide an entry point for the introduction of filters, to account for messages exchanged, or to redirect communication messages in the EE. Communication with local services are of particular interest and are closely related to the reification of agents.

Resources: Resources are reified in the REE. Physical resources (CPU, memory, network bandwidth, etc.) have abstract representations, since it is difficult to map low-level resources that are typically dependent on the underlying operating system. Logical resources (such as threads, agents, etc.) are also reified. The representation of reified logical resources allows to manipulate and to modify the way the EE allocates those resources (e.g., enforcing resource limits).

Similar architectures have been proposed in the context of reflective middleware [5,9], which provide hooks to add new behavior to the environment. In the case of mobile agents, this aspect is not necessary, since the agents themselves provide such functionality.

Reification of Mobile Agents. The REE allows mobile agents to be reified when they arrive. For each base-level agent, the REE associates a meta-agent, which is able to manipulate the reified information of the base-level agent. The reification of the agent is a process that takes an agent as input, reifies internal aspects of the agent, and binds the agent to a meta-agent. All these adaptations transform the agent into a new reflective mobile agent as shown in Fig. 3.

The meta-agents are compositions of several meta-objects that are related to the different reified aspects of the base-level agent. Similarly as for the EE, we have identified three aspects of the base-level agent that are reified: structure, bindings, and resources. The associated meta-objects play the role of micromanagers of the base-level agent. The separation of the meta-agent into domain-specific meta-objects allows to simplify the modification and composition of the different aspects handled at the meta-level.

Structural representation of the agent: The structural representation allows the meta-agent to adapt the base-level agent using the information about composition, communication, and resources, which is exposed by the REE. The reified agent structure allows to write special code to modify this structure. Structural reification involves a high-level representation of the agent code, which can be easily manipulated.

Bindings to services and other agents: On arrival, the base-level agent has unresolved references to other agents or services. By reifying these bindings, the meta-agent can adapt the interactions of the agent. The meta-agent may collect information about agent bindings and use it for optimizations, for accounting, to apply particular communication policies, and for debugging.
**Resources consumed by the agent:** The reification of the resources consumed by an agent exposes the agent’s resource consumption to the meta-agent, which may use this information for monitoring, to enforce resource limits, or for billing purpose.

The reified aspects of the EE and of agents are closely related. However, the manipulations supported at the agents’ meta-level are more flexible than those in the EE itself. The information associated with the reification in the EE’s meta-level is useful for the adaptation in the agents’ meta-level. Only in a restricted number of cases, modifications are performed in the EE’s meta-level, because the components of the EE are rather static when compared to the dynamic nature of mobile agents.

The reification and adaptations applied in a REE are transparent to the agent programmer, who does not need to consider the non-functional aspects, which are incorporated just before the agent starts executing, i.e. at load-time.

One disadvantage of load-time reification is the overhead caused by the reification process. For the reification of structure and bindings, the overhead is rather small, because the necessary information can be obtained without complex processing. However, resource reification may cause considerable overhead. We have studied these aspects in a concrete implementation and evaluated the overhead of resource reification, focusing on CPU and memory. Our results, which are presented in section 5, show that sophisticated implementation techniques keep the overhead due to resource reification reasonably small.

### 4 Reification of Resources

For agent resource reification, we analyze and modify the agent code in order to extract information related to resource consumption and to insert the meta-objects that collect and maintain this information. Modification of source code is a common practice in some reflective systems, since it allows to manipulate and adapt applications [18]. In a mobile agent context load-time transformation based on the (compiled) agent code is better adapted, because it does not depend
on the source code of agents (which usually is not available) and enables the necessary modifications to be applied before an agent starts executing. The REE allows reification and adaptation directly on agent arrival, avoiding the need to implement different versions of the agent for distinct EEs. The reification process is performed using special adaptation code (agent modifiers) that can be dynamically inserted into the REE. Using mobile code for the reification process itself further increases the flexibility of the model. As illustrated in Fig. 4, it is possible that an agent visits EEs that do not support any adaptations.

![Diagram](image)

Fig. 4. Overview of adaptations in REEs.

To illustrate the reification of resources in the REE, let us consider Java as the target language, because it is the common implementation language for mobile agent systems. Moreover, Java offers many features that ease the implementation of our REE, such as e.g. object-orientation, language safety, multi-threading, a portable code format (bytecode), as well as the support for dynamic and customized class-loading, which enables sophisticated adaptation of agent classes before they are loaded and linked by the JVM.

Recently, we have observed an increasing interest in applying reflection to Java-based environments. Frameworks such as Javassist [8] allow structural reflection of Java programs. Javassist reifies the whole structural representation of an application directly from the bytecode and allows the modification of the application structure at load-time, hiding the low-level aspects of bytecode manipulation.

Our approach is based on rewriting of Java bytecode, because it enables a fully portable solution that does not rely on any low-level operating system functionalities. In the following subsections we briefly explain some basic ideas for resource reification in Java environments. More details can be found in [4].
Reification of Network Bandwidth. For the reification of network bandwidth, a straightforward solution consists in redirecting the calls to the component that provides the network service and associating a meta-object to this call (see Fig. 5). This is done by before/after processing of the method call and by trapping of well-known methods. The network resource is reified by an object that maintains information on consumption, thresholds, etc. This solution exposes e.g. the network traffic caused by an agent, and it allows to set limits or to add some filtering of network messages at the meta-level. The necessary modifications at the bytecode level are rather simple, because method invocations are explicit in the agent code.

![Diagram](image)

**Fig. 5.** Reification of network bandwidth.

Reification of Memory. Memory reification is more complex, since it requires accounting of all object allocations. The difficulty comes from the fact that it is no longer sufficient to redirect a call to a given service. We have to modify the way objects are allocated and deallocated in the EE. Another difficulty is to accurately estimate the size of allocated objects and the size of agent code. We have to calculate an estimation of the size of objects that the agent creates and to provide wrappers for the construction of these objects. This also requires forcing the agent to use the wrappers.

Our approach consist in dynamically adding our own version of object allocator to the agent code and replacing all object allocation instructions and invocations of constructors with our reified version (see Fig. 6). The memory reification process is supported by agent modifier code, which is retrieved from a remote node and performs the adaptation of agent. This allows to implement different allocation strategies without having to hard-code the actual reification in the EE.

Object deallocation is also difficult to account for when memory is garbage collected, as in Java, because there are no explicit application-level operations that could be easily tracked to this end. Details can be found in [4].
**Reification of CPU.** CPU is probably the most difficult resource to be reified, because it is not explicitly ‘visible’ as a method invocation or an object allocation. Most operating systems provide some means for CPU control that may be exploited by an agent EE. However, such a solution limits portability and therefore should not be applied as a general-purpose solution in the context of a mobile agent system. Consequently, the REE provides some means in order to reify the CPU resource directly from the agents’ code. We introduce an abstract unit of measurement based on the *number of bytecode instructions* executed by an agent. The reified CPU resource is associated with a meta-object that monitors the CPU consumption of all threads of an agent.

On a JVM using Just-in-Time compilation, this approach only gives an estimation of the actual CPU consumption. Nevertheless, an approximation is sufficient for many purposes, such as for the prevention of denial-of-service attacks. On a JVM implemented in hardware, like recently emerging Java processors that offer competitive performance and low power consumption, accounting the number of executed bytecode instructions gives a precise information on the actual CPU usage. Furthermore, as such processors will be integrated in mobile devices, where preservation of battery power is of paramount importance, the information on CPU consumption may be used to estimate and limit the power consumption of mobile agents.

In our approach the bytecode of an agent is analyzed in order to build control-flow graphs of the agent’s methods (see Fig. 7). The resulting graphs are used to insert accounting instructions at strategic places into the bytecode. Thus, the reified accounting information is updated while the agent is executing. At the meta-level this information may be used to implement dedicated scheduling algorithms, which may e.g. reduce the priority of threads of an agent, if it exceeds its CPU limit.
5 Evaluation

We have implemented a tool based on bytecode rewriting techniques, which transforms Java classes in order to reify resource consumption. While our transformations for memory accounting are related to techniques used by JRes [10], a similar approach for CPU accounting in Java has not been used before. Our current implementation supports off-line transformations of arbitrary Java classes (including JDK classes).

We are also integrating resource reification and appropriate control mechanisms into the J-SEAL2 mobile agent kernel [3], which requires load-time rewriting of mobile objects. J-SEAL2 is a secure mobile agent system implemented in pure Java, which supports the hierarchical process model of the Seal Calculus [22] that was first implemented by the JavaSeal mobile agent system [7]. Resource reification in J-SEAL2 concerns only memory and CPU resources, since the J-SEAL2 design already supports network accounting and the integrating of application-specific security policies.

In this section we present performance measurements showing that the overhead due to our completely portable implementation of CPU and memory accounting is acceptable on modern JVM implementations. We are not measuring the overhead incurred by the utilization of the reified information\(^3\). Our goal is to show that resource reification causes only moderate overhead and opens up interesting possibilities to improve flexibility and adaptability by allowing the application to use the reified information.

Our bytecode rewriting tool is based on BCEL (Byte Code Engineering Library) [11], which allows bytecode manipulations of Java classes and is also entirely written in Java. We chose BCEL since it is one of the most mature bytecode

\(^3\) E.g. for CPU control, the overhead caused by a dedicated scheduler that uses the reified information can be kept small by choosing an appropriate time-slice.
instrumentation frameworks and provides a powerful and intuitive API that is well adapted for our requirements.

To show that our approach may be applied to complex Java applications (and also because there is a lack of standard benchmarks for mobile agent applications), we measured the standard SPEC JVM98 benchmarks [19] on a Linux platform (Intel Pentium III, 733MHz clock rate, 128MB RAM, Linux kernel 2.2.16) with IBM's JDK 1.3 implementation, which includes one of the best Just-in-Time compilers available today. We measured the overhead due to CPU and memory accounting in three different configurations:

- Unmodified benchmarks.
- Rewritten benchmarks for CPU reification.
- Rewritten benchmarks for memory reification.

For each measurement, table 1 shows the execution times of the benchmarks in seconds (rounded to 3 decimal places), as well as the speedup of the original code compared to the rewritten version (rounded to 2 decimal places). In order to minimize the impact of compilation and garbage collection, all results represent the median of 101 different measurements. Furthermore, we also computed the geometric mean for each configuration. We rewrote about 520 Java class-files for the CPU and memory-aware versions of the SPEC JVM98 benchmarks.

Table 1. Benchmarks measuring the overhead of CPU and memory accounting (time in seconds).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Unmodified</th>
<th>CPU reified</th>
<th>Mem reified</th>
</tr>
</thead>
<tbody>
<tr>
<td>.227.mtrt</td>
<td>5,823 (1.00)</td>
<td>7,336 (1.26)</td>
<td>6,898 (1.19)</td>
</tr>
<tr>
<td>.202.ess</td>
<td>7,779 (1.00)</td>
<td>9,145 (1.18)</td>
<td>8,608 (1.11)</td>
</tr>
<tr>
<td>.201.compress</td>
<td>19,130 (1.00)</td>
<td>23,156 (1.21)</td>
<td>19,500 (1.02)</td>
</tr>
<tr>
<td>.209.db</td>
<td>26,740 (1.00)</td>
<td>27,777 (1.04)</td>
<td>27,031 (1.01)</td>
</tr>
<tr>
<td>.222.mpegaudio</td>
<td>8,694 (1.00)</td>
<td>12,425 (1.43)</td>
<td>10,358 (1.19)</td>
</tr>
<tr>
<td>.228.jack</td>
<td>8,184 (1.00)</td>
<td>8,771 (1.07)</td>
<td>9,226 (1.13)</td>
</tr>
<tr>
<td>.213.javac</td>
<td>14,150 (1.00)</td>
<td>15,618 (1.10)</td>
<td>16,016 (1.13)</td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>11,286 (1.00)</td>
<td>13,296 (1.18)</td>
<td>12,508 (1.11)</td>
</tr>
</tbody>
</table>

The results in table 1 show that the overhead due to CPU accounting is about 20%, while in the case of memory accounting the observed overhead is only about 10%. Note that we did not apply any optimizations to reduce the accounting overhead. Simple optimization rules, as discussed in [4], can help to reduce the overhead significantly. The implementation of the optimization algorithm is still in progress.

4 The JDK was not rewritten for the measurements presented in this paper. See [4] for an evaluation of the performance impact of JDK rewriting.
6 Conclusion

In this paper we have presented the architecture of a reflective mobile agent EE supporting the reification of agents and of internal aspects of the EE itself. The reflective EE allows to manipulate information on resource consumption. We suggest an implementation scheme for Java, which is entirely portable and entails only moderate overhead. Moreover, our approach is not restricted to mobile agent applications, but opens up the perspective of building portable resource management policies as dynamic add-on modules for commercial off-the-shelf components.

Acknowledgments. Many thanks to Jarle Hulaas and Klaus Rapf for their valuable comments, and to Rory Vidal for his work on CPU reification. This work was financed by the Swiss National Foundation, grants 5003-057753 and 20-54014.98, and by CoCo Software Engineering GmbH.

References


Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task

Robert S. Gray\textsuperscript{1}, David Kotz\textsuperscript{1}, Ronald A. Peterson\textsuperscript{1}, Joyce Barton\textsuperscript{2}, Daria Chaoon\textsuperscript{2}, Peter Gerken\textsuperscript{2}, Martin Hofmann\textsuperscript{2}, Jeffrey Bradshaw\textsuperscript{3}, Maggie Breedy\textsuperscript{3}, Renia Jeffers\textsuperscript{3}, and Niranjan Suri\textsuperscript{3}

\textsuperscript{1} Dartmouth College Computer Science,\textsuperscript{\dagger} Hanover NH, USA, \{gray,dkz,rap\}@cs.dartmouth.edu
\textsuperscript{2} Lockheed-Martin Advanced Technology Laboratory,\textsuperscript{\ddagger} Camden NJ, USA \{jbarton,pgerken,mhofmann\}@at1.lnco.com, dachaoon@bigfoot.com
\textsuperscript{3} Institute for Human and Machine Cognition,\textsuperscript{\S} Univ. of West Florida (Pensacola) USA \{jbradshaw,mbreedy,rjeffers,nsuri\}@ai.uwf.edu

Abstract. Building applications with mobile agents often reduces the bandwidth required for the application, and improves performance. The cost is increased server workload. There are, however, few studies of the scalability of mobile-agent systems. We present scalability experiments that compare four mobile-agent platforms with a traditional client/server approach. The four mobile-agent platforms have similar behavior, but their absolute performance varies with underlying implementation choices. Our experiments demonstrate the complex interaction between environmental, application, and system parameters.

1 Introduction

One of the most attractive applications for mobile agents is distributed information retrieval, particularly in mobile-computing scenarios. By moving the code to the data, a mobile agent can reduce the latency of individual steps, avoid network transmission of intermediate data, continue work even in the presence of network disconnections, and complete the overall task much faster than a traditional client/server solution.

A common performance concern about mobile-agent systems, however, is that they shift much of the processing load from the clients to the server. This shift is a significant advantage in some environments: the clients may be hand-held...
computers with limited memory and computational power, and the “server” may be a large multiprocessor computer. On the other hand, the shift does raise questions about scalability. As the number of clients increases, how well do the mobile-agent services scale? Where is the trade-off between the savings in network transmission time and the possible extra time spent waiting for a clogged server CPU?

We set out to examine these questions. In the context of a simple information-retrieval application, we compared a traditional client/server (RPC) approach with a mobile-agent approach on four mobile-agent platforms. Our goal was to understand the performance effects that are fundamental to the mobile-agent idea, and separately, the performance effects due to implementation choices made by the different mobile-agent platforms.

We begin a comparison of the four mobile-agent systems we consider. Then we describe the scenario chosen for our tests, and the details of the tests themselves. We present the experimental results and our interpretation. Finally, we compare our results with the most relevant prior literature.

## 2 Mobile-Agent Systems

We evaluate four mobile-agent platforms: D’Agents [7,8] from Dartmouth College, EMAA [12,4] from Lockheed-Martin Advanced Technology Laboratory, K AoS [3,2] from Boeing and the University of West Florida Institute for Human & Machine Cognition (UWF-IHMC), and NOMADS [14] from the UWF-IHMC. We chose these systems because they were available to us and because they represent a range of design choices, yet share a common language (Java). Since a full presentation of these systems is outside the scope of this paper, Table 1 outlines the features most relevant to our experiments. Each feature represents a design decision made by the systems’ authors. We discuss the importance of these decisions here.

K AoS uses a hybrid approach allowing static agents to dispatch small task-specific agents called *mimons*. K AoS allows developers to plug in different mobility solutions. In this case, our experiments used Voyager 3.0 for K AoS mobility, so most performance effects are dependent upon Voyager’s implementation.

<table>
<thead>
<tr>
<th>Feature</th>
<th>D’Agents</th>
<th>NOMADS</th>
<th>EMAA</th>
<th>K AoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) strong/weak</td>
<td>strong</td>
<td>strong</td>
<td>weak</td>
<td>weak (Voyager)</td>
</tr>
<tr>
<td>b) JVM version</td>
<td>1.0.2</td>
<td>Aroma</td>
<td>1.2.2</td>
<td>1.3</td>
</tr>
<tr>
<td>c) JVMs used</td>
<td>multiple</td>
<td>multiple</td>
<td>one</td>
<td>one</td>
</tr>
<tr>
<td>d) what moved</td>
<td>all code, data, stack</td>
<td>data, stack</td>
<td>data</td>
<td>data</td>
</tr>
<tr>
<td>e) code caching</td>
<td>no</td>
<td>yes</td>
<td>preinstalled</td>
<td>preinstalled</td>
</tr>
<tr>
<td>f) encoding</td>
<td>custom, fat</td>
<td>custom</td>
<td>serialized</td>
<td>serialized</td>
</tr>
<tr>
<td>g) communication</td>
<td>sockets</td>
<td>sockets</td>
<td>sockets</td>
<td>sockets (Voyager)</td>
</tr>
<tr>
<td>h) socket reuse</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i) security</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>off</td>
</tr>
</tbody>
</table>
(a) D’Agents and NOMADS support strong mobility, where the agent’s control state, as well as its code and data state, is moved from one machine to another. As a result of this decision, they use different versions of Java. (b) D’Agents uses a modified version of an older Sun JVM, whereas NOMADS uses a custom JVM called Aroma (a “clean-room” implementation of the Java VM specification, and mostly JDK 1.2.2 compatible). This decision has a significant impact on performance, because the newer Sun JVM is generally more efficient and supports Just-In-Time (JIT) compilation. The NOMADS JVM is an un-tuned research prototype with no JIT compiler. Despite its age, the D’Agents JVM has one benefit: optimized string-handling routines that are important for our test application.

(c) For several reasons, D’Agents and NOMADS also create a new JVM process for each incoming agent, while the others simply add a new thread to the existing JVM. This choice raises the cost of jumps in D’Agents and NOMADS.

(d) Only D’Agents moved every bit of agent state (all necessary classes, the stack of the jumping thread, and the reachable parts of the heap) on every jump. NOMADS cached the code on the server during the first jump, so subsequent jumps did not need to move code. EMAA and KAoS do not transmit code with an agent; the recipient fetches the code from a class server and then caches it for future use. As a result, in our experiments they essentially never moved code. EMAA and KAoS never move thread-stack state. As a result, EMAA and KAoS agents are relatively small.

(f) EMAA and KAoS used Java serialization to encode the agent object, but the other two had their own encoding for agent state. The D’Agents encoding is particularly verbose, increasing the size of its agents. Section 3.1 presents the actual agent sizes from our experiments.

(g) Interestingly, none used RMI to move a jumping agent, choosing the more efficient socket mechanism (using TCP/IP). (h) D’Agents and NOMADS created a new socket connection for every jump, whereas EMAA and KAoS (really Voyager) “cached” the open socket and re-used it for subsequent jumps, reducing the overhead of jumps.

(i) Finally, where security mechanisms like encryption or authentication were available, they were disabled for these experiments. Such features have significant performance impact, but varied so much across systems that we chose to eliminate them from this initial study of the impact of other features.

3 Experiments

Our goal was to compare the scalability of the mobile-agent approach versus the client/server approach in an information-retrieval (IR) task as the number of clients increased. In our experiments, we explore the effect of increasing the number of clients and agents on a single mobile-agent server and its network connection. While our experiments do not always identify the boundaries of the performance space (not all experiments reach the limit of CPU or network capacity, for all agent systems), the results invite comparison between mobile-agent system designs, and bring some understanding to the structure of the performance space.
We implemented a simple IR task using both an agent and a client/server architecture. Our task filters the results of a simple keyword query on a collection of documents stored at the server. The client/server application downloads all documents resulting from every query, and does its filtering on the client machine. The agent-based application sends an agent to do the filtering on the server and returns with only the matching documents. The client/server application is written in C++ (for speed), while the agent-based application is written in Java (for mobility). The tradeoff is thus between network bandwidth consumption and processing speed, between a fast language on distributed clients and a slower language on a shared server. In our experience, mobile-agent applications often offer this tradeoff, and are particularly interesting in situations where the server does not support the application’s needs directly in its RPC interface.

We recognize that this experiment does not explore the full range of mobile-agent capability, in particular, the ability to jump to more than one server, but the scalability of mobile-agent systems even for single-hop applications is not yet well understood. The results of the single-hop experiments presented here are a critical foundation for future research, since even a multi-hop agent must decide whether to jump at all.

3.1 The Experiments

Our IR task involved two steps: a keyword query selected a set of documents from the collection, then a filter procedure scanned the selected documents to return those that contain a given string. Our document server implemented only the keyword-query operation. In our client/server implementation, the selected documents were returned to the client, which ran the filter. In our agent implementation, the agent filtered the documents at the server, then carried the resulting subset back to the client host. This application is representative of the type of computational task that might be used in an agent-based information-retrieval application.

Because the keyword query is common to both implementations, we removed that step, and used a fixed list of sixty 4096-byte documents. Although in both implementations we scan all sixty documents, we chose to declare a certain fraction of the documents to “pass”, ignoring the actual results of the filter, to increase our control over the size of the task output. In our experiments the “pass ratio” was either 5% or 20%.

We wrote the client/server applications in C++ using TCP/IP connections with a simple protocol for handshaking between client and server. The total query time is the time recorded at the client host to send the keywords to the server, receive the sixty documents, and filter the sixty documents on the client. We average these times to give average total query time.

We wrote the agent application in Java. The speed of any application written in the Java language, even with a JIT compiler, is slower than that of an equivalent implementation in C++. This difference works only in favor of the client/server approach; any performance benefits seen with the agent approach are not due to language differences. We ported the agent application to each of
Table 2. Comparison of average IR task times.

<table>
<thead>
<tr>
<th>Pass ratio</th>
<th>C++</th>
<th>D'Agents</th>
<th>EMAA</th>
<th>KaoS</th>
<th>NOMADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5% ratio</td>
<td>2.92ms</td>
<td>55.9ms</td>
<td>88.9ms</td>
<td>53.5ms</td>
<td>14497ms</td>
</tr>
<tr>
<td>20% ratio</td>
<td>3.02ms</td>
<td>61.6ms</td>
<td>96.1ms</td>
<td>73.6ms</td>
<td>14516ms</td>
</tr>
</tbody>
</table>

our four agent platforms, and reviewed the ported code to ensure that they were functionally identical.

There are four different virtual machines used by the four different mobile-agent platforms. D'Agents “AgentJava” uses a modified JDK 1.0.2, EMAA uses the Linux Blackdown JDK 1.2.2 port with JIT compiler (although the agents were compiled with a Java 1.3 compiler), KaoS uses the Sun JDK 1.3.0-02, and NOMADS uses its own JVM that has not yet been optimized for speed (Aroma release 20010327). To understand the speed differences, we ran the IR task alone in each platform.

C++ was markedly faster due to inefficient Java file I/O routines. All of the times reflect little actual disk activity because the underlying Linux file cache held all of the documents used. All of the Java tests used the same code, so any difference in performance was due to differences in the JVM or JIT compiler. Due to an optimized string-handling library, D'Agents was significantly faster than EMAA or KaoS, even though it did not use a JIT compiler. These differences accounts for some of the performance differences seen in our scalability tests below.

In our scalability experiments, each client agent looped over many queries. For each query, the agent jumped to the server, obtained the list of sixty documents, ran the filter over those documents, obtained the subset that “pass” the filter, and jumped back. The elapsed time, measured on the client, was the total query time. The agent also measured its time on the server, the “task time.” The “jump time” was the difference between the total time and task time.

In our experiments we varied the number of clients (1 to 20, each on a separate machine), the network bandwidth to the server (1, 10, 100 mbps), shared by all clients, and the pass ratio (5% or 20%).

Other parameters, fixed for these experiments, were the number of documents in the collection (60), the document size (4096 bytes), and the number of queries (10-1000 queries, depending on the agent system, using whatever number of queries was required to get repeatable results). The query rate was set to average one query per two seconds, but uniformly distributed over the range 0.25–0.75 queries per second. This randomness prevents agents from exhibiting synchronous behavior. This query rate is a maximum: if a query takes longer than two seconds to complete its task, the next query will not be started until the agent returns to its client machine.

The agent size depended on the agent system (Table 3). D’Agents includes all the classes with every agent, so its base agent size is the largest. NOMADS can optionally compress the agent state in transit, but that option was not used in our experiments.

1 In this paper we use the prefixes m and k to refer to powers of 10, and the prefixes M and K to refer to powers of 2. Thus 10 mbps refers to 10,000,000 bits per second.
Table 3. Agent sizes in bytes. All were measured “on the wire,” including all protocol overhead.

<table>
<thead>
<tr>
<th>Agent</th>
<th>D’Agents</th>
<th>EMAA</th>
<th>KAoS</th>
<th>NOMADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5% client to server</td>
<td>16,317</td>
<td>21,541</td>
<td>4,360</td>
<td>8,403</td>
</tr>
<tr>
<td>5% server to client</td>
<td>29,311</td>
<td>21,706</td>
<td>17,104</td>
<td>58,959</td>
</tr>
<tr>
<td>20% client to server</td>
<td>17,608</td>
<td>59,361</td>
<td>5,380</td>
<td>8,403</td>
</tr>
<tr>
<td>20% server to client</td>
<td>65,186</td>
<td>59,841</td>
<td>55,183</td>
<td>210,256</td>
</tr>
</tbody>
</table>

The size of agents going from client to server was incorrectly high in some cases, because our implementation had the same agent jump back and forth to obtain an average performance. After the first trip to the server, EMAA carried the documents to the server on every trip, and KAoS and D’Agents carried a small amount of extraneous state information. We expect that the effect on D’Agents and KAoS performance was small, but that EMAA’s jump times may have increased significantly. NOMADS encoded the documents with several bytes per character, while other systems used one byte per character. Although this makes the NOMADS agents much larger, the computational overhead of NOMADS dominated its results so the agent size was not much of a factor.

We ran the experiments on a set of twenty-one identical Linux workstations.² Twenty of the machines act as clients and one acts as the document server. We interconnected the computers with a 100 mbps Ethernet,³ but could reduce the bandwidth available by inserting a software bandwidth manager set to 10 mbps or 1 mbps.⁴ The network was full duplex at all bandwidths.

4 Results and Discussion

We plot several aspects of the results in a series of figures. We first consider the total query time, and then its components “task time” and “jump time.” Then we make a direct comparison between the client/server times and the agent times, by presenting the ratio between client/server and agent times.

The plots are missing some NOMADS points. Also, several of the EMAA points are slightly too low because of early termination of some agents. The general EMAA trends are correct, although little should be interpreted into the details of any non-linear wiggles. (Most of the agent systems had trouble in the 10 mbps tests, due we believe to some bugs in RedHat Linux 7.1 or its interaction with dummynet.)

4.1 Total Query Time

Each plot in Figure 1 shows the averaged per-query time, in milliseconds, for all systems (note there is a separate scale for the NOMADS data).

² VA Linux VarStation 28, Model 2871E, 450 mHz Pentium II, 256 MB RAM, 5400 rpm EIDE disk, running the Linux 2.4.2-2 (RedHat 7.1) operating system.
³ With a one-way measured throughput of 65 mbps.
⁴ DummyNet; see <http://info.iet.unipi.it/~huig/ip dummynet/>. 
Fig. 1. Total query times, for all systems, all three bandwidths, and both pass ratios. We show error bars indicating one standard deviation from the mean. Note that the vertical scale varies. The NOMADS data should all be read using the scale on the right-hand side of the graph.

The figure shows six plots, for three bandwidths (1, 10, and 100 mbps) and two pass ratios (5% or 20%). Generally speaking, any implementation will slow down linearly with the number of clients, due to increasing contention for the network and the server’s CPU. A query time exceeding 2000 milliseconds indicates that the clients have failed to sustain the desired query rate and have slowed to match the system’s capacity.

The slope of the line depends on the overhead of that implementation, the parameters of the query, and the speed of the network and CPU. An inflection point, where the slope suddenly increases, indicates that the load exceeded the limitations of the network or CPU. That effect can be seen most readily in our
10 mbps client/server experiments, where the demands of 12 clients exceed the limits of the network.

In the 1 mbps network, the fact that agents bring back only 5 or 20% of the documents allows them to be less sensitive to the constraints of the slow network, while the client/server approach is bandwidth-limited. Here, as in the 10 mbps network, D’Agents, EMAA, and K AoS clearly perform much better than client/server. NOMADS is much slower, due to its slower JVM (as we discuss in the next section).

In the 100 mbps network, however, client/server is the clear winner. In this environment, the network has more than enough bandwidth to allow the clients to retrieve all of the documents. With the network essentially free, the slower computation of the agents (using Java rather than C++, and sharing the server rather than dispersing among the clients) makes the mobile-agent approach a less attractive option.

The differences between mobile-agent systems are better examined in terms of the task times and jump times.

4.2 Task Times

Each plot in Figure 2 shows the task time for all agent systems. The task time is the time for computation of the filtering task only. Recall, however, that a client will not generate a query until its previous query has finished. In a network-limited configuration the query rate is reduced, reducing load on the server. Thus, the task times do depend on the bandwidth of the network.

The most notable feature in these graphs is the dramatic difference between the NOMADS times (which have a separate y-axis scale) and the other agent systems. This difference is due to the home-grown Aroma JVM used in NOMADS, which has not been tuned. The NOMADS data grows linearly with the number of clients, indicating that the server’s CPU is always the limiting factor for NOMADS.

The D’Agents task time is the fastest, in large part because it uses an older version of the JVM, with native (rather than Java) implementations of the critical string-manipulation routines. Our document-scanning application stresses those routines, leading to better performance for D’Agents.

The D’Agents time is largely constant, because the query rate is low enough to not stress the server CPU. In the 1 mbps network, the EMAA 5% tests begin to overload the server at about 8 clients, whereas the EMAA 20% tests overload the network and thus never overload the server. In the 10 mbps network, EMAA overloads the server CPU after about 10 (20%) or 12 (5%) clients. EMAA is slower than K AoS, largely due to a different version of the JVM (recall Table 2).

4.3 Jump Times

Each plot in Fig. 3 shows the average per-query jump time for each system. Recall that the jump time is the total query time minus the task time, so it includes all of the computational overhead of a jump (serialization, network protocol, deserialization, and reinstating an agent) as well as the network time.
Fig. 2. Task times, for all systems, all three bandwidths, and both pass ratios. We show error bars indicating one standard deviation from the mean. Note that the vertical scale varies. The NOMADS data should all be read using the scale on the right-hand side of the graph.

The jump times are most difficult to interpret, because they depend on the load of both network and server. The higher NOMADS times in fast networks, for example, are likely due to the heavy load on the CPU impacting the time needed for serialization, transmission, and deserialization of jumping agents. Note that NOMADS had the fastest jump times in the most congested network (1mbps at 20% pass ratio and 20 clients), despite having the largest agents.

In slow 1 mbps networks, we expect that systems with smaller agents (like EMAA and KAsS) jump faster than systems with bigger agents (like NOMADS and D'Agents). The results in the top row of Figure 3 are therefore surprising. NOMADS was fast, indeed sometimes fastest by far; the reason is that NOMADS
Fig. 3. Jump times, for all systems, all three bandwidths, and both pass ratios. We show error bars indicating one standard deviation from the mean. Note that the vertical scale varies.

task times were so large that agents only occasionally cross the network, and the network never experiences congestion or heavy load. EMAA was slower than KAoS because the net effect of carrying its payload in both directions was that EMAA’s agents were considerably larger.

In the 1 mbps case, the network was the bottleneck; in the 5% graph we can see D’Agents, EMAA, and KAoS change slope when they first encounter that bottleneck. In faster networks, the server’s load was the bottleneck. Again, we can see inflection points where D’Agents, EMAA, and KAoS first encounter that bottleneck. NOMADS was computation-bound in all cases.
It is difficult to attribute specific design decisions to the jump times measured in our experiments. Clearly it was helpful to have smaller agents, but even in the slowest network we found that the computational overhead was often a determining factor in the time required for a jump.

4.4 Ratio of Client/Server Time to Agent Time

Each plot in Figure 4 shows the “performance ratio,” which is the client/server query time divided by the mobile-agent query time. A ratio of 1 indicates that the agent approach and the client/server approach are equivalent in performance; higher than 1 indicates that agents were faster. The NOMADS ratios are indistinguishable from zero because their times were so large. For the other three systems, there are three different effects, dependent on bandwidth.

In the 1 mbps curves, we see that the performance ratios climbed, and then fell or level off. For small numbers of agents, the performance ratio improved quickly because the client/server approach was bandwidth limited, while the agent approach was not. With a few more agents, it reached the network bandwidth limit and became slower, reducing the performance ratio. Once both client/server and agent performance reached the same slope, the performance ratio leveled off. In the 20% case, the ratio was about 4–5, which is reasonable considering that the agents moved 1/5th of the data, but with some overhead. In the 5% case, where the agents moved 1/20th of the data across the network, the ratio was 8–15.

In the 10 mbps curves, we see a different effect. Here, the agents never hit the network limit, but the client/server implementation hit the limit at 12–13 clients. The performance ratio suddenly improved. The performance ratio for KAOs and EMAA then dropped, due to increasing server load.

In the 100 mbps curves, all performance ratios were low and declined steadily as more clients were added, due to increasing contention for the server’s CPU.

5 Related Work

Researchers have developed numerous mobile-agent systems over the past decade. Few papers, however, present any substantial study of system performance. Fewer still examine scalability. We discuss the most relevant papers here.

Ismail and Tichy [9] compare the performance of client/server (RMI) with mobile agents. The client contacts one server to obtain a list of hotels, and another server to obtain a phone number for each hotel (one a time). In the alternative implementation a mobile agent visits the first server and then the second server, returning with all phone numbers for all candidate hotels. Although the agent is a multi-hop agent, unlike those in our study, the application is analogous. Mobile agents provide a performance advantage when the agent retrieves a sufficient number of candidate hotels. In their study, however, there was only a single client and a single agent, and no other load on the servers.

Johansen [10] used an application like ours, though using images or video files rather than text documents. The results are directly analogous to our own
Fig. 4. Performance ratios for all systems, for both pass ratios, combining all systems on one plot. Note that the vertical scale varies.

results, with similar crossing points in the mobile-agent vs. client-server performance curves. They do not, however, study the scalability of the server, since there is one client sending one agent to the server.

Sträßer and Schwehm consider an abstract application, using an analytic model and parameters derived from the Mole mobile-agent platform [13]. They consider only a single mobile agent, although it may visit multiple servers. They have limited experiments on only one mobile-agent system, and they do not evaluate the scalability of an agent server.

Küpper and Park use an analytic model, parameterized by a small experiment, to predict the scalability of a telecommunications application [11]. They compare two approaches: stationary agents, in which the call-setup code for a user is always resident in the user’s home network, and mobile agents, in which
the call-setup code moves to the user’s current network. Mobile agents lead to
improved call-setup times as long as the user makes enough calls before moving
to a new network. Their paper does not measure real mobile-agent systems, nor
study the scalability of a mobile-agent server.

Baldi and Picco also use an analytic model, parameterized by experiments,
to examine a different aspect of scalability [1]. They compare the network traffic
generated by a variety of approaches (including client/server and mobile agents)
for collecting statistics from a distributed set of network devices. They conclude
that mobile agents (in general, mobile code) can reduce network traffic, relative
to a client-server approach, and thus allow their application to support a larger
number of devices. They consider only a single mobile agent. They compare
only network traffic in bytes and no measures of time. They do not consider the
scalability of the mobile-agent platform itself.

Theilmann and Rothermel also examine the performance of a mobile-agent
application that visits many data servers to filter data [15]. The client/server
approach downloads all data for filtering at the client. In the mobile-agent ap-
proach, one mobile agent is dispatched as close to each data server as possible.
They achieve significant cost savings whether they used hop counts or round-trip
time as basis for measuring distance. “Cost” seems to relate to total number of
bytes transferred across the network. As in our study, the cost savings depend
entirely on the amount of data examined (and filtered out) on each remote host.
They do not study scalability of the agent servers, however, since there is only
one agent sent to each data server.

Woodside [16] proposes a model for scalability and analysis of mobile-agent
systems. The paper examines scalability in terms of the time for a mobile agent
to complete a “tour” (an execution that involves visiting several hosts) as the
number of hosts (agent servers) increases with a corresponding increase in the
number of agents. The model presented does not account for communication
costs, one of the central factors in our study. Also, the paper does not provide
any experimental results.

Dikaiakos and Samaras [6,5] develop a framework for performance analysis
of mobile-agent systems. They propose three layers of benchmarks: micro-
benchmarks that test individual operations such as messaging and migration,
micro-kernels that are small, synthetic tasks that would be part of typical appli-
cations, and application-kernels that use actual application-level functionality
and workloads. They present experimental results for three real mobile-agent
platforms using micro-benchmarks and micro-kernels that describe performance
in terms of throughput for a single agent, but do not address scalability.

The consistent theme of previous work, confirmed by our own work, is that
a mobile-agent approach outperforms a client-server approach as long as the
application involves analysis of enough information, and enough reduction of
the data returned to the client, to outweigh the overhead of sending the mobile
agent in the first place. Our study is unique in its study of a server heavily
loaded by mobile agents from multiple clients, and unusual in its cross-comparison
of several mobile-agent systems.
6 Conclusion

In our experiments we found that the scalability of mobile-agent systems, in
collection to an alternative client/server implementation of the same appli-
cation, depends on many different environmental parameters. Overall, the four
mobile-agent systems we examined scale reasonably well from 1 to 20 clients,
but when we compare them to each other and to a client/server implementation
they differ sometimes dramatically. The client/server implementation was highly
dependent on sufficient network bandwidth. The agent implementations saved
network bandwidth at the expense of increased server computation.

The performance of NOMADS clearly suffered from the untuned virtual
machine. The relative performance of the other three mobile-agent systems varied,
depending on the mix of computation and network in the application, reflecting
their different mix of overheads. The optimized string functions in the D'Agents
JVM helped prevent server overload when the network was fast. The smaller
agents of KaoS and EMMA were an advantage in slower networks, although the
cost of serialization hurt EMMA.

Our experiments are admittedly only a first step toward understanding the
performance of, and scalability of, mobile-agent systems. These results are for
a single application, in which mobile agents hop once to the server and once
back to the client. The application exercises string processing on the server,
and the transportation of documents in a jumping agent, but does not exercise
agent-agent communication, security mechanisms, multi-hop mobile agents, or
complex network topologies. The relative performance of our four mobile-agent
systems depends in part on the current state of their implementations. Indeed, it
is difficult to tease out the performance effects of the design differences outlined
in Table 1, because their effects were confounded.

It is clear from our results that mobile agents can be beneficial in situations
with low network bandwidth and plentiful server capacity. Indeed, in many en-
vironments it is easier to add more server capacity than to add network capac-
ity; particularly those with wireless networks. For applications demanding high
performance and scalability to hundreds or thousands of active agents, further
research is necessary to develop light-weight agent systems and scalable agent
platforms. We are investigating automated ways to build parallel or distributed
mobile-agent servers and services.

References

1. M. Baldi and G. P. Picco. Evaluating the tradeoffs of mobile code design paradigms
   industrial-strength open agent architecture. In J. Bradshaw, editor, Software
3. J. M. Bradshaw, N. Suri, A. K. Cañas, R. Davis, K. Ford, R. Huffman, R. Jeffers,


Performance Evaluation of Mobile-Agent Middleware: A Hierarchical Approach*

Marios Dikaikos, Melinos Kyriakou, and George Samaras

Department of Computer Science
University of Cyprus
CY-1678 Nicosia, CYPRUS

Abstract. In this paper, we introduce a hierarchical framework for the quantitative performance evaluation of mobile-agent middleware platforms. This framework is established upon an abstraction of the typical structure of mobile-agent systems and is implemented through a set of benchmarks, metrics, and experimental parameters. We implement these benchmarks on three mobile agent platforms (Aglets, Concordia and Voyager) and run numerous experiments to validate our framework and compare the mobile-agent middleware environments quantitatively. We present results collected from our experiments, which help us understand MA performance and identify existing bottlenecks. Our results can be used to guide the improvement of existing platforms, the performance analysis of other systems, and the performance prediction of MA applications.

1 Introduction

The Mobile Agent (MA) paradigm is one of the most promising approaches for developing distributed applications on Internet [9]. The employment of Java-based MA technologies for the development of next-generation Internet systems opens numerous research problems. In our work, we focus on quantitative performance evaluation of mobile agents and propose a framework for investigating the performance characteristics of MA-based platforms and applications.

In this context, we introduce a performance evaluation approach that can be used to gauge the performance characteristics of different mobile-agent platforms. This approach extends and refines previous work of ours [12,6], by defining a “hierarchical framework” of benchmarks designed to isolate performance properties of interest at different levels of detail. We identify the structure and parameters of benchmarks and propose metrics that capture performance properties of interest. We implement these benchmarks upon three Java-based, mobile agent middleware platforms (IBM’s Aglets [4], Mitsubishi’s Concordia [13] and ObjectSpace’s agent-enhanced object request broker, Voyager [7]), and run various experiments.

* This work was supported in part by the Research Promotion Foundation of Cyprus, grant PENEK-No 23/2000.

© Springer-Verlag Berlin Heidelberg 2001
Experimental results provide us with initial conclusions that lead to further refinement and extension of benchmarks and help us investigate the performance characteristics of the platforms examined. The remaining of this paper is organized as follows: Sections 2 and 3 introduce our performance analysis framework. Sections 4 and 5 present the implementation of the first two levels of our framework with a suite of micro-benchmarks and micro-kernels, and report our experimentation results. We conclude in Section 6.

2 Basic Elements and Application Frameworks

Typically, the performance assessment of software systems is conducted through experimentation and monitoring, simulation, modeling and combinations thereof. The more complex a system is the harder its performance evaluation becomes, dictating the employment of these techniques at various levels of abstraction. To this end, software systems are modeled as hierarchical structures of interacting modules, i.e., subsystems and objects; each module is assigned a performance model that incorporates performance and load parameters of relevance, and a description of the underlying architecture and workload [14]. Model development is performed in a “top-down” manner, starting from high-level structure and moving toward code implementation. Experimentation and/or simulation can be used at various layers of abstraction to specify the values of modeling parameters.

The development and assembly of performance models for MA middleware is more complicated than for more “traditional” parallel, distributed or object-oriented software; when analyzing the performance of MA-based systems, we must take into account issues such as: the absence of global time, control and state information; the complex architecture of MA middleware and the agility of MA systems; the variety of distributed computing (software) models that are applicable to mobile-agent applications; the diversity of operations found in MA middleware, and the additional complexity introduced by issues that affect the performance of Java (run-time compilation, memory management, garbage collection, etc.).

To cope with the complexity of MA-performance evaluation, we propose the adoption of a hierarchical approach that takes into consideration the structure of typical MA-based applications. This structure is influenced, first, by the mobile-agent platform adopted to develop an application. MA platforms are middleware systems with a programming interface that exposes to the programmer a set of core functionalities providing support for object mobility (transportation and location services), communication between objects, security, fault-tolerance etc. [2, 7,8]. MA platforms are differentiated by their functionality, programming interface and performance characteristics, all of which are influenced by underlying implementation details. The structure of a MA application is further determined by the design choices that the application developer makes on how to use the API provided by the middleware platform, when developing the particular appli-
cation. Typically, these design choices can be abstracted as mobile-agent *design patterns* [1].

Therefore, to investigate the performance of mobile-agent applications, we have first to develop an approach for capturing basic performance properties of MA middleware. These properties must be defined independently of how particular mobile-agent API’s are used to program and deploy applications and systems on Internet. Then, we have to analyze the performance characteristics of design patterns commonly used in MA applications. To facilitate this approach, we introduce two abstractions: *Basic Elements* and *Application Frameworks*.

We define as **Basic Elements** the set of basic abstractions that incorporate the fundamental functionalities commonly found and used in MA platforms. For the objectives of our work, the basic elements of MA platforms are identified from existing, “popular” implementations as follows [2,4,7,8]: a) *Agents*, defined by their state, implementation (byte-code), capability of interaction with other agents/programs (interface), and a unique identifier. b) *Places*, representing the environment in which agents are created and executed. A place is characterized by the virtual machine executing the agent’s byte-code (the *engine*), its network address (location), its computing resources, and any services it may host (e.g., a database gateway or a Web-search program). c) *Behaviors* of agents within and between places, which correspond to the basic functionalities of a MA platform, such as: creating an agent at a local or remote place; dispatching an agent from one place to another; receiving an agent that arrives at some place; communicating information between agents via messages or messenger agents; synchronizing the processing of two agents; locating an agent on the move, etc.

Basic elements of MA systems are combined into scenarios of MA-use, which we call **Application Frameworks**. In Object-orientation, *software frameworks* represent a way of “structuring generic solutions to a common problem by providing the structure of a program but no application-specific details” [3]. The overall control and the flow of execution is provided by the framework and therefore does not need to be rewritten for each new problem. Accordingly, application frameworks of MA’s define solutions common to various problems of agent design and are defined in terms of places participating in a scenario, agents placed at or moving between these places, and interactions of agents and places (agent movements, communication, synchronization, resource use). Application frameworks correspond to widely applicable models of distributed computation on particular application domains, and represent widely accepted and portable approaches for addressing typical agent-design problems [1]. Typically, application frameworks are the building blocks of larger MA applications.

We focus on application frameworks that correspond to the Client-Server model of distributed computing and its extensions for mobile computing: the Client-Agent-Server model, the Client-Intercept-Server model, the Proxy-Server model, and variations thereof that use mobile agents for communication between the client and the server; more details on these models are given in [12]. Additional application frameworks correspond to the *Roaming Mobile-Agent Model*, and the *Forwarding* and *Meeting* agent-design patterns. The Roaming MA model
corresponds to the case of an agent that roams from one place to the other, engaging in some interaction with the places visited. The Forwarding pattern “allows a given place to mechanically forward all or specific agents to another place” [1]. The Meeting pattern provides a way for two or more agents to initiate local interaction at a given place [1,4]. The Forwarding and Meeting patterns represent the performance traits of agents and places in terms of their capability to re-route agents and to host inter-agent interactions.

3 A Hierarchical Performance Evaluation Framework

In view of the remarks above we propose a framework for the Hierarchical Evaluation of MA-performance, which consists of four layers of abstraction (see Figure 1). At a first layer, our framework explores the performance traits of basic elements of MA platforms, seeking to expose their performance behavior: how fast they are, what is their overhead, if they become a performance bottleneck when used extensively, etc.

Having isolated the performance characteristics of basic MA elements, we explore the characteristics of application frameworks in order to explain the performance behavior of full-blown applications that use these frameworks as building blocks. Consequently, at the second layer of our framework, we investigate implementations of popular application frameworks upon simple workloads. In particular, we measure metrics capturing the performance capacity of an application framework, the overhead incurred by the interaction of its constituent elements, the bottlenecks affecting its performance, etc. For example, an application framework could involve an agent residing at a place on a fixed network and providing database-connectivity services to agents arriving from remote places over wireless connections. This framework may exist within a large digital library or e-commerce application. It may, as well, belong to the “critical path” that determines end-to-end performance of that application. To identify how this framework affects overall performance, we have to find out what is the overhead of transporting an agent from a remote place to a database-enabled place, connecting to a database agent, performing a simple query, and returning the results over a wireless connection. Interaction with the database agent should be
kept minimal because we are trying to capture the overhead of this framework and not to investigate database behavior. We also need to quantify how many requests can be served by the database agent per second, etc.

It is interesting to explore the performance behavior of instances of these frameworks under conditions expected to occur in a real execution of a full-blown application. To this end, we can enrich the scenarios implemented in the application frameworks by extending the functionality of mobile agents and by simulating realistic workload conditions. This is the focus of the third layer of our hierarchy, where we study micro-applications, i.e., implementations of application frameworks that realize particular functionalities of interest (e.g., database connectivity) and run on synthetic workloads. Finally, at the fourth layer of our framework, we study full-blown applications running under real conditions and workloads.

Our approach has to be accompanied by proper metrics, which may differ from layer to layer, and parameters representing the particular context of each study, i.e., the processing and communication resources available and the workload applied. It should be stressed that the design of our performance evaluation in each layer of our conceptual hierarchy should provide measurements and observations that can help us establish causality relationships between the conclusions from one layer of abstraction to the observations at the next layer of our performance analysis hierarchy.

To apply our hierarchical performance evaluation framework in the study and comparison of performance characteristics of different MA platforms and MA-based applications, we propose three layers of benchmarks that correspond to the first three layers of the hierarchy of Figure 1. These benchmarks are defined as follows:

- **Micro-benchmarks**: short loops designed to isolate and measure performance properties of basic elements of MA systems, for typical system configurations. Micro-benchmarks test the performance of simple activities (behaviors) provided by the basic elements of a MA system.
- **Micro-kernels**: short, synthetic codes designed to measure and investigate the properties of application frameworks, for typical system configurations.
- **Micro-applications**: instantiations of micro-kernels for real applications.

Here, we propose to use places with full application functionality and employ synthetic workloads complying to specifications like the TPC-W.

In the following sections, we introduce a suite of micro-benchmarks and micro-kernels that we use to evaluate the performance of mobile-agent middleware quantitatively. In earlier work we have examined micro-applications that involved the use of mobile agents to provide database access over the Web [12]; a study of micro-applications will be conducted in future work.

Our benchmarks are accompanied by parameters that define the context of our experimentation, and the metrics measured. Parameters determine the workload that drives a particular experiment, expressed as the number of invocations of some basic element or application framework, and the resources attached to participating places and agents. Metrics represent a concise description of the performance characteristics isolated by our benchmarks.
Our benchmarks can be parameterized according to the following parameters: *Operating System* and *Place Configuration* represent the resources of each place involved in our experimentation; *Channel Configuration* represents the network upon which we conduct our experiments, which can be a LAN, a WAN, a wireless network, or combinations thereof. *Agent Size* and *Message Size* represent the size of an agent and a message exchanged between two agents, respectively. *Loop size* defines the number of times a particular benchmark is executed to gather time measurements. Additional benchmark-specific parameters are employed in micro-kernels and will be described later.

The number of parameters involved in our benchmarks lead to a huge space of experiments, many of which may not be useful or applicable. Therefore, we have conducted preliminary experiments with three commercial platforms, IBM’s Aglets, Mitsubishi’s Concordia, and ObjectSpace’s Voyager, and tried various parameter settings before settling to a small set of experimental parameters and benchmark configurations that provide useful insights. Our experiments involve places located at different computing nodes within the same LAN, agents with the minimum functionality that is required for carrying out the behaviors studied, and messages carrying minimal information between agents. We have used a 100 Mbps Ethernet with 18 PCs, equipped with Pentium III processors running at 500MHz and 64MB main memory. The PCs ran the Microsoft’s Windows NT 4.0 Operating System and Sun’s JRE 1.1.7. On this platform we experimented with Aglets version 1.0.3, the professional edition of Voyager ORB, version 3.1, and an evaluation copy of Concordia, version 1.1.4. The experiments were conducted at night, when the utilization of the LAN was minimal. We also ran some experiments under heavier network load (when the lab was used by students to run applications from a central file-server, to browse the Web, etc.). All data reported in the following sections correspond to the low-network-traffic case, unless mentioned otherwise. In future experiments, we plan to incorporate setups including wireless Ethernet and connectivity over WANs.

For most of our benchmarks we report four metrics: *Total time* is the total elapsed time it takes to run a particular benchmark. This metric represents the performance of the basic activity examined by the benchmark. A study of the total-time for different benchmark parameters can identify bottlenecks that arise under high loads (large loop size) and test the robustness of each platform. *Average time* provides an estimate of the time it takes for a particular basic activity of a MA system to complete; for instance, the time of sending a short message, dispatching a light agent, etc. *Peak rate* is the maximum measured rate of a basic activity, defined as the number of these activities carried out per second. *Sustained rate* is the number of basic activities carried out per second, when we conduct stress-tests, i.e., run an experiment continuously over a long period of time. For instance, a sustained rate of 40 for the agent-creation benchmark means that we can generate approximately 40 agents per second on the particular machine running the experiment, if the experiment is executed continuously over a period of time. Additional, metrics are measured in certain micro-kernels and will be described later.
4 Micro-Benchmarks

In this section, we present the suite of proposed micro-benchmarks and experimental results derived by these benchmarks. The basic components we are focusing on are: a) mobile agents, used to materialize modules of the various distributed computing models and agent patterns; b) messenger agents used for flexible communication, and c) messages used for efficient communication and synchronization. Accordingly, we define the micro-benchmarks presented in Table 1 and present the metrics measured in a number of experiments with these benchmarks. Excerpts of the code implementing these benchmarks are presented in [5].

4.1 Agent Launching: CL, CR, and AD

With the CL micro-benchmark we study the overhead of agent-creation. To this end, we generate 1 to 1000 agents and measure the total elapsed time. The left diagram of Figure 2 reports the average time for generating one agent with respect to the total number of created agents. From this diagram, we can easily see that the overhead of creating a single agent in Concordia is negligible with respect to the overhead in Aglets and Voyager; furthermore, that Voyager outperforms Aglets.

It is interesting to note that the time it takes to create an agent drops with the increase of loop size, for all platforms. This happens because, after the first time an agent is created, its byte-codes are already cached in the agent-host’s memory. Therefore, subsequent agent creations take minimal time. On the other hand, the better “scalability” of Concordia and Voyager over Aglets that we observe in the left diagram of Figure 2, is attributed to memory management mechanisms implemented in both platforms: when heap space is consumed, the two platforms transfer inactive agents to disk, thus maintaining a minimum of free space [10,11]. Table 2, presents the agent-creation capacity of the three platforms (peak and sustained).
Fig. 2. CL, CR & AD: Average timings for agent creation and dispatch.

Table 2. CL, CR and AD: Peak and sustained rates.

<table>
<thead>
<tr>
<th>Platform</th>
<th>CL</th>
<th>CR</th>
<th>AD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Peak Sustained</td>
<td>Peak Sustained</td>
<td>Peak Sustained</td>
</tr>
<tr>
<td></td>
<td>(agents/sec)</td>
<td>(agents/sec)</td>
<td>(agents/sec)</td>
</tr>
<tr>
<td>Concordia</td>
<td>3125 3000</td>
<td>3125 310</td>
<td>25.68 25.6</td>
</tr>
<tr>
<td>Aglets</td>
<td>65.78 11</td>
<td>29.76 11.05</td>
<td>5.9 5.36</td>
</tr>
<tr>
<td>Voyager</td>
<td>1189.06 1100</td>
<td>38.8 38.8</td>
<td>11.58 8.31</td>
</tr>
</tbody>
</table>

The CR benchmark measures the total time it takes to create agents at a remote host. To this end, we use a stand-alone JAVA program running on an “origin” host and issuing instructions to generate 1 to 1000 agents on a remote place. We time the overall overhead of agent creation at the origin place. To remotely create agents, the remote place needs to have the necessary classes locally or to be able to download these classes from another place on demand, during agent-creation. This is accomplished in a number of different ways: (a) Under Concordia, a messenger agent migrates from the origin place to another place. Upon arrival, the messenger creates a new agent at the remote place. The messenger transports with it the classes required by the agent under creation. (b) A Voyager agent at the remote place can load classes from other locations on demand. To this end, it employs a Resource Loader object which resides in its Voyager server. The Resource Loader maintains a registry of remote Voyager servers, which may store useful classes and serve them over the network. Whenever an agent seeks a class that is not available in its local classpath, it invokes the Resource Loader which returns an interface (proxy). Through that interface, the agent can access the remote class. (c) An Aglet can load a remote class on demand from a remote Tahiti server, which is the agent execution environment (place) of Aglets. To this end, the Aglet must establish an additional network connection with the remote place. In order to make the remote classes available through the network, they should be placed in the secondary storage of the remote host and be included in the classpath of the remote place at its initialization.

The middle diagram in Figure 2 shows our measurements for the CR benchmark. As we can see, Concordia and Aglets have better performance than Voy-
ager for a small number of created agents. Again, Concordia is the clear "winner," even for large numbers of created agents. As we increase the number of created agents, however, the average time to create an agent in Voyager drops faster than the respective time in Aglets, and the values of the two platforms converge. The performance of the three platforms in terms of their capacity to create agents remotely is summarized in Table 2. It is interesting to note that remote creation of agents under Concordia and Voyager is approximately an order of magnitude slower than local agent-creation. Furthermore, we note that, for Concordia and Voyager, the peak and sustained rates of agent creation are almost equal, which is a result of their improved robustness. In contrast, Aglets performance drops for very large numbers of created agents.

The AD benchmark measures the overhead of dispatching mobile agents to a remote place in a LAN. We create and dispatch 1 to 1000 agents to the remote place. We measure only the time of the dispatch operation and plot our results in the right diagram of Figure 2. As we can see from this diagram, Voyager has the best performance in dispatching agents for short loop sizes. As we increase the number of agents launched, Concordia's performance improves considerably, due to its caching mechanisms. Furthermore, Concordia is very robust, even in cases of heavy network load. In contrast, we noticed that Voyager and Aglets crashed occasionally when we dispatched more than 600 agents in an experiment, and the network was heavily loaded. From Table 2 we can see that a Concordia place can dispatch 25.6 agents per second, whereas Aglets and Voyager can send only 5.36 and 8.3 agents per second, respectively.

4.2 Inter-agent Communication: MSG-1W, MSG-2W, SYNCH, and MSG-MA

The MSG-1W benchmark measures the elapsed time for sending non-blocking messages from one agent to another. For this benchmark we employ two mobile agents located at two different hosts in the same LAN. The first agent sends a number of messages to the second; there is no explicit acknowledgment of receipt from the second agent. We measure the time it takes to send 1 to 1000 messages of equal, minimal size.

To implement MSG-1W we employ the OneWay method of Voyager. In particular, a Voyager agent sends a message to a destination agent via the destination-agent’s local "proxy." The message consists of the remote agent's name, the name of the method that will be invoked upon receipt of this message by the destination agent, and the arguments that will be passed to this method. The OneWay method does not return a reply and is non-blocking. Voyager employs standard Java serialization to transport messages across the network. In Aglets we implement MSG-1W with the sendAsynMessage() method, which is invoked on the remote-agent’s proxy that serves as a message gateway for the Aglet. Here, the message is an object. On the other hand, Concordia uses events to implement message-passing: events are sent by the dispatching agent to an Event Manager through the postEvent() method. The receiving agent must register with that Event Manager as well, to listen for and receive particular events. Examples of message-passing implementation are given in [5].
Fig. 3. MSG-1W, MSG-2W & SYNCH: Average time measurements

<table>
<thead>
<tr>
<th>Platform</th>
<th>MSG-1W Peak Sustained (msg/sec)</th>
<th>MSG-2W Peak Sustained (2msg/sec)</th>
<th>SYNCH Peak Sustained (synchs/sec)</th>
<th>MSG-MA Peak Sustained (agent-round trips/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concordia</td>
<td>73.39</td>
<td>61.68</td>
<td>16.06</td>
<td>12.147</td>
</tr>
<tr>
<td>Agetes</td>
<td>102.94</td>
<td>80.3</td>
<td>96.15</td>
<td>4.93</td>
</tr>
<tr>
<td>Voyager</td>
<td>1146.78</td>
<td>625</td>
<td>598.32</td>
<td>9.38</td>
</tr>
</tbody>
</table>

Figure 3 (left) presents the diagram of the average time per message for each experiment. From this diagram we can see that Voyager has the fastest messaging. Furthermore, its messaging is very robust, even under heavy network load. One-way messaging performance of Agetes and Concordia is similar; nevertheless, Agetes crashed occasionally when sending too many messages. From the left diagram of Figure 3 we also note that the average time to send a message decreases with respect to the number of messages dispatched during each experiment. This figure is stabilized for larger loop sizes. In Voyager and Agetes this happens because, after the first message is sent to the remote agent, all involved classes are installed in the caches of both places participating in the message-exchange. Consequently, the “initiation” overhead incurred by subsequent messages is minimal. In Concordia, the dispatch of repeated messages from one agent toward another, via an Event Manager, requires only one connection to the Event Manager. As we send more messages, the connection overhead is amortized across all messages.

Table 3 presents the peak and sustained rates for message-dispatching. A Voyager agent can send 1146.78 messages per second, whereas the capacity of Concordia and Agetes are 73.2 and 102.94 agents per second, respectively.

The MSG-2W benchmark measures the time it takes to send non-blocking messages from one agent to another, with asynchronous acknowledgments of receipt. To this end, we use two agents located at two different hosts in our LAN. The first agent sends non-blocking messages to the second; upon arrival of a message, the recipient-agent immediately replies back to the sender, acknowledging the receipt. To this end, we invoke the sendFutureMessage() method in Voyager and the future() in Agetes. We measure the time it takes to send 1 to
1000 messages and receive the respective acknowledgments. In all experiments we use messages of equal, minimal size. As expected, Voyager exhibits the best performance, with minimal fluctuation with respect to the number of dispatched messages (see Figure 3, middle). Concordia and Aglets have comparable performance when dispatching continuously up to 50-60 messages. For larger message numbers, Aglets crash. This explains the very small rates reported for Aglets in Table 3.

The SYNCH benchmark measures the time it takes to perform a synchronization between two agents; the synchronization operation is implemented with the exchange of two messages. To this end, we place the agents at two different places (hosts) in the same LAN. One agent sends a message to the other and gets blocked until it receives a reply. The second agent waits for incoming messages; upon receiving a message, it replies back. We use the `synch()` method in Voyager and the `sendMessa ge()` method in Aglets. We conducted this "ping-pong" experiment from 1 to 1000 times. For each experiment, we measured the total elapsed time it takes to complete all synchronization activities. Figure 3 (right) presents our measurements. In agreement with the MSG-1W and MSG-2W benchmarks, Voyager exhibits a synchronization capacity significantly higher than Concordia and Aglets. Furthermore, it achieves a synchronization rate (number of SYNCH’s per second) which is practically constant with respect to the number of the ping-pong operations performed.

As we can see from Table 3, Voyager agents are capable of conducting 413 synchronizations per second on the same LAN. Aglets come second in the synchronization capacity (92 SYNCH’s per second, sustained) and Concordia achieves only 14 SYNCH’s per second, sustained. We believe that Voyager outperforms Concordia and Aglets due to its low overhead of message initiation. This is also the reason why in Voyager the peak rate of SYNCH’s is reached for small loop-sizes, and does not drop significantly for larger loop-sizes. It is interesting to note that the implementation of a blocking-message exchange in Aglets is much more efficient than the implementation of messaging with asynchronous acknowledgments, and that its performance is comparable to the performance of one-way messaging with no acknowledgment.

The MSG-MA benchmark measures the overhead that arises when two places (hosts) interact via a messenger agent; both hosts reside in the the same LAN. To implement this benchmark, we create an agent in the first place and set its itinerary so that the agent moves to the second place and then returns back. Upon return, the same agent is re-dispatched and retracted for a number of times. Our experimental parameter is the total number of round-trips performed by the messenger agent. We conduct experiments for 1 to 1000 round-trips, and measure the total elapsed time. We present our measurements in the left diagram of Figure 4. Table 3 summarizes the peak and sustained rates as shown in Figure 4 for the average time of messenger round-trips.

As we can see from Figure 4 (left), Concordia and Aglets exhibit better performance for one and two round-trips. Nevertheless, the average time per round-trip in Voyager drops much faster as we increase the number of round-trips. The same figure for Aglets is stabilized after 10 round-trips. Consequently, Voyager exhibits the best performance for larger numbers of round-trips (over
500). It is interesting to note that the average delay of a messenger-agent’s round-trip in Concordia increases with the number of round-trips. We believe this is a side-effect of the agent-roaming implementation in Concordia: every time an agent has to move to another host, a Destination object must be added to the agent’s Itinerary, in order to determine its next move. The Itinerary is a data structure separate than the agent, maintained at a different location than the agent itself. The Itinerary is composed of a list of Destination objects [13]. Each Destination indicates the place (host) to which the agent is expected to travel, and the name of the method that the agent will execute upon arrival to that place. In our experiments for MSG-MA we employ a messenger agent that travels numerous times back and forth between two places.

In contrast to Concordia, an agent in Voyager or Aglets can be re-launched to a new destination, upon arrival to some place. To this end, a method can be called by the agent to determine its next destination. In particular, in Aglets we use the dispatch method to send an Aglet to a remote location. This location is passed as argument to the dispatch method (Aglet.dispatch (URL destination)). Upon arrival to its destination, the Aglet is pulled back to its original place with the retractAgletO method. In Voyager, we use the Mobility.of() method to obtain the mobility facet of an agent and invoke the moveTo() method of its IMobility interface. To pull the agent back, we call again moveTo().

5 Micro-Kernels: ROAM, PROXY, and FORW

Due to space limitations, in this section we focus on three application frameworks only: Roaming MA, Proxy-Server, and the Forwarding pattern; early experimentation with other application frameworks (C/S, C/A/S, and C/I/S) has been presented in [12]. Accordingly, we define the micro-kernels presented in Table 4.

The ROAM micro-kernel investigates the overhead incurred by an agent that roams from place to place in a network. To implement this benchmark we
create an agent at a place, and set its itinerary so that it visits a number of places and then returns back to its place of origin. We dispatch this agent and measure the total time it takes to complete its trip. The itinerary is fixed before the agent starts its voyage. It should be noted that the implementation of agent mobility in ROAM is different than that in MSG-MA, for the Aglets platform: upon successful arrival of an Aglet to a new place, the onArrival() method is invoked automatically. We have overwritten onArrival so that it dispatches the Aglet to its next destination. Experimental parameters of this benchmark are the number of hops taken by the roaming agent before coming back to its origin place, and the different places it visits (in its journey, an agent can visit one place multiple times).

In Figure 5 (right), we report measurements taken when an agent roams four different places (including its starting point), making 4 to 4000 hops totally. As we can see from this diagram, the average time per hop in Voyager is practically constant with respect to the total number of hops. Aglets average performance improves as we increase the number of hops; obviously a side-effect of an initial high overhead incurred when an agent visits a place for the first time, which is amortized by the reduced cost of subsequent re-visits. The performance behavior of Concordia worsens for longer agent voyages, in concordance with the MSG-MA micro-kernel. We believe this is a side-effect of the handling of itineraries in Concordia.

Table 4. Micro-kernels.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROAM</td>
<td>Captures the overhead of a roaming agent.</td>
</tr>
<tr>
<td>PROXY</td>
<td>Captures the performance of a proxy-agent serving requests from a number of</td>
</tr>
<tr>
<td>FORW</td>
<td>Captures the overhead of a forwarding agent, residing at a place, receiving</td>
</tr>
</tbody>
</table>

Fig. 5. PROXY & FORW: Service rates.
The Proxy-Server model is an extension of the Client-Agent-Server model with the “Agent” accepting connections from many clients and forwarding requests to more than one Servers. This scenario arises in cases where an agent is dispatched to the “edge” of the network to act as proxy. This agent receives incoming client requests and forwards them to appropriate servers, optimizing the communication of clients and servers, caching server replies, etc. The PROXY micro-kernel investigates the performance of the Proxy-Server model when implemented on top of a MA middleware platform. To this end, we use a mobile agent as proxy that mediates between several clients and servers. The proxy agent waits for request messages from agent-clients located at different hosts. Whenever it receives a message, it inspects the request message and forwards it to the appropriate server. Upon receipt of a request, a server replies directly to the client that sent it. Upon receipt of the server’s reply, the client issues a new request, following the same procedure.

The PROXY benchmark is parameterized with respect to the number of clients and servers involved in our experiments, and the total number of requests handled by the proxy-agent. Here, we report measurements from one experiment involving three server-agents and twelve client-agents. We measure the time it takes the proxy-agent to receive and forward incoming 1 to 5000 requests to the appropriate servers. Moreover, we report the rate of request-handling achieved by the proxy-agent. Figure 5 [left] presents a diagram with our measurements. Further experiments are reported in [5]. As we can see from this diagram, the performance of each MA platform converges to a certain sustained rate of requests served per second. In the twelve-client case, the Concordia, Aglets and Voyager proxy-agents can handle 9.65, 33.7 and 48.25 requests per second, respectively.

The FORW micro-kernel represents an implementation of the Forwarding pattern. This micro-kernel seeks to capture the overhead that arises when a mobile agent receives incoming mobile agents and re-routes them to other places. To this end, we use a “forwarding” mobile agent parked at a particular place. The forwarding agent “listens” for incoming agents; upon arrival of a new agent, the forwarding agent directs it to another place. The FORW benchmark is parameterized with respect to the total number of mobile agents handled and re-routed by the forwarding agent. We use one dispatching and one destination place only and measure the total elapsed time from the receipt of the first agent to the dispatch of the last one from the forwarding agent. The forwarding capacity of each MA platform converges to a certain sustained rate of requests served per second (see Figure 5, right). Voyager and Aglets can forward 19.84 and 9.54 agents per second respectively, whereas the corresponding number for Concordia is 5.76.

6 Conclusions

In this paper, we introduced a hierarchical framework for the quantitative performance evaluation of mobile-agent middleware platforms. We specified this framework as a hierarchy of benchmarks designed to enable the performance characterization of key components of MA middleware, and analyzed the performance of important classes of MA applications. This hierarchy is defined along a
number of dimensions pertinent to MA systems: the basic elements of MA platforms, distributed computing models of relevance, expected application frameworks, the context of MA execution, and expected workload characteristics. We proposed a set of micro-benchmarks and micro-kernels to implement the lower two levels of our benchmark hierarchy. We implemented these benchmarks in three of Java-based, mobile-agent middleware environments (Mitsubishi’s Concordia, IBM Aglets, and Objectspace’s Voyager). We presented results from experiments conducted to validate our framework and compare the mobile-agent middleware environments quantitatively.

To our knowledge, our framework provides the first structured and layered approach for analyzing the performance of MA middleware quantitatively [extensive coverage of related work is given in [5]]). Experiments with our micro-benchmark and micro-kernel suite provide a corroboration of this approach. Experimental results help us isolate the performance characteristics of MA platforms examined and lead us to the discovery of basic performance properties of MA systems. Furthermore, they provide a solid base for the assessment of the design choices made by middleware developers, from a performance perspective. For instance, our experimental results show that caching of classes and object re-use can lead to significant performance improvements and, therefore, call for a more in-depth study of techniques for their easy integration and optimization in MA middleware design and application development. Raw performance data show that agents cannot sustain the loads expected to arise in Internet middleware where places and agents could face workloads on the order of hundreds or thousands of requests per second, in the form of incoming messages, agents, etc. Furthermore, all examined platforms exhibit problems of robustness and performance scalability under high-loads, which are issues of critical importance for Internet services and applications. In such cases, places and agents should incorporate support for memory and resource management, request scheduling, recovery, high-performance execution of bytecodes, etc. Last, but not least, our approach can provide a basis for the development of performance prediction models and tools for mobile-agent systems.

References


Scheduling Multi-task Agents*

Rong Xie¹, Daniela Rus¹, and Cliff Stein²

¹ Department of Computer Science, Dartmouth College
{rong, rus}@cs.dartmouth.edu
² Department of IEOR, Columbia University
cliff@ieor.columbia.edu

Abstract. We present a centralized and a distributed algorithm for scheduling multi-task agents in a distributed system with the objective of minimizing the overall application completion time. Each agent consists of multiple tasks that can be executed on multiple machines which correspond to resources. The machine speeds and link transfer rates are heterogeneous. Our centralized algorithm has an upper bound on the overall completion time and is used as a module in the distributed algorithm. Extensive simulations show promising results of the algorithms, especially for scheduling communication-intensive multi-task agents.

1 Introduction

A mobile agent system is a single, unified framework for implementing distributed applications. Each distributed application can be implemented as a multi-task agent where there are possible precedence constraints and data transfers among the constituent tasks. The mobile agent executes by migrating from machine to machine, looking for data and resources according to each of its tasks.

A key component of any mobile agent system is controlling how the agents access the resources. Such resources may include CPU time, disk space, database access, etc. and may be provided by many machines in the network. For example, consider implementing a multi-step information retrieval as a multi-task mobile agent. The mobile agent will travel to a remote database to run a query with user-specified filters. The agent will then summarize locally the relevant results into a small number of topics or features. Using these features, the mobile agent will travel to a different database and register a persistent query, returning back to the user only after a set number of hits has been registered. If this database is replicated, the agent would have to choose which site to visit. The decision

*This work has been supported in part by Department of Defense contract MURI F49620-97-1-0382 and DARPA contract F30602-98-2-0107, ONR grant N00014-01-1-0675, NSF CAREER award IRI-9624286, NSF award IIS 9912193, Honda corporation, and the Sloan foundation; we are grateful for this support. We are also grateful to Bob Gray and Susan McGrath for their efforts in developing the scheduling experiments using the D’Agents system. We thank Miguel Fernandez and Santiago Aja for their implementation of the scheduling system on top of the D’Agents.

© Springer-Verlag Berlin Heidelberg 2001
depends on the general network traffic conditions, and the machine load and speed at the site of the database.

Since mobile agents move around in the network, often carrying variable size of data with them, the performance of an agent can be affected largely by data transfer delays, especially in heterogeneous networks with diversified network links. Thus, for scheduling a multi-task agent, there is a tradeoff between the amount of utilized parallelism in the agent and the amount of data transfer overhead incurred.

In this paper, we study the problem of scheduling multi-task agents in heterogeneous networks with the objective of optimizing the overall application completion time. Many assumptions used in traditional scheduling algorithms become unrealistic in this case. In general, scheduling algorithms for a mobile agent system must work in a heterogeneous environment where (1) the number of machines is limited; (2) precedence constraints are general; (3) data transfer delays are general; and (4) task duplication are not allowed. This problem is NP-Complete. In this paper, each agent consists of multiple tasks with precedence constraints, hence can be naturally modeled as a DAG (Direct Acyclic Graph). Both centralized and distributed scheduling algorithms are presented. In the centralized case, we present the FB and PFB algorithms which in a simplified case have a provable performance upper bound. In the distributed case, multi-task agents arrive over time. A distributed scheduling framework is proposed in which each multi-task agent is assigned its own scheduler which uses the PFB results as a module. Extensive simulations show promising results of the algorithms, especially for scheduling communication-intensive multi-task agents.

2 Problem Model

We represent each agent as a distributed application with a set of tasks among which there are possible precedence constraints and data transfers. This suggests using a DAG as representation. An instance of the agent (or, more generally the distributed application) is specified as a DAG $G = (T, E)$, where the set of nodes $T = \{T_1, T_2, \ldots, T_n\}$ denotes the set of tasks to be executed and the set of weighted, directed edges $E$ represents both precedence constraints and data transfers among tasks in $T$. The existence of an edge $(T_i, T_j) \in E$ implies $T_j$ cannot start execution until $T_i$ finishes and sends its result to $T_j$. In this case, we use $d(T_i, T_j)$ to denote the volume of data $T_i$ sends to $T_j$. Let $\text{Pred}(T_i)$ denote the set of all the immediate predecessors of task $T_i$.

Let $M = \{M_1, M_2, \ldots, M_m\}$ be the set of machines across the network. We assume each pair of machines are connected to each other and $r(M_i, M_k)$ represents the data transfer rate between machine $M_i$ and $M_k$. Since there is no communication delay for transferring data between two tasks on the same machine, we define $r(M_i, M_k)$ to be infinity. The processing time of task $T_i$ on machine $M_k$ is denoted by $p(T_i, M_k)$, which could be set to infinity if $T_i$ can not be executed on $M_k$. 
The objective of the scheduling problem is to find an assignment map $M : \{T_1, \cdots, T_n\} \rightarrow \{M_1, \cdots, M_n\}$ and a set of starting times $s(l(T_i))$, $i = 1, \cdots, n$, where each task $T_i$ is scheduled to be processed on machine $M(T_i)$ starting at time $s(l(T_i))$, such that the precedence constraints are satisfied and the schedule length $C_{max}$ is minimized. Here $C_{max}$ is the overall duration of the schedule defined as

$$C_{max} \triangleq \max_{1 \leq i \leq n} f(t(T_i)) = \max_{1 \leq i \leq n} (s(l(T_i)) + p(T_i, M(T_i)))$$

where $f(t(T_i))$ is the finish time of $T_i$.

Many approximation algorithms and heuristics have been proposed for DAG scheduling. Many of them assume the data transfer delay is negligible compared with the task execution time. For those considering data transfer delay, most of the results are purely empirical [10,1,3], or have various assumptions that do not hold for realistic applications, such as allowing task duplication to avoid long data transfer delays and assuming unlimited number of machines [8], restricting the structure of task graph [7], assuming globally small data transfer delay [6] or locally small data transfer delay [2].

3 Centralized Scheduling for a Multi-task Agent

In this section, we propose two scheduling algorithms for a multi-task agent: the forward-backward (FB) dynamic priority algorithm and the partial forward-backward (PFB) dynamic priority algorithm. Both FB and PFB are based on a basic greedy algorithm illustrated in Section 3.1, though they can be combined with many other scheduling algorithms to enhance their performances as well.

3.1 Basic Scheduling

In the basic algorithm, an agent consisting of $n$ tasks in $T$ is scheduled in $n$ steps, one task at a time. Intuitively, if one task can start executing on one machine at the earliest time and with the fastest speed, we schedule this particular task on this particular machine. However, it is possible that by waiting a bit the task can be executed on a faster machine. Therefore, we select the best task-machine pair at each scheduling step by weighing two parameters: the time and speed at which one task can be executed. Fig. 1 presents our basic scheduling algorithm. This algorithm is inspired by the DLS algorithm presented in [10].

Let $S_t$ be the system state at scheduling step $t$, which reflects the partial schedule information up to step $t$. $S_t$ consists of the subset of $T$ of all the tasks which have been scheduled before step $t$ together with the machines they are assigned to and the scheduled starting times. At scheduling step $t$, task $T_i$ is called ready if it is not scheduled yet and all of its predecessors have been scheduled. Let the set of all ready tasks be $R$.

At each scheduling step $t$, we define the data available time $DA(T_i, M_k)$ of a ready task $T_i \in R$ on machine $M_k$ as the earliest time when all the data sent to task $T_i$ from its predecessors is available at machine $M_k$: 
\[ DA(T_i, M_k) \triangleq \max_{T_j \in \text{Pred}(T_i)} \left[ ft(T_j) + \frac{d(T_j, T_i)}{r(M(T_j), M_k)} \right], \quad T_i \in R, \ 1 \leq k \leq m. \]

In other words, \( DA(T_i, M_k) \) reflects how soon all the data passed from \( T_i \)'s predecessors can arrive at machine \( M_k \). The machine available time \( MA(M_k, S_i) \) for each machine \( M_k \) is the time when all the tasks assigned to \( M_k \) so far finish processing. \( MA(M_k, S_i) \) is defined to be 0 if no task has been assigned to \( M_k \).

**Algorithm 1 (Basic Algorithm)**

1. Initialization: Let the set of ready tasks \( R \) be the set of entry tasks in \( T \), i.e. those tasks with no predecessors.
2. At each scheduling step \( t \), do:
   - For each pair of machine \( M_k \) and ready task \( T_i \) where \( T_i \in R, \ 1 \leq k \leq m \), compute its dynamic priority \( DP(T_i, M_k, S_i) = \max\{DA(T_i, M_k), MA(M_k, S_i)\} + c \times \frac{p(T_i, M_k)}{\max_{1 \leq j \leq m} \{p(T_i, M_j)\}} \) \hfill (1)
   - Find the task-machine pair \( (T_i^*, M_k^*) \) such that \( DP(T_i^*, M_k^*, S_i) = \min_{T_i \in R, 1 \leq k \leq m} DP(T_i, M_k, S_i) \)
   - Schedule task \( T_i^* \) on machine \( M_k^* \) after the last scheduled task on this machine.
   - Let \( t = t + 1 \). Update \( R \) and \( S_i \).
   - Terminate if \( R = \emptyset \).

---

**Fig. 1.** Basic algorithm

The max term in equation (1) (see Fig. 1) represents the earliest time task \( T_i \) can begin execution on machine \( M_k \) if \( T_i \) is scheduled on \( M_k \). The second term reflects how fast task \( T_i \) can be executed on machine \( M_k \). Since the execution time for one fixed task could be very different on different machines, we use this term to represent the relative efficiency of different machine-task combinations. The weight \( c \) is used to boost the weight of the second item in order to achieve a good compromise between these two criteria. The choice of \( c \) is currently experimental and deserves further study. In the case when two task-machine pairs have identical \( DP \) value, ties are broken by choosing the pair in which the task has a higher bottom-level, where the bottom-level of a task is defined as the largest sum of execution times along any path from this task to any exit task.

Notice that for each specific pair of ready task and machine, its \( DP \) value is different at different scheduling steps, and is nondecreasing with the increase of the scheduling steps. Hence algorithm 1 is a dynamic priority scheduling algorithm.
3.2 FB Scheduling

There are situations in which the basic algorithm may generate very unsatisfactory schedules. Fig. 2 shows such an example.

The key structure in the agent's task DAG that causes performance degradation is the small triangle formed by tasks $T_2, T_4, T_6$, where $T_6$ requires large volume of data from $T_2$ and $T_4$, respectively. This is an important scenario, as it captures many mobile agent applications which perform information gathering and retrieval. Due to the greedy nature of the basic algorithm, when $T_2$ and $T_4$ are considered for scheduling, the scheduler only evaluates the quantities of data transferred to ready tasks $T_2$ and $T_4$, no consideration is given to the large data transfers from them to their common successors $T_6$. So the basic algorithm fails to assign $T_2$ and $T_4$ to the same machine, hence at least one of the two large data transfer delays must occur. In general, as long as the task graph contains structures where the data transfered to a single node from its multiple predecessors are all very large compared with task execution times, similar performance degradation will occur.

Fig. 2. In this scenario we have two identical machines and the time needed to transfer $d$ units of data between any two machines is $d$ units of time. The weight of each node denotes the task execution time, and the weight of each edge denotes the volume of data to be transferred. The subgraph in (b) shows the optimal schedule, while subgraph in (c) shows the schedule generated by the basic algorithm, which is considerably longer than the optimal.

To remedy this situation, we can enhance our scheduler by taking advantage of the forward-backward symmetry of the problem. Specifically, we define the inverse version of a given multi-task agent scheduling problem $G = (\mathcal{T}, \mathcal{E})$ as $\hat{G} = (\mathcal{T}, \hat{\mathcal{E}})$, where $\hat{\mathcal{E}} = \{(T_i, T_j) | (T_j, T_i) \in \mathcal{E}\}$. The task graph of the inverse problem is the same as the original one except the direction of each edge, i.e. the precedence relation, is inverted.

**Proposition 1** The inverse problem and the original problem have the same minimal makespan.

**Proof.** Proof omitted for space considerations.
In the inverse problem, the data transferred from ready tasks in the original problem becomes data transferred to ready tasks, thus can be evaluated by the scheduler. This suggests that we can run Algorithm 1 on the inverse problem, then reverse the generated schedule (which is a feasible schedule for the inverse problem) to get a feasible schedule for the original problem. Fig. 4 summarizes this algorithm which we call Forward Backward (FB) dynamic priority scheduling. For the motivating example in Fig. 2, FB generates the optimal schedule shown in subgraph (c) of Fig. 3.

**Algorithm 2 (FB algorithm)**

1. Run the basic algorithm (Algorithm 1) on original problem \( G = (T, E) \), get schedule \( S \);
2. Run the basic algorithm on inverse problem \( \overline{G} = (\overline{T}, \overline{E}) \), reverse the generated schedule to get a feasible schedule \( S' \) for the original problem;
3. If \( C_{\text{max}}(S) < C_{\text{max}}(S') \), output \( S \), otherwise output \( S' \).

**Fig. 3.** A motivating example for extension

**Fig. 4.** FB algorithm

### 3.3 PFB Scheduling

Certain substructures in the multi-task agent's DAG enable the performance improvement of FB over the basic algorithm, particularly those “bad” in-tree structures where the data transferred to a single node from its multiple predecessors are all very large. By reversing the DAG, these in-tree structures will become “bad” out-trees (the data transferred from a single node to its multiple successors are all very large) and will be easily handled by the basic algorithm. However, when the DAG contains both bad in-trees and bad out-trees, the FB algorithm may fail to generate good schedules, since the forward or backward scheduling alone cannot handle both types of “bad” subgraphs simultaneously. Consider the example shown in Fig. 5. Both of the schedules generated by forward and backward scheduling suffer one long data transfer delay (100). The x-structure contains both the bad in-tree (ABC) and bad out-tree (CDE), which cause the considerable performance degradation. The bad in-trees and bad out-trees can also be independent of each other, as is shown in Fig. 6.

One natural solution is to use backward scheduling only on those parts of the DAG containing bad in-trees and forward scheduling on the remainder of the DAG, then assemble these two partial schedules together to get the final one. Partitioning the DAG optimally and efficiently is difficult. Fig. 7 shows
our solution which we call the partial forward backward (PFB) dynamic priority scheduling algorithm.

For the example shown in Fig. 5, the partial backward scheduling is implemented by first reversing the part of the schedule for C, D and E generated by forward scheduling to get a partial schedule $S_1$ of the inverse DAG, in which C, D and E are scheduled on the first machine and start at time 2, 1, 0 respectively. Then tasks A and B are backward scheduled on the same machine as their predecessor C in the reversed DAG. Reversing the schedule for the inverse DAG, we get an optimal schedule for the original problem.

![Fig. 5. A bad case for the FB algorithm.](image1)

![Fig. 6. Another bad case for the FB algorithm.](image2)

**Algorithm 3 (PFB algorithm)**

1. Run the basic algorithm (Algorithm 1) on original problem $G = (T, E)$, get schedule $S$.
2. Let $S' = S$. For each task $T_j \in T$, do:
   - Reverse the part of schedule $S'$ consisting of those tasks starting after time $\max_{T_k \in \text{Pred}(T_j)}(f(T_k))$ in $S'$ to get a partial schedule $S_1$, which is a schedule for those tasks in the inverse DAG.
   - Starting from $S_1$, run Algorithm 1 on the inverse DAG for the remaining tasks to generate a complete schedule $S_2$ for the inverse DAG.
   - Reverse $S_2$ to get a schedule $S''$.
   - If $C_{max}(S'') < C_{max}(S')$, let $S' = S''$.
3. Output $S'$.

![Fig. 7. PFB algorithm](image3)

This scheduling process is demonstrated in Fig. 8.

### 3.4 Performance Analysis

In this section, we present an upper bound for a simplified version of the basic algorithm in the computing environment in which machines are identical, but
communication links differ. This is a salient feature of agent systems. In this situation, the second term in equation (1) becomes identical for all task-machine pairs, and thus can be ignored in evaluating the dynamic priority. The basic algorithm becomes the first-start-pair-first algorithm, i.e. the starting times of successively scheduled tasks are a non-decreasing sequence in time. Our basic algorithm generates the same schedule as the ETF algorithm in [4]. Our analysis is inspired by [4], but is much simpler.

We associate with each scheduling step a a time \( \gamma_q \) which is the DP value of the task-machine pair selected at that step, i.e. the starting time of the task scheduled at step \( q \). We assume scheduling step \( q \) starts and completes instantly at time \( \gamma_q \). Thus, saying that a task is ready-to-schedule at scheduling step \( q \) implies that a task is ready-to-schedule at time \( \gamma_q \).

For scheduling problem \( G = (T, E) \), let the schedule generated by the basic algorithm be \( S \). As defined before, \( sl(B) \) and \( ft(B) \) are the starting and finish time of task \( B \) in schedule \( S \), respectively.

**Lemma 1.** In schedule \( S \), for every machine \( M_i \) and any task \( B \) such that \( DA(B, M_i) < sl(B), M_i \) is busy during the time interval \([DA(B, M_i), sl(B)]\).

**Proof.** Suppose otherwise \( M_i \) is idle during interval \([s, s + \Delta s] \subseteq [DA(B, M_i), sl(B)]\) (see Fig. 9). Let \( C \) be the first task scheduled on \( M_i \) after \( s + \Delta s \), then \( C \neq B \) (otherwise the algorithm will schedule \( B \) at a time no later than \( s \)). Furthermore, task \( C \) must be scheduled before \( B \), for if \( B \) is scheduled at a step \( p \) when \( C \) has not been scheduled, then \( MA(M_i, S_p) \leq s \), which together with \( DA(B, M_i) \leq s \) implies \( DP(B, M_i, S_p) \leq s < sl(B) \), a contradiction.

![Fig. 9. Proof of Lemma 1](image)

At the scheduling step \( q \) when \( C \) is being scheduled, \( B \) must be ready since it has not been scheduled and its data has been available since \( DA(B, M_i) \leq s < sl(C) = \gamma_q \). Moreover, \( MA(M_i, S_q) \leq s \), for \( M_i \) has been idle at least since time \( s \). So \( DP(B, M_i, S_q) = \max\{DA(B, M_i), MA(M_i, S_q)\} \leq s < sl(C) = DP(C, M_i, S_q) \). Therefore \( B \) instead of \( C \) should be scheduled at this step, contradiction.
Let $B_k$ be the last task finishing in schedule $S$. Choose any chain $L = B_K \rightarrow \cdots \rightarrow B_2 \rightarrow B_1$ in $G$ starting from some entry task $B_K$ and ending at $B_1$. Denote the length of the schedule $S$ as $C_{\text{max}}$, and the optimal schedule length ignoring data transfer delays as $C^*_{\text{max}}$.

**Theorem 1.** For scheduling problems with identical machines and general communication links,

$$C_{\text{max}} \leq (2 - \frac{1}{m}) C^*_{\text{max}} + D, \quad (2)$$

where

$$D = \sum_{k=1}^{K-1} \left[ \frac{1}{m} \sum_{j=1}^{m} DA(B_k, M_j) - ft(B_{k+1}) \right].$$

**Proof.** Define $t_{idle}$ to be the sum of idle time on all machines before time $C_{\text{max}}$ in schedule $S$. Similarly $t_{busy}$ is the sum of busy time on all machines before time $C_{\text{max}}$. Hence $t_{idle} + t_{busy} = m C_{\text{max}}$. Since $B_K$ has no predecessors, all machines must be busy before time $st(B_k)$, so

$$t_{idle} \leq (m-1) \sum_{i=1}^{K} p(B_i) + \sum_{k=1}^{K-1} \sum_{j=1}^{m} (st(B_k) - ft(B_{k+1})).$$

Since $C^*_{\text{max}}$ is no smaller than the sum of execution time along any chain, and by Lemma 1, every machine $M_i$ must be busy during the time interval $[DA(B_k, M_j), \ st(B_k)]$ if $DA(B_k, M_j) < st(B_k)$ for $i = 1, \ldots, K$, we have

$$t_{idle} \leq (m-1) C^*_{\text{max}} + \sum_{k=1}^{K-1} \sum_{j=1}^{m} [DA(B_k, M_j) - ft(B_{k+1})]$$

(3)

Therefore, $C_{\text{max}} = \frac{1}{m} (t_{idle} + t_{busy}) \leq C^*_{\text{max}} + \frac{1}{m} t_{idle}$ which together with (3) completes the proof of Theorem 1.

Among all the chains satisfying the conditions preceding Theorem 1, we can define a particular one $\bar{L}$ as follows: Let $\bar{L}_1 = B_1$. Fixing the starting times and the associated machines of all tasks in $\text{Pre}(\bar{B}_1)$ as in schedule $S$, choose $\bar{i}_1$ such that $DA(\bar{B}_1, M_{\bar{i}_1}) = \max_{1 \leq i \leq m} DA(\bar{B}_1, M_i)$, and let $\bar{B}_2$ be the immediate predecessor of $\bar{B}_1$ whose data for $\bar{B}_1$ arrives last at machine $M_{\bar{i}_1}$. So

$$f_{\bar{B}_2} + d(\bar{B}_2, \bar{B}_1) + v(M(\bar{B}_2), M_{\bar{i}_1}) = DA(\bar{B}_1, M_{\bar{i}_1}).$$

Inductively define $\bar{B}_k, i = 3, \ldots, K$ in this way, until reaching an entry task $\bar{B}_K$. For this particular chain $\bar{L}$, Theorem 1 becomes:

**Corollary 1.** In equation (2), $D$ can be written as

$$D = \sum_{i=1}^{K-1} d(\bar{B}_{i+1}, \bar{B}_i) \leq \sum_{i=1}^{K-1} d(\bar{B}_{i+1}, \bar{B}_i) \leq \sum_{i=1}^{K-1} d(\bar{B}_{i+1}, \bar{B}_i), \quad (4)$$

where $v_{\text{min}}$ is the speed of the slowest link.
3.5 Experimental Results

In this section, we present simulation results for our two multi-task agent scheduling algorithms and compare their performances with the DLS algorithm of [10]. DLS is one of the few task scheduling algorithms that supports general computation and data transfer delay in heterogeneous domains.

Our simulations are run on two sets of task graphs: random DAGs with predetermined optimal schedules proposed in [5] and random DAGs with unknown optimal schedules. We define ACCR (Average Communication to Computation Ratio) of a distributed application as the average communication (data transfer) delay divided by the average computation time of tasks. The parameter c in equation (1) is set to be 10.

Fig. 10 shows the comparison result of running simulations on random DAGs with predetermined optimal schedule length in homogeneous environment. Both of the FB and PFB algorithms generate considerably better schedules than DLS, especially for communication-intensive applications.

![Fig. 10. Random DAG with pre-determined optimal schedule length 100.0. The x-axis represents the ACCR; the y-axis represents the average schedule length (averaged over 60 simulation runs) color-coded for each of the three algorithms. Seven different values of ACCR were selected: 0.1, 0.25, 0.5, 1, 2, 4, 6, 8, 10, to show the relative performance over a range of distributed applications from computation-intensive ones (when ACCR is small) to communication-intensive ones (when ACCR is large).](image)

Fig. 11 gives the average speedup when running simulations on random DAGs with unknown optimal schedules in heterogeneous environments. Here the speedup of algorithm A over algorithm B means the ratio of the schedule length generated by algorithm B to that generated by algorithm A. One hundred random DAGs were generated as test bed: the number of tasks in each DAG is of uniform distribution over [20,100], the average task execution times, average data transfer delays, machine speeds and link speeds are uniformly distributed over different ranges. The results shown in the graphs are an average over 100 separate simulations.

Fig. 11(a) shows the average speedup of our two algorithms over DLS with respect to machine heterogeneity, when the average task execution time, average data transfer delays were uniformly distributed over [1.0,9.0] and [0.0,40.0], respectively. The speed of each machine is of uniform distribution over the range $[\frac{1}{4} \cdot p_u, \beta \cdot p_u]$, i.e. a large value of $\beta$ indicates a high heterogeneity of machine speeds. Seven different values of $\beta$ ranging from 1.25 to 20 are selected to indicate
different level of heterogeneity of machine speeds. The link rates vary uniformly over $[0.5 \times \text{average rate}, 1.5 \times \text{average rate}]$. Our algorithms outperform DLS, and this performance improvement gets more evident as the range in which the machine speeds vary increases. We also observe a significant improvement of PFB over FB.

Fig. 11(b) shows the speedup of our algorithms over DLS with respect to ACCR, where the average task execution time, link rates and machine speeds are of uniform distribution over $[1.0, 9.0]$, $[0.5 \times \text{average link rate}, 2.0 \times \text{average link rate}]$ and $[0.2 \times \text{average machine speed}, 5.0 \times \text{average machine speed}]$, respectively. When $ACCR < 1$, DLS slightly outperforms our algorithms, but when $ACCR \geq 1$, our algorithms outperform DLS considerably. A significant improvement of PFB over FB is also observed.

4 Distributed Scheduling for Many Multi-task Systems

In a mobile agent system, multi-task agents arrive over time. In this section we use the ideas from scheduling a single multi-task agent to schedule many multi-task mobile agents in a distributed system. We propose a distributed scheduling framework by assigning to each such agent its own scheduler (resource manager), which uses Algorithm 4 for scheduling.

4.1 Model

We assume that each agent has its own scheduler, called the agent scheduler. We assume there is no communications between different agent schedulers, thus each agent scheduler works independently without cooperation. An agent scheduler takes a snapshot of the system state and makes scheduling decisions for the agent’s tasks dynamically. Since multiple agents execute in the system, the actual starting time of a task may be different from the one computed by the agent scheduler. Thus, the notion of scheduling here is slightly different from what we have used in centralized scheduling, in that an agent scheduler does not specify the absolute starting time of each task.
Algorithm 4 (Distributed Algorithm)
1. Run Algorithm 3 on the multi-task agent to get its PFB schedule $S_0$.
2. Initialize the set of ready tasks $R$ as the set of entry tasks in $T$, i.e. those tasks with no predecessors.
3. While not all tasks of the agent have been scheduled, do:
   a) Update $R$.
   b) While $R \neq \emptyset$, do:
      a) For each pair of task $T_i$ and machine $M_k$, where $T_i \in R$, $1 \leq k \leq m$, compute its dynamic priority from equation 1.
      b) Find the task-machine pair $(T_{i'\ast}, M_{k\ast})$ such that $DP(T_{i'\ast}, M_{k\ast}) = \min_{T_j \in R, k \leq m} DP(T_j, M_k)$.
      c) If the Average Communication-to-Computation Ratio of the DAG is larger than $\lambda$, do: for each already scheduled task $A$ that (1) has common successors with $T_{i'\ast}$; (2) is assigned on the same machine as $T_{i'\ast}$ in $S_0$; (3) the minimum of the data transfer delays from $A$ and $T_{i'\ast}$ to their common successor is $\alpha$ times larger than the maximum of the standard execution times of $A$ and $T_{i'\ast}$, set $M_{k\ast}$ as the machine that $A$ is assigned to.
   d) Schedule task $T_{i'\ast}$ on machine $M_{k\ast}$.

Fig. 12. Distributed scheduling

For incoming agent tasks, each machine has two specific FIFO queues: a waiting queue and a ready queue, both of which are manipulated by a local "coordinator" agent residing on this machine. An incoming agent task is allocated to the ready queue or the waiting queue depending on whether its input data is available on this machine or not. For those tasks in the waiting queue, they can be reallocated to the ready queue by the coordinator agent at a later time when all its input data has arrived on this machine. Thus the tasks in the ready queue can start running instantly once the machine becomes idle, while the waiting queue consists of those tasks waiting for the arrivals of their input data. The coordinator agent is also responsible for notifying each agent scheduler when one of its tasks starts or finishes execution. By implementing these two queues on each machine, the tasks which are scheduled early but whose input data arrive very late will not block other tasks which are scheduled later but whose input data come earlier.

4.2 Distributed Scheduling

In a distributed mobile agent system, different multi-task agents arrive over time. Thus one factor that can affect the decisions of a scheduler is the time-varying machine states resulting from the arrival of tasks from other agents. Agent schedulers should dynamically, rather than statically, schedule their tasks
to take into account the time-varying system states affected by the incoming
tasks of other agents over time. An important issue in dynamic scheduling is
timing, i.e. when to schedule the tasks of an agent. The scheduling time of a
task can be as early as its agent's arrival time or as late as the time when the
task is ready to run. If we schedule a task early, a large part of the scheduling
and task submission overhead can be overlapped with the task computations and
communications of the agent, but the state information can be stale. So there is
a tradeoff between scheduling and task submission overhead, and the accuracy
of the state information used by the scheduler. In our algorithm, we choose the
scheduling time of a task to be the latest time among the starting times of all its
predecessors, i.e. the earliest time when the data available times of this task on
all machines can be calculated precisely. A task is ready to be scheduled when
all its predecessors have started executions.

The centralized algorithms we developed in the previous sections can be ex-
tended to the distributed case. Algorithm 1 is adaptive, hence can be easily
adopted by each agent scheduler. However, it can generate very unsatisfactory
schedules when there are bad in-trees and bad out-trees in the multi-task agent
DAG. On the other hand, the PFB algorithm can overcome this difficulty and
improve the performance considerably, but its extension to the distributed en-
vironment is not straightforward. Therefore, it is natural to use the scheduling
results of PFB algorithm as hints for the agent schedulers which utilize Algo-


4.3 Simulation Results

Simulations are carried out in a heterogeneous computing environment, where
the total number of machines is 16 and the machine speeds and link rates are
generated randomly. There are 32 multi-task agents arriving over time, where
the agent arrivals are given by a Poisson process. Each agent consists of 64 tasks,
whose structure is generated randomly under the constraint that the maximum
number of edges emitting from one task is 16.

In Fig. 13, we simulate our distributed scheduling algorithm to compare the
performances of Algorithm 4 using PFB hints versus not using PFB hints. We
choose the threshold $\lambda$ and $\alpha$ to be 4 and 2 respectively. The results of each
case are obtained by taking the average of five simulation runs. We observe
that, by using PFB hints, the distributed algorithm has significant performance
improvement when the ACCR is large, i.e. when scheduling multi-agent systems
that are communication intensive.
Fig. 13. The performance of scheduling multi-task multi agent systems. The x-axis is the ACCR. The y-axis is the speedup in the sum of the application turnaround time. The four curves correspond to different average agent arrival intervals.

The distributed algorithm assumes that the scheduler has full knowledge of the multi-task agent to be scheduled and the global information of the network, which is not realistic in most real systems. In many cases we are only able to feed the scheduler with the estimated values of parameters. So it is important to evaluate how tolerant the distributed scheduling algorithm is to the estimation errors of parameters. Three parameters are chosen to be tested individually: the standard task execution time, the size of data transferred among tasks, and the transfer rate of communication links. For each of them, three estimation error ranges are simulated, where the estimated parameters are uniformly distributed within ±10%, ±25%, ±50% of the correct values respectively. We define the degradation ratio as the ratio of the sum of the total application turnaround times using correct parameters to that using estimated parameters. We use this ratio to indicate the tolerance of our algorithm to parameter variations, where a degradation ratio far below 1.0 means that the algorithm is very sensitive to parameter variations. We evaluate the mean and the standard deviation of the degradation ratio under different average communication-to-computation ratios by averaging over twenty simulation runs. Fig. 14 and Fig. 15 show the simulation results. The average application arrival interval is 100. We observe that the algorithm is more sensitive to the data sizes and link rates than to the standard task execution times. The overall degradations are acceptable, where the worst-case performance degradation is 20%. When ACCR is large, we observe large standard deviations.

4.4 An Experiment

We are currently implementing our distributed mobile agent scheduling algorithms in the context of a multi-step information retrieval which is a component of the MURI application described in [9]. The scheduling algorithm has already been implemented on top of the D’Agents mobile agent system. The main components of the implementation consist of the scheduling modules used in our simulations and modules used to estimate network delay, machine load, and machine speed for each of the machines in the system. We hope to collect data consistent with our simulations for multi-step information retrievals in the near future.
5 Conclusions

We presented a solution to distributed multi-task multi-agent scheduling for mobile agent environments with heterogeneous hosts and communication delays. We approached this problem by first developing a centralized algorithm for scheduling a single multi-task agent. An upper bound is provided for this algorithm in a simplified case when all the machines in the system are identical but the communication delays vary. We then extend this algorithm for the distributed multi-agent problem, by associating a scheduler with each agent. Extensive simulation results show that the proposed algorithm is promising, especially for the distributed scheduling of communication-intensive multi-task agents.

Fig. 14. The x-axis represents the ACCR; the y-axis represents the mean of the degradation ratio. Tested parameter: (a) Standard task execution time; (b) Size of data transferred among tasks; (c) Transfer rates of communication links.

Fig. 15. The x-axis represents the ACCR; the y-axis represents the standard deviation of the degradation ratio. Tested parameter: (a) Standard task execution time; (b) Size of data transferred among tasks; (c) Transfer rates of communication links.

References

Author Index

Agha, Gul 119
Barton, Joyce 229
Bettini, Lorenzo 182
Bic, Lubomir F. 88
Binder, Walter 213
Bradshaw, Jeffrey 229
Brandt, Raimund 70
Breedy, Maggie 229
Cao, Jiannong 135
Carbunar, Bogdan 54
Chacón, Daria 229
Chan, Henry 135
Chang, Po-Hao 119
De Nicola, Rocco 182
Dikaiakos, Marios 244
Dillencourt, Michael B. 88
Feng, Xinyu 135
Fischmeister, Sebastian 31
Gendelman, Eugene 88
Gerken, Peter 229
Gray, Robert S. 229
Haller, Klaus 152
Hannotin, Xavier 42
Hofmann, Martin 229
Illmann, Torsten 198
Jeffers, Renia 229
Kargl, Frank 198
Kemmerer, Richard A. 31
Kotz, David 229
Krueger, Tilman 198
Kyriakou, Melinos 244
Lü, Jian 135
Maggi, Paolo 42
Moreau, Luc 15
Peters, Jan 169
Peterson, Ronald A. 229
Reiser, Helmut 70
Roth, Volker 1,169
Rus, Daniela 260
Samaras, George 244
Satoh, Ichiro 103
Schuldt, Heiko 152
Sisto, Riccardo 42
Stein, Cliff 260
Suri, Niranjan 229
Tan, Hock Kim 15
Thati, Prasanna 119
Valente, Marco Tulio 54
Vigna, Giovanni 31
Villazón, Alex 213
Vitek, Jan 54
Weber, Michael 198
Xie, Rong 260