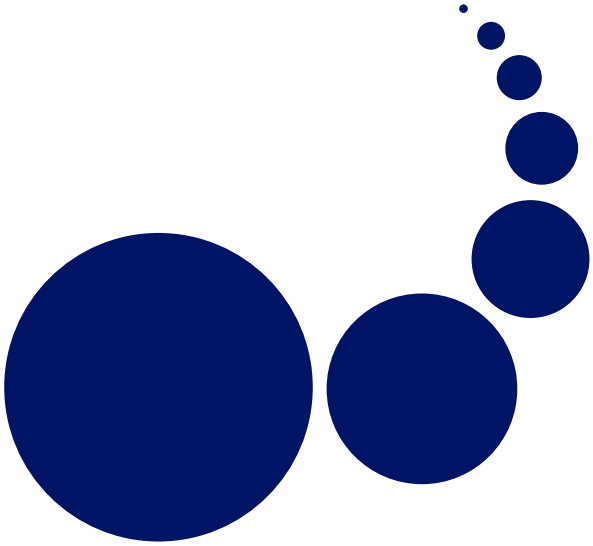
SCALABLE COMPUTING
Practice and Experience

Special Issue: Software Agent Mobility

Editors: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri



Volume 7, Number 4, December 2006

ISSN 1895-1767

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 7, Number 4, December 2006

TABLE OF CONTENTS

Editorial: Mobility in Distributed Systems **i**
Niranjan Suri, Henry Hexmoor

Guest Editor's Introduction **iii**
Henry Hexmoor, Marcin Paprzycki, Niranjan Suri

SPECIAL ISSUE PAPERS:

Security Risks in Java-based Mobile Code Systems **1**
Walter Binder and Volker Roth

Implementing Mobile and Distributed Applications in X-Klaim **13**
Lorenzo Bettini, Rocco De Nicola and Michele Loreti

Leveraging strong agent mobility for Aglets with the Mobile JikesRVM framework **37**
Raffaele Quitadamo, Letizia Leonardi and Giacomo Cabri

A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers **53**
Luc Moreau

Generative Mobile Agent Migration in Heterogeneous Environments **89**
B. J. Overeinder, D. R. A. de Groot, N. J. E. Wijngaards, and F. M. T. Brazier

A Distributed Content-Based Search Engine Based on Mobile Code and Web Service Technology **101**
Volker Roth, Jan Peters and Ulrich Pinsdorf

BOOK REVIEWS:

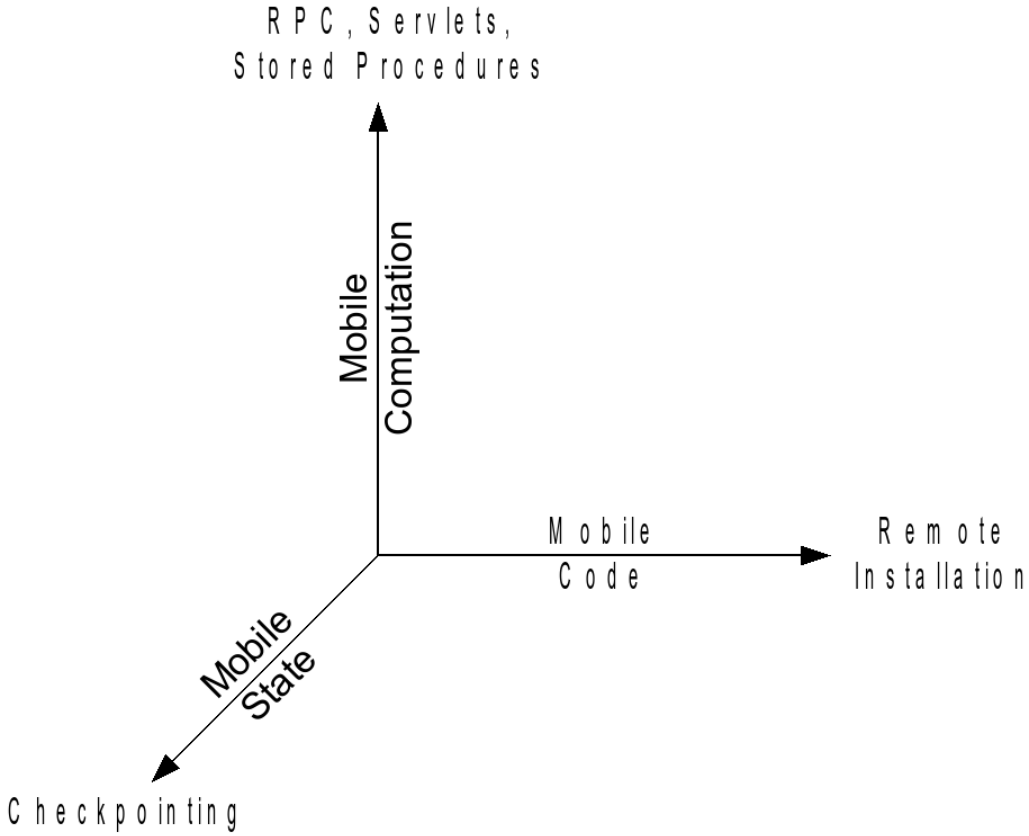
Expert Systems: Principles and Programming **119**

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. i–ii. http://www.scpe.org
ISSN 1895-1767

EDITORIAL: MOBILITY IN DISTRIBUTED SYSTEMS

Mobile agents are programs with the additional capability to move between computers across a network connection. Movement implies that the running program that constitutes an agent moves from one system to another, taking with the agent the code that constitutes the agent as well as the state information of the agent. The movement of agents may be user-directed or self-directed (i.e. autonomous). In the case of user-directed movement, agents are configured with an itinerary that dictates the movement of the agents. In the case of self-directed movement, agents may move in order to better optimize their operation. Mobility may also be a combination of user- and self-directedness.

Mobile agents provide three basic capabilities: mobile code, mobile computation, and mobile state. These three capabilities are shown in the figure below. Each of the capabilities is an evolution of previously well-established notions in distributed and networked systems.



Mobile computation involves moving a computation from one system to another. This capability is an evolution of remote computation, which allows a system to tap into the computational resources of another system over a network connection. One of the original mechanisms for remote computation was Remote Procedure Call (RPC). Java Remote Method Invocation (RMI) is another example of remote computation as are servlets and stored procedures.

The difference between mobile and remote computation is that mobile computation supports network disconnection. In a traditional remote computation model, the system requesting the service (the client) must remain connected to the system providing the service (the server) for the duration of the remote computation operation. Additionally, depending on the interface exposed by the server, an interaction can require an arbitrary number of messages between client and server. If network connectivity is lost, the remote computation will become an orphaned computation that will either be terminated or whose results will be discarded. A mobile computation, on the other hand, is an autonomous entity. Once the computation moves from the first system (which may nominally be called the client) to the second system (the server), the computation continues to execute on the server even if the client becomes disconnected. The agent returns to the client with the results of the computation when (and if) the connectivity is recovered.

Mobile Code is the ability to move code from one system to another. The code may be either source code that is compiled or interpreted or binary code. Binary code may further be either machine dependent or be some intermediate, machine-independent form.

Mobile code is used in other contexts besides mobile agents. For example, system administrators use mobile code in order to remotely install or upgrade software on client systems. Similarly, a web browser uses mobile code to pull an applet or script to execute as part of a web page.

Code may be mobile in two different ways: push and pull. In the push model, the system sending the code originates the code transfer operation whereas in the pull model, the system receiving the code originates the code transfer operation. An example of the pull model is a Web browser downloading components such as applets or scripts. Remote installation is an example of the push model. Mobile agent systems use the push model of code mobility.

Pull mobility is often considered to be more secure and trustworthy because the host receiving the code is the one that requested the code. Usually, the origin of the request lies in some action carried out by a user of the system and hence pull mobility is superficially more secure. Push mobility on the other hand allows a system to send code to the receiving system at unexpected or unmonitored times. Hence push mobility is less trustworthy from a users point of view. In practice the overwhelming majority of security exploits encountered in distributed systems originates in careless user actions such as running mail attachments.

Mobile code allows systems to be extremely flexible. New capabilities can be downloaded to systems on the fly thereby dynamically adding features or upgrading existing features. Moreover, if capabilities can be downloaded on demand, temporarily unused capabilities can also be discarded. Swapping capabilities on an as-needed basis allows systems to support small memory constrained devices. Discarding capabilities after use can also help improve system security.

Mobile state is an evolution of state capture, which allows the execution state of a process to be captured. State capture has been traditionally used for checkpointing systems to protect against unexpected system failure. In the event of a failure, the execution of a process can be restarted from the last checkpointed state thereby not wasting time by starting from the very beginning. Checkpointing is thus very useful for long-running processes. Operating system research has investigate capturing entire process states, a variant of checkpointing, for load balancing purposes in the early 1980s, but that avenue of research proved to be a dead-end due to coarse granularity of process and semantics problem due to the impossibility of capturing operating system resources such as open file descriptors.

Mobile state allows the movement of the execution state of an agent to another system for continued execution. The key advantage provided by mobile state is that the execution of the agent does not need to restart after the agent moves to a new host. Instead, the execution continues at the very next instruction in the agent.

Not all mobile agent systems provide support for state mobility. The term strong mobility is used to describe systems that can capture and move execution state with the agent. Operationally, strong mobility guarantees that all variables will have identical values and the program counter will be at the same position. Weakly mobile-agent systems, on the other hand, usually support the capture of most of a programs data, but restart the program from a predefined program point and thus require some programmer involvement at each migration. The advantage of strong mobility is that the result of migrating is well defined and easier to understand, but its disadvantage is that it is much more complex to implement efficiently.

The most important advantage provided by strong mobility is the ability to support external asynchronous migration requests (also known as forced mobility). This allows entities other than the agent (such as other system components, an administrator, or the owner) to request that an agent be moved. Forced mobility is useful for survivability, load-balancing, forced isolation, and replication for fault-tolerance, and is an important aspect of the agile computing approach.

For some applications, the technical issues of software mobility are dwarfed by security issues. Despite fits and starts, mobile agents herald emerging technological solutions. This special issue provides a good sampling of this frontier. It is our hope and desire that the software industry embrace the capabilities offered by mobile agents. In the future, we hope to see general purpose programming languages that support mobility as well as large scale development platforms and toolkits.

Niranjan Suri,
Institute of Human and Machine Cognition.
Henry Hexmoor,
Sourthern Illinois University.

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, p. iii. http://www.scpe.org
ISSN 1895-1767

GUEST EDITOR'S INTRODUCTION.

This issue is the first of a two issue collection of selected papers from the annual AIMS (Agents, Interactions, Mobility, and Systems) conference track spanning 2002-2006. AIMS began in 2002 as part of the ACM SAC (Symposium on Applied Computing) and continued for five years. The first conference was held in Madrid, Spain. Subsequent conferences were held in Melbourne (Florida, USA), Nicosia, Cyprus, Santa Fe, New Mexico, and Dijon, France. The track was primarily created to provide a venue for applied topics in software agents but became the primary venue for papers on mobile agents, as the IEEE conference on Mobile Agents was discontinued beyond 2002. Hence, the initial collection of six papers from the AIMS track focuses exclusively on topics related to mobile agents.

In the first paper, Binder and Roth examine the security capabilities of the Java platform with respect to supporting mobile agents. The paper raises the question as to whether Java is an appropriate language and has the necessary security mechanisms to safely support mobile agents. After highlighting the missing capabilities, the paper concludes by pointing to ongoing work that fortify shortcomings in Java.

The second paper, by Bettini, Nicola, and Loreti, describes their X-KLAIM programming language that is designed to simplify the task of writing mobile agents as well as distributed applications in general. X-KLAIM builds on the concept of tuple spaces as introduced by the Linda programming language. The paper presents examples of distributed applications and mobile agents that are written in the X-CLAIM language.

The third paper, by Quitadamo, Leonardi, and Cabri, describe an approach to provide strong mobility for agents. Strong mobility implies that the execution state of the treads of the agent are preserved, thereby allowing the agent to continue execution after reaching the destination node. Very few mobile agent systems are capable of supporting strong migration. Their approach takes advantage of a modified Jikes RVM (Research VM) from IBM and supports mobile agents written in the Aglets programming language.

In the fourth paper, Moreau addresses the problem of routing messages to agents that are roaming a network. His approach begins with a fault-tolerant directory service that is used to keep track of agent locations and messages being sent. Nodes use forwarding pointers when agents move and the system provides redundancy in order to handle failure of nodes.

The fifth paper, by Overeinder, de Groot, Wijngaards, and Brazier addresses the problem of agent migration across heterogeneous platforms. Their approach aims to provide cross-platform compatibility via authoring the mobile agent in a meta-language that provides a "blueprint" for the agent's behavior. Each platform then contains an agent factory that generates the code for the agent based on the blueprint.

Finally, in the sixth paper, Roth, Peters, and Pinsdorf describe an application of mobile agents to search for multimedia by content-matching. Instead of transmitting large multimedia objects to clients, a distributed search engine is used that embeds the features of interest into a mobile agent that is dispatched to the sources of the images. Content matching is then performed on the provider nodes, thereby reducing the need to unnecessarily transmit large amounts of data over the network.

Finally, we would like to express our gratitude to everyone who worked with us in the Program Committee of the AIMS Workshop. THANK YOU!

Henry Hexmoor,
Marcin Paprzycki,
Niranjan Suri.

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 1–11. http://www.scpe.org
ISSN 1895-1767

SECURITY RISKS IN JAVA-BASED MOBILE CODE SYSTEMS

WALTER BINDER* AND VOLKER ROTH†

Abstract. Java is the predominant language for mobile agent systems, both for implementing mobile agent execution environments and for writing mobile agent applications. This is due to inherent support for code mobility by means of dynamic class loading and separable class name spaces, as well as a number of security properties, such as language safety and access control by means of stack introspection. However, serious questions must be raised whether Java is actually up to the task of providing a secure execution environment for mobile agents. At the time of writing, it has neither resource control nor proper application separation. In this article we take an in-depth look at Java as a foundation for secure mobile agent systems.

Key words. Java, JVM, mobile agents, security, process isolation, resource management

1. Introduction. The proliferation of the Java programming language [18] led to the development of numerous mobile agent platforms. Actually, Java seems perfect for developing an execution environment for mobile agents, because Java offers many features that ease its implementation and deployment. Java runtime systems are available for most hardware platforms and operating systems. Therefore, mobile agent platforms that are built on Java are highly portable and run seamlessly on heterogeneous systems. Furthermore, mobile agents profit from continuous performance and scalability enhancements, such as increasingly sophisticated compilation techniques and other optimizations, which are provided by the underlying Java Virtual Machine (JVM) [23].

In addition to portable code, Java offers a *serialization* mechanism allowing to capture a mobile agent's object instance graph before it migrates to a different host, and to resurrect the agent in the new environment. Java also supports dynamic loading and linking of code by means of a hierarchy of *class loaders*. A class loader constitutes a separate *name space* that can be used to isolate classes of the agent system and of different agents from each other.

In general, mobile agent platforms execute multiple agents and service components concurrently in a time sharing fashion. Java caters for this need by means of multi-threading. Java is also a *safe* language, which means that the execution of programs proceeds strictly according to the language semantics (this is not entirely true, as we discuss in Section 2.3). For instance, types are not misinterpreted and data is not mistaken for executable code. The safety properties of Java depend on techniques such as *bytecode verification*, *strong typing*, *automatic memory management*, *dynamic bound checks*, and *exception handlers*. On top of that, the Java 2 platform includes a sophisticated security model with flexible access control based on dynamic stack introspection.

In summary, Java is highly portable and provides easy code mobility. This caused numerous mobile agent systems based on Java being developed and experimented with. From the point of security, two points can still be criticized: first, all systems focus on a particular aspect of agent mobility and none address all problems simultaneously, whose solution is required to come up with a system ready for field use. This is particularly true for security. Second, practical experience with Java shows that considerable security concerns remain, which are discussed in the next section.

This article, which is based on reference [8], is structured as follows: Section 2 discusses a series of shortcomings of current Java environments that reduce system stability in the presence of erroneous code and may be easily exploited for attacks by malicious mobile agents. These flaws can be overcome by a combination of isolation of mobile agents executing in a single JVM and resource control. While current standard JVM implementations do not address these issues, research has matured and several solutions have been suggested, also within the Java Community Process (JCP). Section 3 deals with issues of mobile agent isolation, whereas Section 4 addresses resource control. In both of these sections we compare the strengths and limitations of different approaches. Finally, the last section concludes this article.

2. Java Security Problems. In this section we give an overview of shortcomings in the JVM that hinder the development of secure and reliable mobile agent platforms. We discuss problems in the area of object management (Section 2.1), thread management (Section 2.2), bytecode verification (Section 2.3), as well as shortcomings of the Java security model (Section 2.4) and the lack of resource control (Section 2.5).

2.1. Object Management. A Java class is represented in the JVM by a *class object*. The class object that represents a class is initialized upon the first *active* use of that class. Mere declaration of a typed variable does not constitute active use and does not trigger initialization of the class that represents that type. However, as soon as a static method actually declared in that class is invoked, or a constructor is invoked, or a non-constant field is accessed, the class is initialized.

*Faculty of Informatics, University of Lugano, Lugano, Switzerland, walter.binder@unisi.ch
†FX Palo Alto Laboratory, Inc., 3400 Hillview Ave, Bldg 4, Palo Alto, CA, USA, volker.roth@acm.org

LISTING 1
Signature for a callback in the Serialization Framework.

```
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

LISTING 2
Example DoS attack on thread creation.

```
synchronized(Thread.class) {
  while (true);
}
```

When a Java class is initialized, first its class variable initializers and static initializers are executed. This opens a loophole for executing potentially malicious code before even the first instance is generated.

Further loopholes are hidden in the *Serialization Framework* of Java. During deserialization of an object instance, no constructors are invoked. The fields of unmarshalled objects are initialized directly. However, if the object that is unmarshalled implements a method of the exact signature given in listing 1 then this method is invoked in the deserialization process. A similar callback method exists for object serialization. This means that:

- agent state appraisal and authentication must complete before the agent instance is marshalled and the first agent class is loaded into the VM. Once this happens, it is already too late to defend against malicious agents.
- whenever an agent class is initialized, or an agent instance is marshalled or unmarshalled, the agent may take over the current thread and perform a variety of unexpected operations.

The loopholes described above potentially allow agents to run code before the system is prepared for it.

Another loophole in Java's garbage collector allows to "plant an egg" that is hatched after the agent has been terminated. When the VM detects that no strong references exist anymore to some object then it garbage collects this object and reclaims the memory occupied by the object. However, prior to that the garbage collector gives the object an opportunity to clean up any leftover state by invoking the finalizer of that object (if it is implemented). Consequently, if the method call does not terminate then no garbage is collected anymore and the VM eventually crashes from a lack of memory.

A less obvious attack would be to set an alarm (by means of a new thread) that triggers a destructive method only after a delay. In that case, the log files of the agent system (if there are any) show that the malicious agent already left the server and thus cannot possibly be responsible for the crash. At the very least, it becomes complicated to determine what actually happened and to prove it to someone else.

The developer of an agent system might be tempted to eliminate these loopholes by refusing to load any classes that implement the finalizer method. However, this is insufficient because several classes in the Java core packages already implement a finalizer and invoke additional callbacks in it. A malicious agent might, for instance, provide a class that inherits e.g., from *FileInputStream, FileOutputStream,* or *ZipFile*.

Any of these classes invokes method *close()* in its finalizer. Hence, rather than overriding the finalizer itself, malicious code may override the *close()* method. Consequently, all classes that inherit from one of these must be blocked as well (at least, if they implement *close()*).

Regardless how an agent becomes executed, once it runs it may hamper other threads and agents in a variety of ways. One option to launch a *denial of service* (DoS) attack is to synchronize on class locks. In Java all class objects of classes loaded by the system class loader are visible and some classes synchronize on their class locks. A simple DoS attack is illustrated in listing 2. If executed, no new threads can be generated because any attempt to increase the thread counter will lead to a deadlock situation (see also the relevant source code excerpt from the JDK 1.6 given in listing 3). Class locks can also be used to implement covert channels [22].

Clearly, touching an object is a dangerous thing. Yet, mobile agent systems often allow uncontrolled aliasing (sharing of object references), which is both convenient and typical of object-oriented programming. Again, DoS attacks can take on various forms. Catching the current thread is one possibility, keeping references to other agents' objects is another. As long as a strong reference to some object exists, it will not be garbage collected. In Java it is not possible to revoke an object reference. However, dynamic proxy generation mechanisms (which are available since JDK version 1.3) can be applied to this problem. This approach is taken e.g., in the SeMoA mobile agent system.[1]

[1] `http://www.semoa.org`

LISTING 3
Vulnerable code in the JDK 1.6.

```
private static long threadSeqNumber;

private static synchronized long nextThreadID() {
  return ++threadSeqNumber;
}
```

In summary, the concurrent execution of multiple agents requires isolation boundaries, where the passing of references has to be controlled. Marshalling and unmarshalling of objects must be done by a thread that can be sacrificed, or belongs already to a sandbox that is set up in advance for the object in question. Migration must take place only after all threads of a mobile agent have terminated and none of its classes is on the stack of any running thread anymore (or referenced by any object with a strong reference). Abuse of class locks is a matter that is addressed best by means of a modification of Java.

2.2. Thread Management. The Java language includes a set of APIs and primitives to manage multiple concurrent threads within a Java program. Synchronization between threads is based on *monitors*, which are associated with objects. Java *synchronized*{} statements are mapped to matching *monitorenter* and *monitorexit* instructions at the bytecode level. Monitors are implemented based on *locks*; each object has an associated lock that is used whenever a *synchronized*{} statement refers to that object. Methods of an object can be declared *synchronized*, which implies that the method is executed in a monitor whose lock is the one associated with that object. Instance methods are associated with the lock of the object instance, whereas static methods are associated with the lock of the object instance that represents the object class (and which is of type *java.lang.Class*).

2.2.1. Inconsistency due to Asynchronous Termination. One important function of a mobile agent platform is the termination of agents. When an agent migrates or terminates, all of its allocated resources should be reclaimed as soon as possible. That is, all threads of the agent shall be stopped and memory allocated by the agent shall become eligible for garbage collection. Especially when a misbehaving agent is detected it has to be removed from the system with immediate effect.

Java allows to asynchronously terminate a running thread by means of the *stop* method of class *java.lang.Thread*. This method causes a *ThreadDeath* exception to be thrown asynchronously in the thread to be stopped. Unfortunately, thread termination in Java is an inherently unsafe operation, because the terminated thread immediately releases all monitors. Consequently, objects may be left in an inconsistent state. As long as these objects are exclusively managed by the agent to be removed from the system, a resulting inconsistency may be acceptable.[2]

However, if a thread is allowed to cross agent boundaries for communication purpose (e.g., inter-agent method invocation), the termination of a thread has to be deferred until it has completed executing in the context of other agents. Otherwise, the termination of one agent may damage a different agent that is still running in the system. Unfortunately, delayed thread termination prevents immediate memory reclamation, because references to objects of the terminated agent may be kept alive on the execution stack of the thread. Even worse, if shared objects, such as certain internals of the JVM, are left inconsistent, asynchronous thread termination may result in a crash of the JVM. For this reason, the *stop* operation has been deprecated in the Java 2 platform.

To solve these problems, the mobile agent platform has to enforce a thread model where each thread is bound to a single agent. Threads must not be allowed to cross agent boundaries arbitrarily. Upon the invocation of a method in a different agent, a thread switch is necessary. The called agent has to maintain worker threads to accept external method calls. However, this approach negatively affects performance, because thread switches are rather expensive operations.

To ensure the integrity of shared data structures and of JVM internals, the mobile agent system has to enforce a user/kernel boundary, where shared structures are manipulated only within the kernel. With the aid of a locking mechanism, it is possible to ensure atomic kernel operations. That is, requests for asynchronous termination are deferred until the thread to be stopped has left the kernel. Because kernel operations can be implemented with a short and bounded execution time, domain termination cannot be delayed arbitrarily. All critical JVM operations have to be guarded by a kernel entry. Again, this solution causes some overhead.

[2]If the agent state is captured (e.g., serialized) after termination, inconsistencies may corrupt the further execution of the agent on other platforms. The agent is responsible to freeze its non-transient state before requesting migration.

LISTING 4
A method to prevent thread termination.

```
while (true) {
  try {
    while (true);
  }
  catch (Throwable t) {}
}
```

LISTING 5
Catching ThreadDeath can be prevented by rewriting the bytecode in listing 4 in a way that is functionally equivalent to the given Java code transformation.

```
while (true) {
    try {
        while (true);
    }
    catch (Throwable t) {
        if (t instanceof ThreadDeath) {
          throw t;
        }
    }
}
```

2.2.2. Interception of Asynchronous Termination. There are further problems with asynchronous thread termination: The *stop* method does not guarantee that the thread to be killed really terminates, because the thread may intercept the *ThreadDeath* exception. For instance, consider the code fragment in listing 4, which cannot be terminated easily.

Note that not only exception handlers may intercept *ThreadDeath* exceptions, but *finally*{} clauses may prevent termination as well. However, the Java compiler maps *finally*{} statements to special exception handlers. Thus, it is sufficient to solve the problem with exception handlers that catch *ThreadDeath* or a superclass thereof.

The JavaSeal mobile agent kernel [10] enforces a set of restrictions on exception handlers that may catch *ThreadDeath*, in order to ensure the termination of such handlers. However, this approach imposes severe restrictions on the programming model. For instance, untrusted agents may not use *finally*{} clauses. Furthermore, the JavaSeal implementation is incomplete, as a *monitorexit* instruction[3] in an exception handler may throw *NullPointerException* or *IllegalMonitorStateException*, which can be caught by user code.

Another solution to this problem involves rewriting of agent bytecode so that *ThreadDeath* exceptions are immediately thrown again by all exception handlers. This approach is used in the J-SEAL2 mobile agent kernel [3]. Listing 4 shows a portion of Java code and listing 5 its rewritten counterpart. For the ease of reading, we give the transformation at the Java level, whereas rewriting would be done actually at the JVM bytecode level.

2.2.3. Undefined Thread Scheduling. Neither the Java language [18] nor the JVM specification [23] define the scheduling of Java threads. Therefore, it is not guaranteed that, on every Java platform, high-priority surveillance threads preempt other threads. A related problem is priority inversion. This means that a high-priority thread may have to wait until a low-priority thread releases a monitor. However, the low-priority thread may not be scheduled if there are other threads ready to run that have a higher priority.

Most standard Java runtime systems offer native threads that are scheduled by the operating system i. e., the scheduling is platform-dependent. This means that a surveillance task using a high-priority thread that has been tested in one environment may not work well in another one, contradicting the Java motto “write once, run anywhere.” Moreover, an increasing system load (an increasing number of threads) often affects the scheduling and may prevent high-priority threads from executing regularly.

Priority inversion may be addressed by temporarily raising the priority of a thread executing a critical section. However, in many JVMs adapting thread priorities is an expensive operation, since it triggers the scheduler.

The realtime specification for Java [9] specifies priority-based preemptive scheduling with at least 28 different levels of priority for all compliant implementations. The realtime specification covers many other topics important for realtime

[3] The compilation of a *synchronized*{} statement creates an exception handler whose task is to release the monitor in case of any exception. Because synchronization is an important concept of the Java language, JavaSeal allows agents to use the *monitorexit* instruction within exception handlers.

LISTING 6
Example bytecode that acquires a lock that is not released after completion.

```
static void captureMonitor(java.lang.Class)
   0 aload_0
   1 monitorenter
   2 return
```

systems. Therefore, standard Java runtime systems will not likely conform to the realtime specification, because they target environments without realtime requirements and the underlying operating system does not necessarily offer realtime guarantees either. Consequently, we think that a subset of the realtime specification should be integrated into a new version of the standard Java specification, so that applications that depend on scheduling mechanisms – such as mobile agent systems – may run consistently across different JVM implementations.

2.3. Bytecode Verification. Java relies on static and dynamic checks to ensure that the execution of programs proceeds according to the language semantics. Before a program is linked into the JVM, the Java bytecode verifier performs static analysis of the program to make sure that the bytecode actually represents a valid Java program. Dynamic checks (e.g., array bounds checks) are incorporated in many JVM instructions.

Unfortunately, bytecode verifiers of several current standard Java implementations also accept bytecode that does not represent a valid Java program. The result of the execution of such bytecode is undefined, and it may even compromise the integrity of the Java runtime system.

At the Java bytecode level, the allocation of an object is separated from its initialization. One important task of the Java verifier is to prevent uninitialized objects from being used (e.g., the fields of uninitialized objects must not be accessed and only a constructor can be invoked on an uninitialized object). In [14] the authors take an in-depth look at object initialization in Java and define rules to be enforced by the Java verifier. However, one particular issue is not addressed: finalizers will be invoked on uninitialized objects, which undermines the properties of object initialization that are meant to be enforced by the Java verifier.

Another source of problems are the synchronization primitives of the Java bytecode. The example given in listing 6 depicts the bytecode of a method that acquires a class lock without releasing the lock after completion:

This code sequence does not constitute a valid Java program because the *monitorenter* instruction (which acquires a lock) is not paired with a matching *monitorexit* (which releases the lock). Neither is an exception handler present that releases the lock in the case of an exception. Nonetheless, several Java verifier implementations do not reject this code. The effects of executing this code are undefined and depend on the particular JVM implementation. We tested the method invocation *captureMonitor(Thread.class)* with 3 different JVMs on a Windows platform and observed varying outcomes with each of them:

Hotspot Server VM 2.0: An *IllegalMonitorStateException* is thrown at the end of the method and the monitor is released.

JDK 1.4/1.5 (Hotspot server and client): An *IllegalMonitorStateException* is thrown at the end of the method.

JDK 1.2.2 Classic VM: No exception is thrown and the monitor remains locked until the thread that has acquired the monitor terminates.

IBM JDK 1.3.0 Classic VM: The monitor is not released even after the locking thread has terminated. Subsequent attempts by other threads to create new threads are blocked, because thread creation involves a static synchronized method, which waits for the release of the class lock. In fact, this kind of attack is similar to the DoS attack shown in Section 2.1. However, this attack is even worse, because the lock is not released after all attacker threads have terminated, whereas the attack in Section 2.1 can be resolved by stopping the attacking thread.

Listing 7 depicts another example of disarranged bytecode that is not rejected by several standard Java verifiers. In this bytecode sample, the target of the exception handler is the first instruction protected by the same handler. Such a construction is not possible at the Java language level, because a *try*{} block cannot serve as its own *catch*{} clause.

At bytecode position 1 there is an infinite loop (*goto 1*), which is protected by the exception handler.[4] In case of an exception, the handler continues the same loop. Therefore, it is not possible to stop a thread executing such code. Even the transformation shown in listing 5 does not help, since its application would cause an infinite loop of catching and re-throwing the same *ThreadDeath* exception.

[4]The *aconst_null* instruction at position 0 ensures that there is always a single reference on the stack at code position 1 (this constraint is enforced by the Java verifier). When an exception is caught, the stack is cleared and a reference to the exception object is pushed onto the stack. The *return* instruction is never reached.

LISTING 7
Example bytecode whith a disarranged exception handler.

```
static void preventTermination()
   0 aconst_null
   1 goto 1
   4 return

Exception table:
   from   to  target type
     1     4     1   <Class java.lang.Throwable>
```

In order to prevent such attacks, improved bytecode verification is necessary. A better solution would be the definition of an alternative Java class format, which enables simpler verification. For instance, *Slim Binaries* [17] encode the abstract syntax tree of a program and can be verified easily, because the code can be restricted to valid syntax trees of the programming language. Thus, expensive bytecode verification can be avoided. This is particularly beneficial for mobile agents, whose startup overhead frequently exceeds execution time before migration.

2.4. Security Model. One of the most prominent security features in the Java security model is the fact that permissions of a thread are limited to the permissions granted to the least privileged class on its execution stack. Permissions are assigned by the *class loader* when the class is defined. Any class can check whether the current thread has a particular permission, by invoking the *access controller* with a template of the permission that shall be checked. The access controller responds by throwing an exeception if the permission is not granted, and silently returns otherwise.

Classes can execute *privileged actions*, which means that the class assumes responsibility for subsequent actions, and the permissions granted subsequently to the executing thread shall be the ones granted to the class that invoked the privileged action (in that case, stack introspection stops at the privileged context). New permission types can be introduced by means of a permission class that represents the type. While this gives great flexibility in terms of implementing security checks it also lacks central control.

Security checks as well as privileged actions may be scattered throughout the class packages, and it is next to impossible to determine with certainty whether a given application actually enforces a particular security policy. Even a small error can have disastrous effects on the system security as a whole; in particular, multiple small errors culminate into bigger ones. For instance, write access to the VM binary or permission to execute native code is virtually equivalent with granting the *all permission*.

Recall that local classes are visible globally. For instance, assume that a *logger* class writes log entries to a file. We assume that the class is initialized with a file name, and that it is allowed to write arbitrary files. Log events are potentially caused by threads with a trust level that is lower than the trust level of the logger, therefore the logger uses a privileged action to write to the log file. Since the logger class is globally visible and is initialized with a file name, any code (including code without file access permissions) may instantiate the class with an arbitrary file name and use it to write log entries to it.

Rather than binding permissions to a class, permissions need to be bound to a particular *instance*. This can be achieved as follows: in its constructors, the trusted class stores a snapshot of the current access control context (ACC) in a private variable. Whenever the privileged action is executed, the stored ACC is set. The effective set of permissions granted to the thread is therefore the intersection of the privileges granted to the trusted class and the permissions current at the time when the class was created. Consequently, less privileged code will gain no additional permissions by instantiating the trusted class, whereas the trusted *instance* can be used without restrictions. However, this is a strict design requirement and must be enforced consistently throughout the design and implementation phase.

A feature that is desirable for any mobile agent system is instant revocation of permissions. In other words, permissions granted to an agent can be expanded or withdrawn dynamically (e.g., when the agent misbehaves). One way to achieve this is to implement a custom *ProtectionDomain* that supports that feature, and which is assigned to all classes of the agent by means of a custom class loader. However, this approach is not guaranteed to work because protection domains may be compressed as a consequence of optimizations (although it does work as intended in the reference implementation of the JDK as of version 1.3).

Starting with version 1.3, the JDK provides the *DomainCombiner* mechanism that could be applied to permission revocation. However, using this mechanism is tricky. Once a permission is assigned statically to a malicious class by means of a protection domain, this permission cannot be revoked even with a *DomainCombiner*. The malicious class can always issue a privileged action with its given permissions which blocks invocation of the *DomainCombiner* further

down the stack. The solution is to grant no permissions a priori, but only dynamically by means of a *DomainCombiner*.

On the other hand, if code shall have access to reserved resources when being invoked on behalf of theads of other logical entities, then that code needs to save an ACC in its original thread, and use it in a privileged action. Otherwise, the privileged action relinquishes all permissions (because none were granted statically) and its *DomainCombiner* will not be invoked on permission checks.

In summary, the security model of the Java 2 platform is very flexible and powerful on the one hand, but on the other hand it is also very complicated and depends on the perfect orchestration of all components of the application and the mobile agent middleware. This constitutes considerable risk, because breach of security or integrity of the VM may expose the account under whose authority the VM is executed.

2.5. Lack of Resource Control. Current standard JVMs do not support resource control e.g., mobile agents may spawn an arbitrary number of threads and each thread may consume an arbitrary number of CPU cycles and allocate memory until an `OutOfMemoryError` is thrown.

The lack of resource management features makes it easy to launch DoS attacks. Moreover, even in the absence of an attack, lack of awareness of the resource consumption of executing mobile agents may lead to an overload of the system and negatively impacts the system's stability.

3. Mobile Agent Isolation. In this section we argue for a strong isolation of mobile agents that execute within the same JVM, and we discuss several approaches to accomplish this goal.

3.1. Requirements. If multiple mobile agents execute within the same JVM process, faults within one agent shall not affect the stability of the JVM and of other agents i. e., strong isolation of agents is needed [4].

Language safety in Java (a combination of strong typing, memory protection, automatic memory management, and bytecode verification) already guarantees some basic protection, as it is not possible to forge object references [31]. However, language safety itself does not guarantee isolation of mobile agents. Pervasive aliasing in object-oriented languages leads to a situation where it is impossible to determine which objects belong to a certain agent and therefore to check whether an access to a particular object is permitted or not. Thus, it is crucial to introduce the concept of isolation in the JVM, similar to the process abstraction in operating systems.

What is needed is an isolation boundary around each mobile agent, which encapsulates the set of classes required by the agent, its threads, as well as all objects allocated by these threads. Different mobile agents must not share any structures that could cause unwanted side effects, such as class locks (see Section 2.1).

As we have seen in Section 2.2.1, threads crossing mobile agent boundaries hamper the safe termination of agents. To solve this problem, a thread model is needed where each thread is bound to a single agent. Threads must not be allowed to cross agent boundaries arbitrarily. Upon the invocation of a method in a different agent, a thread switch is necessary. The called agent has to maintain worker threads to accept requests by other agents. Messages must not be passed by reference between agents, since this would create inter-agent aliasing. However, this approach may negatively affect performance, because of the extra overhead due to thread switches.

Closely related to mobile agent isolation is the safe termination of agents. If there is no sharing between agents and threads cannot cross agent boundaries, the removal of an agent will not leave other agents in an inconsistent state. For instance the execution platform does not need to care about potential inconsistencies and instead may simply terminate all threads running in an agent in a brute force fashion. Still, as shown in Section 2.2.2, the `stop()` method of `java.lang.Thread` is not suited for this purpose. Hence, a dedicated primitive for agent termination is necessary. Mobile agent isolation itself does not solve the problems of bytecode verification discussed in Section 2.3, but it confines them to the faulty agents, which can be safely terminated.

3.2. Solutions. At a high level, we distinguish between solutions that aim at providing isolation within standard Java runtime systems and approaches that rely on a modified JVM.

Among the first systems that added some sort of process model to standard JVMs were J-Kernel [29] and JavaSeal [27]. Both of these systems were implemented in pure Java. Hence they are available on any standard JVM.

3.2.1. J-Kernel. J-Kernel [29] is a Java micro-kernel supporting isolated components. In J-Kernel communication is based on capabilities. Java objects can be shared indirectly between components by passing references to capability objects. However, in J-Kernel inter-component calls may block infinitely and may delay component termination. Moreover, the classes of the JDK are used by all components, which may introduce some unwanted side-effects between components. Consequently, J-Kernel offers only incomplete isolation.

3.2.2. JavaSeal and J-SEAL2. JavaSeal [27] was designed as a secure system to execute untrusted, mobile code. It supports the hierarchical process model of the Seal calculus [28], where isolated components are organized in a tree. Each component has its separate set of threads, which are not allowed to cross component boundaries. In contrast to J-Kernel, component termination in JavaSeal cannot be delayed by blocked threads.

JavaSeal allows only for very restrictive communication between components that are in a direct parent-child relationship. This allows policy components to be inserted in the hierarchy in order to control all communication of a child component. If a message has to be transmitted between components that are not direct neighbours in the hierarchy, it must be routed along the edges of the component tree, where all involved components have to actively forward the message. Communication requires deep copies of the objects to be passed between components. As the generation of deep copies is based on the serialization and de-serialization of object graphs, the communication overhead can be excessive when compared to direct method invocation. This problem is even more severe if a message is routed through multiple components.

J-SEAL2 [3] builds on JavaSeal, but offers several improvements. The communication model supports so-called "external references," which allow indirect sharing between components. "External references" allow to shortcut communication paths in the hierarchy. They are similar to capabilities in J-Kernel, but are implemented in such a way that component termination cannot be delayed. Other improvements of J-SEAL2 include extended bytecode verification, which prevents problems as outlined in Section 2.3. Moreover, J-SEAL2 supports resource management, which will be discussed in Section 4.2.2.

In contrast to J-Kernel, JavaSeal and J-SEAL2 do not allow untrusted mobile agents to access arbitrary JDK classes, in order to prevent side-effects between components. Only a few methods of some core classes (such as `java.lang.Object`, `java.lang.String`, etc.) may be called by untrusted components. For other features (e.g., network access), dedicated, trusted service components have to be installed to mediate access to these features.

As a result, JavaSeal and J-SEAL2 offer the best possible isolation achievable on standard Java runtime systems. However, this comes at a very high price: As most of the JDK functionality cannot be directly accessed, components must be manually rewritten to use service components instead. Hence, this approach does not allow to run untrusted legacy code and it severely changes the programming model Java developers are familiar with. For this reason, such an approach to enforce strict isolation on standard JVMs does not appeal to most developers.

3.2.3. KaffeOS. The other approach is to modify a given JVM in order to support component isolation. For instance, the Utah Flux Research Group has worked on the development of specialized Java runtime systems supporting component isolation [2, 1].

Their most prominent system, KaffeOS [1], is a modified version of the freely available Kaffe virtual machine [30]. KaffeOS supports the operating system abstraction of processes to isolate components from each other, as if they were run on their own JVM. The advantage of this approach is that full component isolation is achieved without compromising the Java programming model. However, there are also severe drawbacks to this solution: The modified JVM is available only on a limited number of platforms and the costs for porting it to other environments are high. Moreover, optimizations found in standard JVMs are not available for the Kaffe virtual machine, resulting in inferior performance. In [1] the authors reported that IBM's JVM [24] was 2–5 times faster than KaffeOS. Since then, the performance of standard Java runtime systems has significantly improved; hence, an even bigger performance difference could be expected if such a comparison was made at the time of writing.

3.2.4. Java Isolation API and MVM. The urgent need for component isolation within standard JVMs has been realized also by the industry. In the context of JSR-121 [20], a Java Isolation API has been defined. The core of this API is an abstraction called *Isolate*, which allows to strongly protect Java components from each other. The Isolation API ensures that there is no sharing between different Isolates. Even static variables and class locks of system classes are not shared between Isolates in order to prevent unwanted side effects. Isolates cannot directly communicate object references by calling methods in each other, but have to resort to special communication links which allow to pass objects by deep copy. An Isolate can be terminated in a safe way, releasing all its resources without affecting any other Isolate in the system.

The Isolation API follows a rather minimalistic approach in order to enable implementation across a wide variety of systems, including the Java 2 Micro Edition (for embedded devices). In particular, the communication model is very simple and may cause high overhead if large messages are transmitted frequently between different Isolates.

The Java Isolation API is supposed to be supported by future versions of the JDK. For the moment, it is necessary to resort to research JVMs that already provide the Isolation API, such as the MVM [11]. The MVM is a modified version of the Sun JDK 1.5.0 HotSpot Client VM supporting multiple Isolates within a single JVM process. Unfortunately, the MVM is currently only available on SPARC systems running the Solaris operating system. Moreover, as the more efficient HotSpot Server VM is not yet supported, there may be performance problems for complex applications.

4. Resource Management. In this section we explain why resource management, a missing feature in current standard JVMs, is needed in future Java runtime systems and we discuss several approaches to add resource awareness to the JVM.

4.1. Requirements. Using current standard Java runtime systems, it is not possible to monitor or limit the resource consumption of mobile agents. In order to ensure a secure, reliable, and scalable system that optimally exploits available resources, such as CPU and memory, support for resource management is a prerequiste. Resource management comprises resource accounting and resource control. Resource accounting is the non-intrusive monitoring of resource consumption, whereas resource control implies the enforcement of resource limits. Resource accounting helps to implement resource-aware agents that adapt their execution according to resource availability (self-tuning). Resource control is essential to guarantee security and reliability, as it allows to prevent malicious or erroneous agents from overusing resources. E.g., resource control is crucial to prevent denial-of-service attacks.

Below we summarize our requirements for the provision of resource management features in the JVM:

- Support for resource accounting as well as resource control.
- Support for physical resources (e.g., CPU, memory) as well as for 'logical' resources (e.g., number of threads).
- Extensibility—Support for user-defined, application-specific resources (e.g., database connections in an application server).
- Flexibility—Support for user-defined resource consumption policies.
- Low overhead.
- Compatibility—Existing agent code shall be executable without changes.
- Portability—Implementations shall be available on a large number of different systems.

4.2. Solutions. As before in Section 3, we distinguish between resource management libraries that are compatible with standard JVMs and specialized JVM implementations.

4.2.1. JRes. JRes [13] was the first resource management system for standard JVMs. It takes CPU, memory, and network resource consumption into account. The resource management model of JRes works at the level of individual Java threads. In other words, there is no notion of isolated component, and the implementation of resource control policies is therefore cumbersome. JRes is a pure resource management system and does not enforce any protection of components. However, JRes was integrated into the J-Kernel [29] (see Section 3.2.1) to support isolation and resource management within the same system.

JRes relies on a combination of bytecode instrumentation and native code libraries. To perform CPU accounting, the approach of JRes is to make calls to the underlying operating system, which requires native code to be accessed. A polling thread regularly obtains information concerning CPU consumption from the operating system. For memory accounting, JRes uses bytecode rewriting, but still needs the support of a native method to account for memory occupied by array objects. Finally, 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.

The drawbacks of JRes are the limited number of supported resources and the lack of extensibility. Moreover, as JRes depends on native code and on specific operating system features, it is not easily portable. Another problem is the use of a polling thread to obtain CPU consumption information, as the regular scheduling of this thread is not guaranteed (see Section 2.2.3). Hence, the activation of the polling thread is platform-dependent and an eventual CPU overuse may be detected too late.

4.2.2. J-SEAL2. The J-SEAL2 system [3] also supports resource management [7]. In contrast to JRes, J-SEAL2 is implemented in pure Java, i. e., it is fully portable and compatible with any standard JVM. J-SEAL2 makes heavy use of bytecode instrumentation for CPU and memory management. Like in the case of JRes, the set of supported resources is rather restricted: In the default configuration, only CPU, memory, the number of threads, and the number of isolated components can be managed.

J-SEAL2 focuses on CPU management [7]. In order to achieve full portability, J-SEAL2 does not rely on the CPU time as metric, but it exploits the number of executed JVM bytecode instructions as platform-independent, dynamic metric [15]. In the J-SEAL2 system, each thread keeps track of the number of bytecode instructions it has executed within a thread-local counter. Periodically, a high-priority supervisor thread executes and aggregates the CPU consumption (i. e., the number of executed bytecodes) for all threads within each isolated component. This supervisor thread may also take actions against an overusing component: It may terminate the component or lower the priority of the component's threads. As in the case of JRes, the use of a high-priority thread to check CPU consumption is flawed by the underspecified scheduling semantics of Java threads.

While the use of the bytecode metric for measuring CPU consumption has many advantages [7], the J-SEAL2 approach also suffers from several drawbacks: The execution of native code cannot be accounted for (including also garbage collection and dynamic compilation), and the relationship between CPU time and the number of executed bytecode instructions remains unclear. The complexity of various bytecode instructions is very different. Furthermore, in the presence of just-in-time compilation, the execution time for the same bytecode instruction may depend very much on the context where the instruction occurs. Nonetheless, at least for simple JVM implementations, it may be feasible to assign weights to (sequences of) bytecodes in order to estimate CPU time on a particular system [26].

4.2.3. J-RAF2. J-RAF2[5], the Java Resource Accounting Framework Second Edition [5, 19], builds on the resource management ideas that were first integrated into J-SEAL2 [7]. In contrast to J-SEAL2, J-RAF2 does not assume a particular component model, it is a general-purpose resource management library, similar to JRes. However, in contrast to JRes, J-RAF2 is implemented in pure Java to guarantee portability.

The most important improvement of J-RAF2 over J-SEAL2 concerns CPU management. J-RAF2 does not rely on a dedicated supervisor thread to enforce CPU consumption policies. J-RAF2 uses an approach called 'self-management', where each thread in the system periodically invokes a so-called CPU manager that enforces a user-defined CPU consumption policy on the calling thread. The regular invocation of the CPU manager is achieved by bytecode instrumentation, where polling code is inserted in such a way that the number of executed bytecode instructions between consecutive polling sites is limited. I.e., polling code is injected before and after method invocations (call/return polling [16]) and in loops. This approach has the big advantage of not depending on the scheduling of the JVM. Thanks to several optimizations, the average overhead of CPU management could be kept reasonable, about 15–30% depending on the JVM [6].

4.2.4. NOMADS and the Aroma VM. NOMADS [25] 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 Java compatible VM, the Aroma VM, a copy of which is instantiated for each agent. Resources are managed manually, on a per-agent basis or using a non-hierarchical notion of group. The major limitation of the Aroma VM is the lack of a just-in-time compiler, resulting in low performance.

4.2.5. Resource Management API. The work by Czajkowski et al. [12] is the first attempt to define a general-purpose, flexible, and extensible resource management API for Java, which has been validated in the MVM prototype [11].

The API supports user-defined resources, which are specified through a set of attributes that define the resource semantics. The API allows for resource reservations and notifications that are issued upon user-defined resource consumption situations. Such notifications ease the implementation of resource-aware program behaviour. Moreover, notifications can be synchronous and prevent actions that would exceed a given resource limit from succeeding, which is essential for resource control.

One limitation of the resource management API is that it requires a JVM with support for Isolates [20], because the Isolate is the smallest execution unit to which resource management policies can be applied. On the one hand, this approach simplifies the management of resources that can be handed over from one thread to another one, such as allocated heap memory (objects). As threads are confined to a single Isolate throughout their lifespan and object references cannot be shared across Isolate boundaries, handover of allocated heap memory across Isolate boundaries is not possible. Hence, managing such resources at the level of Isolates avoids complications due to the change of resource ownership. On the other hand, there are resources for which management at the level of threads can be very useful, such as CPU time. For instance, assume a service component that manages a pool of threads to handle incoming requests. A CPU management policy may want to limit the CPU time spent on processing a particular request, which would require binding individual threads to the CPU management policy. Unfortunately, the resource management API presented in reference [12] is not well adapted for such a scenario.

The need for a standardized resource management API has been realized also by the industry. In the context of JSR-284 [21], an improved resource management API is now under development.

5. Conclusion. The widespread distribution of Java as well as the mass of code and support that is available to Java developers makes it a sine qua non for mobile agent systems. This said, Java is probably the best and the worst that happened to mobile agents. The best because developing and deploying mobile agent systems became easy and highly portable; the worst because it is next to impossible to preserve Java's usefulness and to build a sufficiently secure system at the same time. In order to deploy industrial strength mobile agent systems that are robust against various forms of DoS as well as breaches of confidentiality, Java has to evolve from an application-level runtime system into a true operating

[5] `http://www.jraf2.org/`

system with proper accounting and application separation capabilities. In the Java Community Process, this path has already been taken.

REFERENCES

[1] G. BACK, W. HSIEH, AND J. LEPREAU, *Processes in KaffeOS: Isolation, resource management, and sharing in Java,* In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000), San Diego, CA, USA, October 2000.

[2] G. BACK, P. TULLMANN, L. STOLLER, W. HSIEH, AND J. LEPREAU, *Techniques for the design of Java operating systems,* In: Proceedings of the 2000 USENIX Annual Technical Conference, pages 197–210, San Diego, CA, June 2000.

[3] WALTER BINDER, *Design and implementation of the J-SEAL2 mobile agent kernel,* In: The 2001 Symposium on Applications and the Internet (SAINT-2001), pages 35–42, San Diego, CA, USA, January 2001.

[4] WALTER BINDER, *Secure and reliable Java-based middleware—Challenges and solutions,* In: First International Conference on Availability, Reliability and Security (ARES-2006), pages 662–669, Vienna, Austria, April 2006. IEEE Computer Society.

[5] WALTER BINDER AND JARLE HULAAS, *A portable CPU-management framework for Java,* IEEE Internet Computing, 8(5):74–83, Sep./Oct. 2004.

[6] WALTER BINDER AND JARLE HULAAS, *Java bytecode transformations for efficient, portable CPU accounting,* In: First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005), volume 141 of ENTCS (Electronic Notes in Theoretical Computer Science), pages 53–73, Edinburgh, Scotland, April 2005.

[7] WALTER BINDER, JARLE G. HULAAS, AND ALEX VILLAZÓN, *Portable resource control in Java,* ACM SIGPLAN Notices, 36(11):139–155, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[8] WALTER BINDER AND VOLKER ROTH, *Secure mobile agent systems using Java: Where are we heading?* In: Seventeenth ACM Symposium on Applied Computing (SAC-2002), Madrid, Spain, March 2002.

[9] G. BOLLELLA, B. BROSGOL, P. DIBBLE, S. FURR, J. GOSLING, D. HARDIN, AND M. TURNBULL, *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, USA, 2000.

[10] CIARÁN BRYCE AND JAN VITEK, *The JavaSeal mobile agent kernel,* In: First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99), Palm Springs, CA, USA, October 1999.

[11] GRZEGORZ CZAJKOWSKI AND LAURENT DAYNÈS, *Multitasking without compromise: A virtual machine evolution,* In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), pages 125–138, Tampa Bay, Florida, October 2001.

[12] GRZEGORZ CZAJKOWSKI, STEPHEN HAHN, GLENN SKINNER, PETE SOPER, AND CIARAN BRYCE, *A resource management interface for the Java platform,* Software Practice and Experience, 35(2):123–157, November 2004.

[13] GRZEGORZ CZAJKOWSKI AND THORSTEN VON EICKEN, *JRes: A resource accounting interface for Java,* In: Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98), volume 33, 10 of *ACM SIGPLAN Notices*, New York, USA, October 1998.

[14] S. DOYON AND M. DEBBABI, *On object initialization in the Java bytecode,* Computer Communications, 23(17):1594–1605, November 2000.

[15] BRUNO DUFOUR, KAREL DRIESEN, LAURIE HENDREN, AND CLARK VERBRUGGE, *Dynamic metrics for Java,* ACM SIGPLAN Notices, 38(11):149–168, November 2003.

[16] MARC FEELEY, *Polling efficiently on stock hardware,* In: The 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark, pages 179–187, June 1993.

[17] MICHAEL FRANZ AND THOMAS KISTLER, *Slim binaries,* Communications of the ACM, 40(12):87–94, December 1997.

[18] JAMES GOSLING, BILL JOY, AND GUY L. STEELE, *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1st edition, 1996.

[19] JARLE HULAAS AND WALTER BINDER, *Program transformations for portable CPU accounting and control in Java,* Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation), pages 169–177, Verona, Italy, August 24–25 2004.

[20] Java Community Process. *JSR 121 – Application Isolation API Specification,* `http://jcp.org/jsr/detail/121.jsp`

[21] Java Community Process. *JSR 284 – Resource Consumption Management API,* `http://jcp.org/jsr/detail/284.jsp`

[22] B. W. LAMPSON, *A note on the confinement problem,* Communications of the ACM, 10:613–615, October 1973.

[23] TIM LINDHOLM AND FRANK YELLIN, *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[24] T. SUGANUMA, T. OGASAWARA, M. TAKEUCHI, T. YASUE, M. KAWAHITO, K. ISHIZAKI, H. KOMATSU, AND T. NAKATANI, *Overview of the IBM Java Just-in-Time compiler, IBM Systems Journal*, 39(1), 2000.

[25] NIRANJAN SURI, JEFFREY M. BRADSHAW, MAGGIE R. BREEDY, PAUL T. GROTH, GREGORY A. HILL, RENIA JEFFERS, TIMOTHY S. MITROVICH, BRIAN R. POULIOT, AND DAVID S. SMITH, *NOMADS: toward a strong and safe mobile agent system,* In: Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00), NY, June 2000.

[26] JAMES DAVID TURNER, *A Dynamic Prediction and Monitoring Framework for Distributed Applications*. Phd thesis, Department of Computer Science, University of Warwick, UK, May 2003.

[27] JAN VITEK, CIARÁN BRYCE, AND WALTER BINDER, *Designing JavaSeal or how to make Java safe for agents,* Technical report, University of Geneva, July 1998. `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/98/javaSeal.pdf`

[28] JAN VITEK AND GIUSEPPE CASTAGNA, *Seal: A framework for secure mobile computations,* In: Internet Programming Languages, 1999.

[29] T. VON EICKEN, C.-C. CHANG, G. CZAJKOWSKI, AND C. HAWBLITZEL, *J-Kernel: A capability-based operating system for Java, Lecture Notes in Computer Science*, 1603:369–394, 1999.

[30] T. WILKINSON, *Kaffe—a Java virtual machine,* Web pages at `http://www.kaffe.org/`.

[31] F. YELLIN, *Low level security in Java,* In: Fourth International Conference on the World-Wide Web, MIT, Boston, USA, December 1995.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 10, 2006

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 13–35. http://www.scpe.org
ISSN 1895-1767

IMPLEMENTING MOBILE AND DISTRIBUTED APPLICATIONS IN X-KLAIM*

LORENZO BETTINI, ROCCO DE NICOLA AND MICHELE LORETI

Abstract. In this paper we present X-KLAIM, an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple distributed tuple spaces and mobile code. The language consists of a set of coordination primitives inspired by Linda, a set of operators for building processes borrowed from process algebras and a few classical constructs for sequential programming. We present some programming examples in X-KLAIM, dealing with mobile code programming paradigms, such as client-server, code mobility and mobile agents.

1. Introduction. Technological advances of both computers and telecommunication networks, and development of more efficient communication protocols are leading to an ever increasing integration of computing systems and to diffusion of so called Global Computers [19]. These are massive networked and dynamically reconfigurable infrastructure interconnecting heterogeneous, typically autonomous and mobile components, that can operate on the basis of incomplete information. Designing and implementing applications over a global network is inherently different from designing and implementing stand–alone ones. Network programming has to deal with the following additional issues [20]:

- the physical distribution of hosts and data can be essential, and local and remote behaviors can be significantly different;
- systems are asynchronous and less predictable: a temporary disconnection of a remote host cannot be distinguished from a system fault;
- different forms of program termination have to be considered; applications can terminate due to missing permissions or to the low level of quality of services.

Global Computers are thus fostering a new style of distributed programming that has to take into account variable guarantees for communication, cooperation and mobility, resource usage, security policies and mechanisms for dealing with failures. This has stimulated the proposal of new theories, computational paradigms, linguistic mechanisms and implementation techniques. We have thus witnessed the birth of many calculi and kernel languages intended to support programming according to the new style and to provide tools for formal reasoning over the modeled systems.

In this paper we present X-KLAIM, an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code (possibly object-oriented). X-KLAIM is based on KLAIM the *Kernel Language for Agents Interaction and Mobility* [23, 8]. The distinguishing features of the approach is the explicit use of localities for accessing data or computational resources. KLAIM can be seen as an asynchronous higher–order process calculus whose basic actions are the original Linda [28] primitives enriched with explicit information about the location of the nodes where processes and tuples are allocated.

The *blackboard* approach, of which tuple space based models are variants, is one of the most appreciated model for dealing with mobile agents (see, e.g., [26], that examines several messaging models for mobile agents) also because of its flexibility. The Linda *asynchronous* communication model permits

- *time uncoupling*: tuples' life time is independent of the producer process' life time,
- *destination uncoupling*: the creator of a tuple is not required to know the future use or the destination of that tuple,
- *space uncoupling*: communicating objects need to know a single interface, i. e., the operations over the tuple space. This approach is also called *flow-of-objects* [4] as opposed to *method invocation*, which requires many interfaces for the operations supplied by remote objects.

When moving to open distributed systems and large-scale, multi-users applications, the Linda coordination model suffers from the lack of *modularity* and *scalability*: identification tags of tuples, which are conceptually part of different contexts, may collide. In other words, processes of different computations could interfere and a mechanism to structure communication and hide information, e.g., to create areas restricted to a subset of the processes, is needed. Explicit localities enable the programmer to distribute and retrieve data and processes to and from the sites of a net and to structure the tuple space as multiple, located spaces. Moreover, localities, considered as first–order data, can be dynamically created and communicated over the network. The overall outcome is a powerful programming formalism that, for example, can easily be used to model encapsulation. In fact, an encapsulated module can be implemented as a tuple space at a private locality, and this ensures controlled accesses to data.

*The work presented in this paper has been partially supported by EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract IST-3-016004-IP-09).

In the rest of the paper we present the main features of X-KLAIM, and some programming examples dealing with code and agent mobility (e.g., a load balancing system, Section 3.3, and a mobile agent based information retrieval, Section 3.2) and with distributed applications in general (a chat system, Section 5). The last two examples show two more involved systems: a mobile agent based system for distributed document updates (Section 6; this is a modified version of the system presented in [12]) and an distributed implementation of the strategy game Cluedo (Section 7).

For a more complete description of the programming language X-KLAIM we refer the interested reader to the tutorial that can be found in [10] (from where we borrow modified versions of some examples shown in this paper).

2. An overview of X-KLAIM**.** X-KLAIM (*eXtended* KLAIM) [11, 10] is an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code. It is based on the kernel language KLAIM (*Kernel Language for Agent Interaction and Mobility*) [8] and is inspired by the coordination language Linda [28], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are sequences of information items (called *fields*). There are two kinds of fields: *actual fields* (i. e., expressions, processes, localities, constants, identifiers) and *formal fields* (i. e., variables). Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier. Tuples are anonymous and content-addressable; *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields have matching values or formals;
- formal fields match any value of the same type, but two formals never match, and two actual fields match only if they are identical.

For instance, tuple (`"foo"`, `"bar"`, 100 + 200) matches with (`"foo"`, `"bar"`, !*val*). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *val* (an integer variable) will contain the value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. There are two kinds of localities:

- *Physical localities* are the identifiers through which nodes can be uniquely identified within a net.
- *Logical localities* are symbolic names for nodes. A distinct logical locality, `self`, can be used by processes to refer the node where they are executing on.

Physical localities have an absolute meaning within the net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought of as aliases for network resources. Logical localities are associated to physical localities through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the physical locality of the node. An allocation environment has the shape $\{\ldots, l_i \sim s_i, \ldots\}$, where l_i are logical localities and s_i are physical localities.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can execute the following operations over tuple spaces and nodes:

- **in**$(t)@l$: evaluates tuple t and looks for a matching tuple t' in the tuple space located at l. Whenever a matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available.
- **read**$(t)@l$: differs from **in**$(t)@l$ only because the tuple t' selected by pattern-matching is not removed from the tuple space located at l.
- **out**$(t)@l$: adds the tuple resulting from the evaluation of t to the tuple space located at l.
- **eval**(`proc`)$@l$: spawns process `proc` for execution at node l.
- **newloc**(l): creates a new node in the net and binds its physical locality to l. The node can be considered a "private" node that can be accessed by the other nodes only if the creator communicates the value of variable l, which is the only means to access the fresh node.

X-KLAIM extends KLAIM with the typical constructs of high level programming languages: variable declarations, assignments, conditionals, sequential and iterative process composition. Moreover, X-KLAIM provides specific statements to simplify multiple access to tuple spaces (**forall**). Please notice that, all the X-KLAIM constructs can be *encoded* within KLAIM. However, introducing the new statements in the X-KLAIM syntax makes the programmers life easier.

The implementation of X-KLAIM is based on KLAVA, a Java [5] package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. The

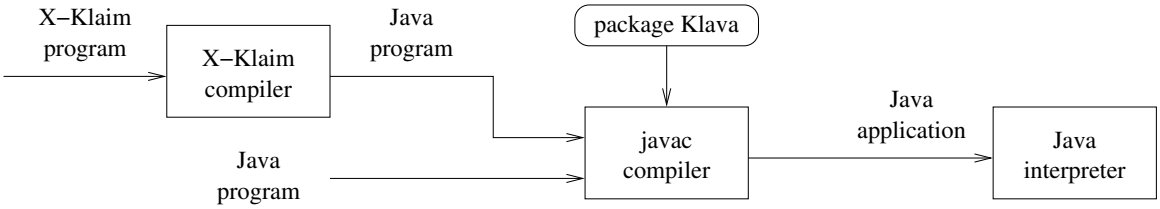


FIG. 2.1. *The framework for* X-KLAIM.

structure of the KLAIM framework is outlined in Figure 2.1. X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the X-KLAIM paradigm. With this respect, by using KLAVA directly, the programmer is able to exchange, through tuples, any kind of Java object, and implement a more fine-grained kind of mobility, as shown in [14]. X-KLAIM provides both *weak mobility* (via operation **eval**) and *strong mobility* (via operation **go**, explained later in this section). Conversely, KLAVA supports weak mobility only; indeed, Java does not allow to save and restore the execution state. X-KLAIM and KLAVA are available on line at `http://music.dsi.unifi.it`. KLAVA is presented in detail in [14, 7].

TABLE 2.1
X-KLAIM *process syntax. Syntax for other standard expressions is omitted.*

RecProcDefs	::=	**rec** id formalparams procbody | **rec** id formalparams **extern**
	|	RecProcDefs **;** RecProcDefs
formalParams	::=	**[ ]** | **[** paramlist **]**
paramlist	::=	id **:** type | **ref** id **:** type | paramlist **,** paramlist
procbody	::=	declpart **begin** proc **end**
declpart	::=	ε | **declare** decl
decl	::=	**const** id **:=** expression | **locname** id | **var** idlist **:** type | decl **;** decl
idlist	::=	id | idlist **,** idlist
proc	::=	KAction | **nil** | id **:=** expression | **var** id **:** type | proc **;** proc
	|	**if** boolexp **then** proc **else** proc **endif**
	|	**while** boolexp **do** proc **enddo**
	|	**forall** Retrieve **do** proc **enddo**
	|	procCall | **call** id | **(** proc **)** | **print** exp
KAction	::=	**out(** tuple **)@**id | **eval(** proc **)@**id | Retrieve | **go@**id | **newloc(** id **)**
Retrieve	::=	Block | NonBlock
Block	::=	**in(** tuple **)@**id | **read(** tuple **)@**id
NonBlock	::=	**inp(** tuple **)@**id | **readp(** tuple **)@**id | Block **within** numexp
boolexp	::=	NonBlock | *standard bool exp*
tuple	::=	expression | proc | **!** id | tuple **,** tuple
procCall	::=	id **(** actuallist **)**
actuallist	::=	ε | expression | proc | id | actuallist **,** actuallist
expression	::=	$*$ expression | *standard exp*
id	::=	*string*
type	::=	**int** | **str** | **loc** | **logloc** | **phyloc** | **process** | **ts** | **bool**

X-KLAIM syntax is shown in Table 2.1. We just briefly recall the more relevant features. Local variables of processes are declared in the **declare** section of the process definition. Standard base types are available (**str**, **int**, etc.) as well as X-KLAIM typical types: **loc** for generic locality variables (without specifying whether it is logical or physical), **logloc** (resp. **phyloc**) for logical (resp. physical) localities, **process** for process variables and **ts**, i. e., tuple space, for implementing data structures by means of tuple spaces, e.g., lists, that can be accessed through standard tuple space operations. Logical locality constants are declared by using the type **locname**. Finally, Comments start with the symbol **#**.

A locality variable can be initialized with a string that will correspond to its actual value. Logical localities are basically names, while physical localities must have the form <IP_address>:<port>, so a physical locality variable has to be initialized with a string corresponding to an Internet address.

Logical locality resolution can be performed by putting the operator $*$ in front of the locality that has to be evaluated:

```
l := *output; # retrieve the physical locality associated to output
out(*output)@self; # insert the physical locality associated to output
```

However, logical localities used as "destination" are still evaluated automatically in both network models, i. e., if the locality used after the `@` is a logical one, it is first translated to a physical locality.

Apart from standard KLAIM operations, X-KLAIM also provides non-blocking version of the retrieval operations, namely **readp** and **inp**; these act like **read** and **in**, but, in case no matching tuple is found, the executing process does not block but `false` is returned. Indeed, **readp** and **inp** can be used where a boolean expression is expected. These variants, used also in some versions of Linda [21], are useful whenever one wants to search for a matching tuple in a tuple space with no risk of blocking. For instance, **readp** can be used to test whether a tuple is present in a tuple space.

A timeout (expressed in milliseconds) can be also specified for **in** and **read**, through the keyword **within**; the operation becomes a boolean expression that can be tested in order to establish if the operation succeeded (these boolean expressions can be combined in order to execute more complex retrieval operations):

```
if in(!x, !y)@l within 2000 then
 # ... success!
else
 # ... timeout occurred
endif
```

Time-outs can be used when retrieving information to avoid that processes block because of network latency or of missing tuples.

X-KLAIM provides the construct **forall** that can be used for iterating actions through a tuple space by means of a specific template. Its syntax is:

```
forall Retrieve do
 proc
enddo
```

We refer the reader to Table 2.1 for the syntax of "Retrieve". The informal semantics of this operation is that the loop body "proc" is executed each time a matching tuple is available. Even duplicate tuples are repeatedly retrieved by the **forall** primitive; it is however guaranteed that each tuple is retrieved only once. Thus, instead of the while-based code above, we write:

```
forall readp(!i, !s)@self do
 out(i + 1, s)@l
enddo
```

Now, if the tuple space contains three matching tuples (of which two are identical): (10, `"foo"`), (10, `"foo"`), (20, `"bar"`), after the execution of the loop instruction the tuple space at `l` will contain the tuples (11, `"foo"`), (11, `"foo"`), (21, `"bar"`).

Notice however that the tuple space is not blocked when the execution of the **forall** is started, thus this operation is not atomic: the set of tuples matching the template can change before the command completes. A locked access to such tuples can be explicitly programmed (see, e.g., Listing 6.3). Our version of **forall** is different from the one proposed in [17] since parallel processes are not created for each retrieved tuple (this would not be consistent with the "iterating" nature of **forall**; a similar functionality could be easily achieved by using **eval** in the loop body). Our **forall** is similar to the **all** variations of retrieval operations in *PLinda* [3].

The **forall** primitive has a different semantics depending on the nature of the retrieval operation: if a blocking action is used the process executing **forall** is blocked until another (never retrieved) tuple becomes available; instead, when a nonblocking action is used, the process exits from the **forall** loop and continues its execution when no other matching tuple is available.

Data structures can be implemented by means of the data type **ts**; a variable declared with such type can be considered as a tuple space and can be accessed through standard tuple space operations, apart from **eval** that would not make sense when applied to variables of type **ts**. Furthermore **newloc** has a different semantics when applied to such a variable: it empties the tuple space.

forall is then useful for iterating through such data structures; for instance the following piece of code transforms a list, stored in the variable `list` of type **ts**, containing data of the shape (**str**, **int**) into a list containing data of the shape (**int**, **str**):

```
declare
  var s : str;
  var i : int;
  var list : ts;
...
forall inp(!s, !i)@list do
  out(i, s)@list
enddo
```

Notice that the non-blocking version of **in** is used, otherwise the process would be blocked when it finishes iterating through the list.

The action **go**@l [9] makes an agent migrate to l and resume its execution at l from the instruction following the migration. This action permits modeling strong mobility. Thus in the following piece of code an agent retrieves a tuple from the local tuple space, then migrates to the locality `l` and inserts the retrieved tuple into the tuple space at locality `l`:

```
in(!i, !j)@self;
go@l;
out(i, j)@self
```

I/O operations are implemented as tuple space operations. For instance the logical locality *screen* is actually attached to the output device. Hence, operation **out**("foo\n")@*screen* corresponds to printing the string "foo\n" on the screen. Similarly, the locality *keyboard* can be attached to the input device, so that a process can read what the user typed with a **in**(!s)@*keyboard*. Further I/O devices, such as files, printers, etc., can also be handled through the locality abstraction.

A node of an X-KLAIM net can be specified as follows:

```
physloc :: { ... , l~ s, ... } init_processes
```

where `physloc` is the physical locality of the node and `{ ... , l ~ s, ... }` is its allocation environment. Notice that `self` is automatically associated to the physical locality of the node and it does not have to be specified in the environment. `init_processes` are the processes executed automatically when the node is started; basically they have the same functionality of `main` in C and Java. Throughout the paper we will omit the definition of nodes when it is not strictly relevant.

3. Mobility Examples. In this section we show a few programming examples taking advantage of process mobility, implemented in X-KLAIM.

News gathering. The first example is a *news gatherer*, that relies on mobile agents for retrieving information on remote sites. We assume that some data are distributed over the nodes of an X-KLAIM net and that each node either contains the information we are searching for, or, possibly, the locality of the next node to visit in the net.

The agent *NewsGatherer* first tries to read a tuple containing the information we are looking for, if such a tuple is found, the agent returns the result back home; if no matching tuple is found within 10 seconds, the agent tests whether a link to the next node to visit is present at the current node; if such a link is found the agent migrates there and continues the search, otherwise it reports the failure back home.

The implementation of the agent exploiting strong mobility (by means of the migration operation **go**) is reported in Listing 3.1. Notice that the use strong mobility makes the source quite clear.

Information retrieval. The next example is still an autonomous information retrieval agent in the context of a virtual *market place*: suppose that someone wants to buy a specific product at a market made of geographically distributed shops. To decide at which shop to buy, she/he activates a migrating agent which is programmed to find and return the name of the closest shop (i. e., the shop within the chosen area, determined by a maximal distance parameter) with the lowest price. The implementation of the agent `MarketPlaceAgent` is shown in Listing 3.2.

The `MarketPlaceAgent` takes as parameters the product name, the maximal distance and the locality where the result of the search must be returned. The agent is sent (by means of an **eval** not shown here) for execution at the node containing the marketplace directory, where it asks for the list of the shops in the selected shopping area. Then, `MarketPlaceAgent` migrates to the first shop in the list. At each shop, `MarketPlaceAgent` checks the price of the wanted product, possibly updating the information about the lowest price and the shop that offers it, and migrates to the next shop in the list. If there are no more shops to visit, `MarketPlaceAgent` sends the result of the search back to the locality received as parameter. The list of nodes to visit is stored in a list (implemented through a **ts**) and **forall** is used for iterating over this list.

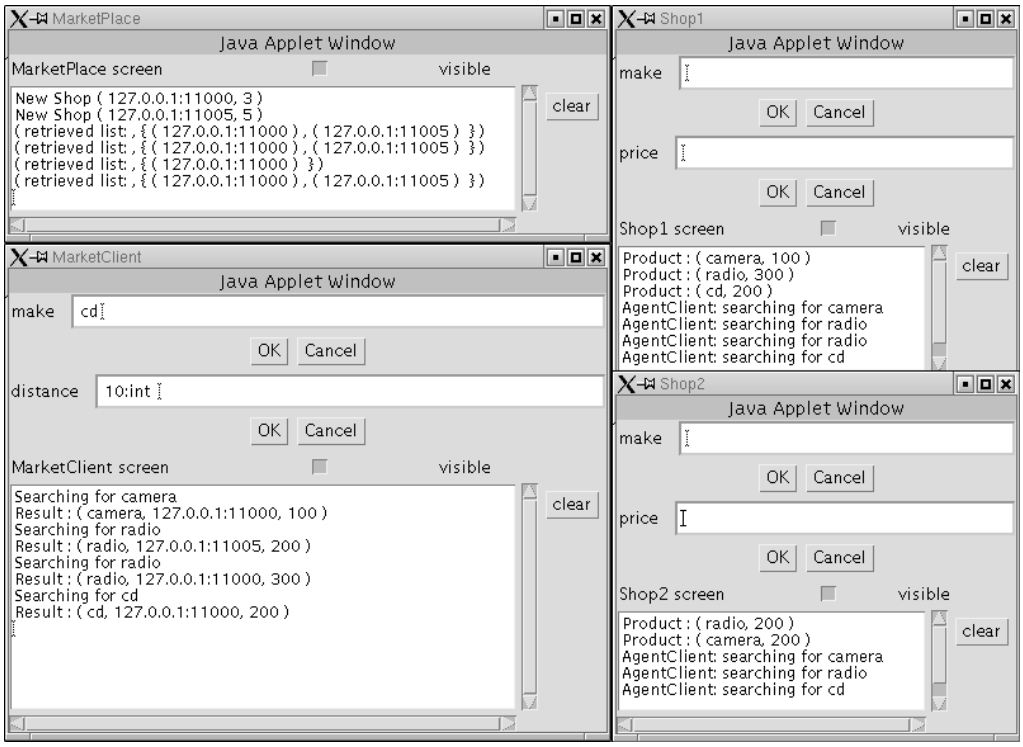
LISTING 3.1
X-KLAIM *implementation of a news gatherer using strong mobility.*

```
rec NewsGatherer[ item : str, retLoc : loc ]
 declare
  var itemVal : str ;
  var nextLoc : loc ;
  var again : bool
 begin
  again := true;
  while again do
   if read( item, !itemVal )@self within 10000 then
    go@retLoc;
    print "found " + itemVal;
    again := false;
   else
    if readp( item, !nextLoc )@self then
     go@nextLoc
    else
     go@retLoc;
     print "search failed";
     again := false
    endif
   endif
  enddo
 end
```

LISTING 3.2
X-KLAIM *implementation of an agent visiting shops of a virtual market place searching for an item with the lowest price.*

```
rec MarketPlaceAgent[ ProductMake : str, retLoc : loc, distance : int ]
 declare
  var shopList : TS ;
  var nextShop, CurrentShop, thisShop : loc ;
  var CurrentPrice, newCost : int ;
  locname screen
 begin
  out( "cshop", distance )@self; # ask for a list of shops within a distance
  in( "cshop", !shopList )@self;
  out( "retrieved list: ", shopList )@screen;
  CurrentPrice := 0 ;
  CurrentShop := self ;
  forall inp( ! nextShop )@shopList do  # while there are shops to visit
   thisShop := nextShop ;
   go@nextShop ; # migrate to the next shop ;
   out( "AgentClient: searching for ", ProductMake )@screen ;
   if read( ProductMake, ! newCost )@self within 10000 then
    if ( CurrentPrice = 0 OR newCost < CurrentPrice ) then
     CurrentPrice := newCost; # update the best price
     CurrentShop := thisShop
    endif
   endif
  enddo ;
  out( ProductMake, CurrentShop, CurrentPrice )@retLoc # OK, let's send the results
 end
```

Screenshot 3.1 shows a client that performs some searches through the `MarketPlaceAgent` in two shops. In this example there are two shops affiliated to the market place: `Shop1` at physical locality `127.0.0.1:11000` with a distance of 3, and `Shop2` at physical locality `127.0.0.1:11005` with a distance of 5; this information is shown in the window of the market place directory (up left). The client sends the agent searching for a `camera` within a distance of 10, so the market place directory provides the agent with a list made of the localities of the two shops, and after visiting both, the agent reports home that the first shops sells the searched item at the lower cost. The second query has basically the same parameters but the agent has to search for a `radio` and this time the second shop sells it at the lower price. Then it still searches for a radio but within a closer distance (e.g., 4) and this time the second shop is not even visited (since its distance is 5, so the market place directory does not put it into the list communicated to the agent). Finally a `cd` is searched for (within a wider distance) and when visiting the second shop a timeout is raised, since that shop does not sell that item.



SCREENSHOT 3.1. *The market place directory (up left), the market client (down left) and two shops of the virtual market place.*

Load balancing. We conclude this section by presenting an example that uses the remote evaluation paradigm, thus, the code does not to autonomously migrate: it is moved by another process. This example implements a *load balancing system* that dynamically redistributes mobile code among several processors: we suppose that remote clients send processes for execution to a server node that distributes the received processes among a group of processors by using, each time, the (estimated) idlest one. Each processor sends a number of "credits" to the server (this number corresponds to the processor availability to perform computations on behalf of the server); the server stores the number of credits in a database and, when needed, it chooses the processor with the highest number of credits and decreases this number.

When a processor receives a process, it immediately starts executing the process (in a parallel thread) and sends a credit back to the server. Indeed, the system is based on the heuristic that if a processor is busy, it cannot send a credit back, or at least it does not send a credit immediately (this is also known as *Leaky Bucket Of Credits* pattern [2]).

This example is implemented by the code fragment in Listing 3.3 that shows the server that dispatches the received process to the idlest processor (left) and the processor that receives a process for execution from the server and sends a credit back to it. The code presented here is simplified in order to concentrate on the code mobility related parts (e.g., it does not handle cases such as all credits are exhausted for all processors). Notice that processes are exchanged by means of **out** and **in**.

4. Node Connectivity in X-KLAIM**.** The original KLAIM model of [23] has been extended in [15] to deal more directly with *open nets*. The original formalism is enriched with explicit connectivity actions and with a new kind of processes, that we called *Node Coordinators*, which are the only ones allowed to perform privileged connectivity actions. This distinction provides a fine-grain separation between the coordination level and the standard action execution level. Furthermore, the allocation environment can be modified dynamically with the primitive **bind**.

In the hierarchical model, any node plays both the role of computational environment (for processes and tuples), and a *gateway*, (for managing subnets of other nodes). Nodes can act both as clients (belonging to a specific subnet) and as servers (taking charge of, possibly private, subnets). Logical localities represent the names that client nodes can specify when entering the subnet of a server node, and allocation environments, that can be dynamically updated with such information, actually represent dynamic tables mapping logical names (possibly not known in advance) into physical addresses; these mappings are allowed to change during the evolution. The client-server relation among nodes smoothly leads to a hierarchical model, also because of the way logical names are "resolved": in order to find the mapping for a

LISTING 3.3

Load balancing: (left) the server receives a process and dispatches it to the idlest processor; (right) the processor node receives a process and executes it locally and sends a credit back to the server.

```
rec DeliverProcess[ ProcessorDB : ts ]
 declare
  var P : process ;
  var HighestCredit, Credits : int ;
  var Processor, HighestProcessor : loc
 begin
  while ( true ) do
   in( !P )@self ; # wait for a process
   HighestCredit := 0 ;
   forall readp( !Processor, !Credits )@ProcessorDB do
    if ( Credits > HighestCredit ) then
     HighestCredit := Credits ;
     HighestProcessor := Processor
    endif
   enddo ;
   out( P )@HighestProcessor ;
   # update its credits
   in( HighestProcessor, HighestCredit )@ProcessorDB ;
   out( HighestProcessor, HighestCredit − 1 )@ProcessorDB
  enddo
 end
```

```
rec ReceiveProcess[ server : loc ]
 declare
  var P : process ;
  locname screen
 begin
  while ( true ) do
   in( !P )@self ;
   eval( P )@self ;
   out( "SERVER", "CREDIT",
       self )@server
  enddo
 end
```

locality, allocation environments of nodes in this hierarchy are now inspected from the bottom upwards. This resembles name resolution within DNS servers. We shall consider further this issue in Section 4, where we will describe how node connectivity is managed in X-KLAIM.

X-KLAIM provides all the primitives for explicitly dealing with node connectivity. Consistently with the hierarchical model of KLAIM such actions can be performed only by *node coordinators*. The syntax of node coordinators is shown in Table 4.1, and is basically the same of standard X-KLAIM processes (Table 2.1) apart from the new privileged actions. We briefly comment these new actions:

TABLE 4.1

X-KLAIM *node coordinator syntax. This syntax relies on standard process syntax shown in Table 2.1.*

NodeCoordinator	::=	**rec** NodeCoordDef
NodeCoordDef	::=	**nodecoord** id formalparams declpart nodecoordbody
	|	**nodecoord** id formalparams **extern**
nodecoordbody	::=	**begin** nodecoordactions **end**
nodecoordaction	::=	*standard process action* | **login(** id **)** | **logout(** id **)**
	|	**accept(** id **)** | **disconnected(** id **)** | **disconnected(** id **,** id **)**
	|	**subscribe(** id **,** id **)** | **unsubscribe(** id **,** id **)**
	|	**register(** id **,** id **)** | **unregister(** id **)**
	|	**newloc(** id **)** | **newloc(** id **,** nodecoordactions **)**
	|	**newloc(** id **,** nodecoordactions **,** num **,** classname **)**
	|	**bind(** id **,** id **)** | **unbind(** id **)**

- **login**(*loc*), where *loc* is an expression of type **loc**, logs the node where the node coordinator is executing at the node at locality *loc*; **logout**(*loc*) logs the node out from the net managed by the node at locality *loc*. **login** can be used as a boolean expression in that it returns **true** if the login succeeds and **false** otherwise.
- **accept**(*l*) is the complementary action of **login** and indeed, the two actions have to synchronize in order to succeed; thus a node coordinator on the server node (the one at which other nodes want to log) has to execute **accept**. This action initializes the variable *l* to the physical locality of the node that is logging. **disconnected**(*l*) notifies that a node has disconnected from the current node; the physical locality of such node is stored in the variable *l*. **disconnected** also catches connection failures. Notice that both **accept** and **disconnected** are blocking in that they block the running process until the event takes place. Instead, **logout** does not have to synchronize with **disconnected**.

An example of these four operations is shown in Listing 4.1, where the node coordinators executing on the client are presented on the left, and the complementary ones executing on the server are presented on the right. Notice that the process that executes the **login** communicates with the one that has to execute the **logout** by using a tuple. **accept** and **disconnected** are initializers for the corresponding variables.

LISTING 4.1
*An example showing **login** and **logout** (left) and the corresponding **accept** and **disconnected**.*

```
rec nodecoord SimpleLogin[ server : loc ]
 begin
  print "try to login to " +
      server + "...";
  if login( server ) then
   print "login successful";
   out("logged", true)@self
  else
   print "login failed!"
  endif
 end

rec nodecoord SimpleLogout[ server : loc ]
 begin
  in("logged", true)@self;
  print "logging off from " +
      server + "...";
  logout(server);
  print "logged off."
 end
```

```
rec nodecoord SimpleAccept[]
 declare
  var client : phyloc
 begin
  print "waiting for clients...";
  accept(client);
  print "client " + client + " logged in"
 end

rec nodecoord SimpleDisconnected[]
 declare
  var client : phyloc
 begin
  print "waiting for disconnections...";
  disconnected(client);
  print "client " + client +
" disconnected."
 end
```

- **subscribe**(*loc*, *logloc*) is similar to **login**, but it also permits specifying the logical locality (*logloc* is an expression of type **logloc**) with which a node wants to become part of the net coordinated by the node at locality *loc*; this request can fail also because another node has already subscribed with the same logical locality at the same server. **unsubscribe**(*loc*, *logloc*) performs the opposite operation.
- **register**(*pl*, *ll*), where *pl* is a physical locality variable and *ll* is a logical locality variable, is the complementary action of **subscribe** that has to be performed on the server; if the subscription succeeds *pl* and *ll* will respectively contain the physical and the logical locality of the subscribed node. The association $pl \sim ll$ is automatically added to the allocation environment of the server. **unregister**(*pl*, *ll*) records the unsubscriptions. Notice that an alternative version of **disconnected**, namely **disconnected**(*pl*, *ll*) is supplied, in order to detect lost connections with nodes, that also specifies the logical locality with which a node was subscribed. As the other **disconnected** explained above, this action is more powerful in that it is able to catch also connections brutally closed without an **unsubscribe**. Let us observe that **disconnected** catches also the events of **unregister** so if program uses both, it is up to the programmer to coordinate the two notification actions (an example of such a scenario is shown in Section 5).

bind(*logloc*, *phyloc*) allows to dynamically modify the allocation environment of the current node: it adds the mapping $logloc \sim phyloc$. On the contrary, **unbind**(*logloc*) removes the mapping associated to the logical locality *logloc*. These two operations privileged and only node coordinators can execute them.

In this version of X-KLAIM **newloc** has become a privileged action and is supplied in three forms in order to make programming easier: apart from the standard form that only takes a locality variable, where the physical locality of the new created node is stored, also the form **newloc**(*l*, *nodecoordinator*) is provided. Since **newloc** does not automatically logs the new created node in the net of the creating node, this second form allows to install a node coordinator in the new node that can perform this action (or other privileged actions).

Notice that this is the only way of installing a node coordinator on another node: due to security reasons, node coordinators cannot migrate, and cannot be part of a tuple. In order to provide better programmability, this rule is slightly relaxed: a node coordinator can perform the **eval** of a node coordinator, provided that the destination is `self`.

Finally the third form of **newloc** takes two additional arguments: the port number where the new node is going to be listening (and this also determines its physical locality, since the IP address will be the same of the creator node), and the (Java) class of the new node. Since I/O devices can be abstracted into nodes, this form of **newloc** enables to construct, for instance, the graphical interface of a node, made up of several I/O sub-nodes. For an example, see Section 5, where some I/O logical localities are used as interfaces for text areas, and input text boxes and lists.

5. A Chat System with Connectivity Actions. In this section we present the implementation in X-KLAIM of a chat system. The chat system we present in this section is simplified, but it implements the basic features that are present in several chat systems. The system consists of a `ChatServer` and many `ChatClients`.

The system is dynamic because new clients can enter the chat and existing clients may disconnect. The server represents the gateway through which the clients can communicate, and the clients logs in the chat server by specifying their "nickname", represented here by a logical locality. A client that wants to enter the chat must subscribe at the chat

LISTING 5.1
Node coordinators of the chat server dealing with clients' subscriptions.

```
rec nodecoord HandleLogin[ usersDB : ts ]
 declare
  var nickname : logloc ;
  var client : phyloc ;
  locname users, screen, server
 begin
  while ( true ) do
   if register( client, nickname ) then
    out( nickname, client )@usersDB ;
    out( true )@client ;
    SendUserList( client, usersDB ) ;
    out( (str)nickname )@users ;
    out( "Entered Chat : " )@screen ;
    out( nickname, client )@screen ;
    BroadCast( "USER", "ENTER",
          nickname, server, usersDB )
   endif
  enddo
 end

rec SendUserList[ newEnter : phyloc, usersDB : ts ]
 declare
  var nickname : logloc ;
  var userLoc : phyloc ;
  var userList : ts
 begin
  newloc( userList ) ;
  forall readp( !nickname, !userLoc )@usersDB do
   if ( userLoc != newEnter ) then
    out( nickname )@userList
   endif
  enddo ;
  out( userList )@newEnter
 end

rec nodecoord HandleDisconnected[ usersDB : ts ]
 declare
  var nickname : logloc ;
  var client : phyloc ;
  locname screen
 begin
  while ( true ) do
   disconnected(client, nickname);
   out("disconnected: ", nickname, client)@screen;
   RemoveClient(nickname, usersDB)
  enddo
 end

rec nodecoord HandleUnregister[ usersDB : ts ]
 declare
  var nickname : logloc ;
  locname screen
 begin
  while ( true ) do
   unregister(nickname);
   out("unsubscription: ", nickname)@screen;
   RemoveClient(nickname, usersDB)
  enddo
 end

rec RemoveClient[ nickname : logloc, usersDB : ts ]
 declare
  var client : phyloc ;
  locname screen, users, server
 begin
  if inp( nickname, !client )@usersDB and
      inp( (str)nickname )@users then
    out( "Left Chat : " )@screen ;
    out( nickname, client )@screen ;
    BroadCast( "USER", "LEAVE",
          nickname, server, usersDB )
  endif
 end
```

server. The server must keep track of all the registered clients and, when a client sends a message, the server has to deliver the message to every connected client. If the message is a private one, it will be delivered only to the clients in the list specified along with the message.

The Chat Server. When a new client issues a subscription request, the server accepts it only if there is no other client with the same nickname, and in case the access is granted, every client is notified about the new client; moreover the new client is also provided with the list of the clients currently in the chat (Listing 5.1). The server keeps a database of all connected clients in a variable `usersDB` of type **ts** where there is a tuple of the shape `(nickname, locality)` for each client, where `nickname` is a logical locality and `locality` is a physical one. Notice that all the processes running on the chat server share this database.

The server uses two (node coordinator) processes for intercepting clients' disconnections: `HandleUnregister` and `HandleDisconnected`. The second one would be useless if the network communications are reliable (i. e., no communication suddenly crashes without further notice); however, this assumption may be too strong in a realistic scenario. Thus `HandleDisconnected` intercepts also this kind of disconnections. As we said above the **disconnected** action returns even after an ordinary unsubscription, so the process `RemoveClient` has to further check whether a client has already been removed from the database.

The broadcasting of messages to clients is managed by two processes running on the `ChatServer` node: `BroadCast` and `BroadCastTo` (Listing 5.2): the former sends a message to all connected clients while the latter sends a message only to the clients specified in the list `to`. This second version is useful when delivering *personal* messages.

All messages have the following tuple shape:

```
(communication_type, message_type, message, from)
```

where `communication_type` and `message_type` specify the type of message (e.g., the values `"USER"` together with `"ENTER"` indicate that a user entered the chat, while `"MESSAGE"` and `"ALL"` indicate a chat message that is destined to every client). `message` is the content of the message (e.g., the nickname of the user that entered the chat or the body of a chat message) and `from` is the nickname (logical locality) of the client that originated the message.

LISTING 5.2
Processes on the server dealing with message dispatching.

```
rec HandleMessage[ usersDB : ts ]
 declare
  var message : str ;
  var sender : logloc ;
  var from : phyloc
 begin
  while ( true ) do
   in( "MESSAGE", !message, !from )@self ;
   if readp( !sender, from )@usersDB then
    BroadCast( "MESSAGE", "ALL",
          message, sender, usersDB )
   endif # ignore errors
  enddo
 end

rec HandlePersonal[ usersDB : ts ]
 declare
  var message : str ;
  var sender : logloc ;
  var from : phyloc ;
  var to : ts
 begin
  while ( true ) do
   in( "PERSONAL", !message, !to, !from )@self ;
   if readp( !sender, from )@usersDB then
    BroadCastTo( "MESSAGE", "PERSONAL",
          message, to, sender, usersDB )
   endif
  enddo
 end

rec BroadCast[ communication_type : str, message_type : str,
          message : str, from : logloc, usersDB : ts ]
 declare
  var nickname : logloc ;
  var user : phyloc
 begin
  forall readp( !nickname, !user )@usersDB do
   out( communication_type, message_type,
      message, from )@user
  enddo
 end

rec BroadCastTo[ communication_type : str, message_type : str,
          message : str, to : ts, from : logloc, usersDB : ts ]
 declare
  var nickname : str ;
  var user : phyloc
 begin
  forall inp( !nickname )@to do
   # recipients are specified as strings in the "to" list
   # so we have to convert them first
   if readp( (logloc) nickname, !user )@usersDB then
    out( communication_type, message_type,
       message, from )@user
   endif
  enddo
 end
```

Messages are received by the chat server by means of two processes `HandleMessage` and `HandlePersonal` (respectively for standard chat messages and for personal messages) also shown in Listing 5.2. When a client wants to send a personal message it has to specify also a list (a **ts** tuple field) containing the nicknames of the clients it is destined to). These processes are responsible for delivering a message to all the recipient clients.

The Chat Client. A chat client executes two processes for handling messages dispatched by the server (Listing 5.3): `HandleMessages` takes care of processing chat messages and `HandleServerMessages` handles server messages informing of new clients joining the chat or existing clients leaving (the list of connected clients is updated accordingly). This information is printed on the screen of the client (attached to the locality `screen`).

The user can insert messages for the server (i. e., commands for entering and exiting from the chat) and standard chat messages in two text fields that are attached, respectively, to the localities `serverKeyb` and `messageKeyb`. For each of these localities there is a process, respectively `HandleServerKeyboard` and `HandleMessageKeyboard` (also in Listing 5.3) that read the input of the user and communicate with the server. When `HandleServerKeyboard` reads a tuple of the shape `("ENTER", nickname)` it tries to subscribe at the chat server with that specific nickname. On the contrary, if the tuple contains `"LEAVE"` it unsubscribes.

A user can specify that a chat message is destined only to a restricted number of clients by selecting them from the list of connected clients. Such list is indeed attached to the locality `usersList` that, in turn, is a special tuple space that provides a sort of interface for accessing the items of such list (in the KLAVA implementation this tuple space is an interface for a `java.awt.List` object). Thus a process can access the elements of such a list through tuples that start with the string `"command"` and consist of a specific command and its arguments. For each command the template of the tuple is different. If the result of a command has to be retrieved the request is issued with an **out** and the response retrieved with an **in**. An identifier has to be provided so that a process does not retrieve the result of the request of another process. For instance the following two lines retrieve multiple selected items in the list (the result is stored in the **ts** variable `selected`):

```
out( "command", "getSelectedItem", ID )@usersList ;
in( "command", "getSelectedItem", ID, !selected )@usersList ;
```

If there is some client selected in this list, the message is sent as `"PERSONAL"` and the list of recipients is sent along with the message; otherwise the message is considered destined to all connected clients.

Screenshot 5.1 shows three chat clients and the chat server.

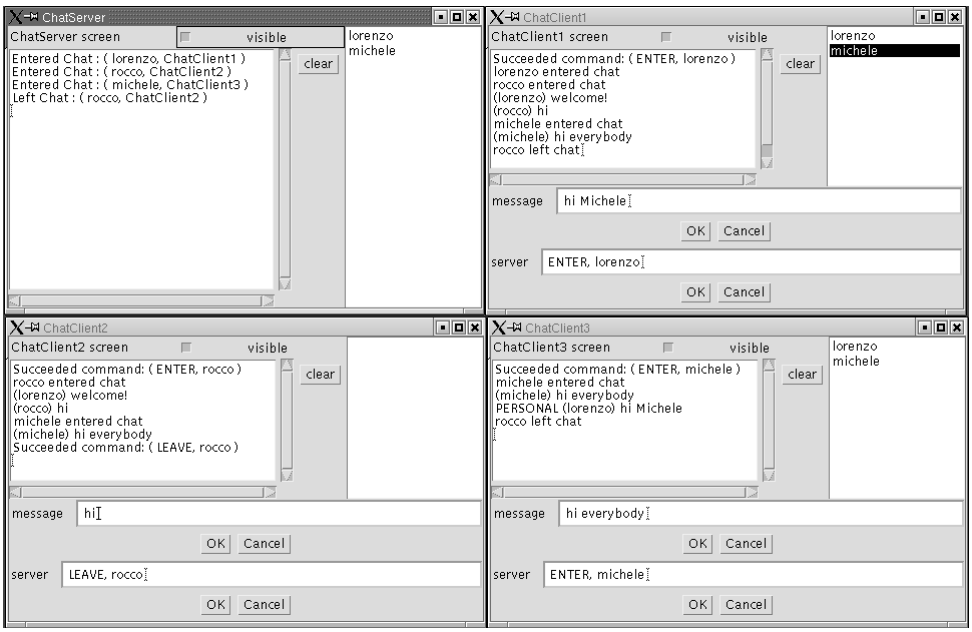
6. A mobile agent based system for document update. In this section, we describe how to use mobile agents to develop a prototype system that permits maintaining *up to date* different documents stored on several heterogeneous

LISTING 5.3
Node coordinators and processes running on a chat client.

```
rec HandleMessages[]
 declare
  locname screen ;
  const standard_message := "MESSAGE";
  var message, message_type : str ;
  var from : logloc
 begin
  while ( true ) do
   in( standard_message, !message_type,
       !message, !from )@self ;
   if message_type = "PERSONAL" then
    out( "PERSONAL " )@screen
   endif;
   out( "(" )@screen ;
   out( (str)from )@screen ;
   out( ") " )@screen ;
   out( message )@screen ; out( "\n" )@screen
  enddo
 end

rec HandleServerMessages[]
 declare
  locname screen, usersList ;
  const user_message := "USER" ;
  var command, nickname : str;
  var from : logloc
 begin
  while ( true ) do
   in( user_message, !command,
       !nickname, !from )@self ;
   if command = "ENTER" then
    out( nickname )@screen ;
    out( " entered chat\n" )@screen ;
    if not readp(nickname)@usersList then
     out( nickname )@usersList
    endif
   else
    if command = "LEAVE" then
     out( nickname )@screen ;
     out( " left chat\n" )@screen ;
     inp( nickname )@usersList
     # ignore non existing names
    endif
   endif
  enddo
 end

rec nodecoord HandleServerKeyboard[]
 declare
  locname server, screen, serverKeyb, usersList;
  var command, nick : str ;
  var nickname : logloc ; var chat_server : phyloc ;
  var response : bool ; var userList : ts
 begin
  chat_server := *server;
  while ( true ) do
   in( !command, !nick )@serverKeyb ;
   if ( command != "ENTER" and command != "LEAVE" ) then
    out( "Unknown command: " )@screen ;
    out( command )@screen ;
    out( "\n" )@screen
   else
    # nick was entered as a string
    nickname := (logloc) nick;
    if command = "ENTER" then
     if subscribe( chat_server, nickname ) then
      out( "Succeeded command: " )@screen ;
      in( !userList )@self ;
      UpdateUserList( userList )
     else
      out( "Failed command: " )@screen
     endif
    else # it is a LEAVE
     unsubscribe( chat_server, nickname ) ;
     out("command", "removeAll")@usersList
    endif ;
    out( command, nickname )@screen
   endif
  enddo
 end

rec HandleMessageKeyboard[]
 declare
  const ID := "messageKeyboard" ;
  var message, selected : str ;
  var selectedUsers : ts ;
  locname messageKeyb, usersList, server
 begin
  while ( true ) do
   in( !message )@messageKeyb ;
   out( "command", "getSelectedItem", ID )@usersList ;
   in( "command", "getSelectedItem", ID, !selected )@usersList ;
   if ( selected != "" ) then
    newloc( selectedUsers ) ;
    out( selected )@selectedUsers ;
    out( "PERSONAL", message, selectedUsers, *self )@server
   else
    out( "command", "getSelectedItems", ID )@usersList ;
    in( "command", "getSelectedItems",
        ID, !selectedUsers )@usersList ;
    if readp( !selected )@selectedUsers then
     out( "PERSONAL", message, selectedUsers, *self )@server
    else
     out( "MESSAGE", message, *self )@server
    endif
   endif
  enddo
 end
```

computers distributed over a network. Documents are installed and updated only on the central server, where clients register for them. When a new version of a document is stored on the server, some agents are scattered along the network to update the corresponding documents on the clients. These mobile agents implement a *push strategy*: subscribed customers are provided with the latest software as soon as it is available.

Upon subscription the client will get the most recent version of the requested documents. Subscription may require a registration and possibly a payment, but we are not addressing these issues, that can be easily added to the system. The delivery of a document and of new versions are made by means of mobile agents, that will migrate to the client's site, and install all the necessary modules. We want to avoid distributed transactions for guaranteeing the correct storage of documents and of new versions. One of the advantages of mobile agents is that they encapsulate transactions, which will take place locally, with no other network connections, apart from the one for sending the agent to a remote computer.



SCREENSHOT 5.1. *Three chat clients and the chat server.*

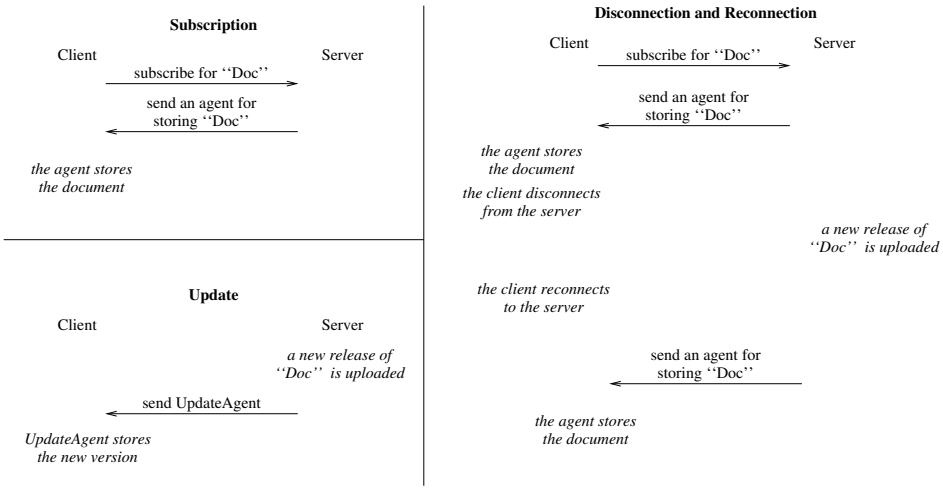


FIG. 6.1. *Subscription and Update*

When the update agent arrives at the client's site, if the document is currently opened, the agent also notifies that a new version of the document is available. Please notice that, the client may decide to disconnect from the server; indeed another advantage of mobile agents is the easy implementation of *disconnected operations*. When the client reconnects to the server, all the documents, which in the meantime have been updated, are sent to the client. Subscription and update (even after disconnection and reconnection) are depicted in Figure 6.1.

6.1. A prototype implementation in X-KLAIM**.** Since this is to be intended as a prototype system, and we are just interested in the design of this kind of applications, not in the details of the implementation, we are not considering advanced features that can be added to the system afterwards. In the X-KLAIM implementation the server waits for connections from clients by executing process `AcceptAgent` (Listing 6.1).

When a new client gets connected with the server, agent `AcceptAgent` verifies if this client has been already registered. Indeed, a list of registered clients is stored in the tuple space `clientlist` where, together with client location, is also maintained the status of the client connection (e.g. `"ON-LINE"` or `"OFF-LINE"`). If the client has been already connected to the server, the status of the connection is changed and a process that updates the documents stored in the client

LISTING 6.1
The process that waits for client connections at the server

```
rec nodecoord AcceptAgent[ ]
 declare
  var client : phyloc;
  locname screen, clientlist
 begin
  while( true ) do
   out( "Waiting for incoming connections...\n" )@screen;
   accept( client );
   out( "Connection established with "+client+"\n")@screen;
   if (inp( client , "OFF-LINE" )@clientlist) then
    eval( CheckForClientUpdate( client ) )@self;
    out( client , "ON-LINE" )@clientlist
   else
    eval( SubscriptionAgent( client ) )@self
   endif
  enddo
 end
```

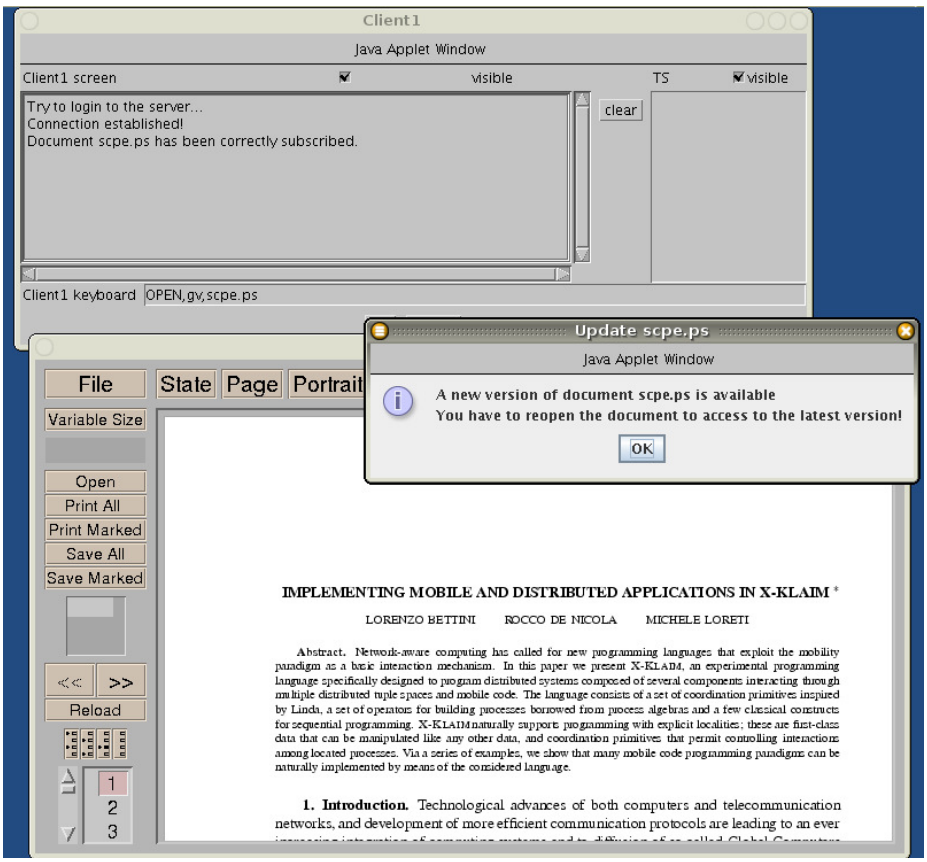
LISTING 6.2
The process that handles document subscriptions

```
rec SubscriptionAgent[ ClientLoc : loc ]
  declare
    locname ClientDB ;
    var Version : int ;
    var foo: int ;
    var Document: process ;
    var DocumentName : str ;
    locname screen
  begin
   while( true ) do
    in( "SUBSCRIBE" , !DocumentName )@ClientLoc;
    # chooses the document, according to DocumentName,
    # and the current version
     if read( DocumentName, !Document , !Version )@ClientDB within 2000 then
      out( "A new request for "+DocumentName+" has been received\n")@screen;
      inp( ClientLoc , DocumentName, !foo )@ClientDB;
      out( ClientLoc , DocumentName, Version )@ClientDB;
      out( "Client "+ClientLoc+
" informations have been successfully stored\n")@screen;
      eval(
         out( DocumentName , true )@self ;
       out( DocumentName, Version )@self;
         eval( Document )@self
      )@ClientLoc;
      out( "Document "+DocumentName+
" has been successfully sent remotely\n")@screen
     else
      out( DocumentName , false )@ClientLoc
     endif
    enddo
  end
```

is executed (see below). Otherwise, when a new client gets connected, its location is stored in the client list. Moreover, process `SubscriptionAgent` is activated Listing 6.2.

When connected, a client can subscribe a document named `docname` by inserting tuple (`"SUBSCRIBE"`, *docname*) in its tuple space. The `SubscriptionAgent` of the client will retrieve such a tuple and, if the document exists, an agent that will store the document is evaluated at client side. The client subscription is also stored at `ClientDB`.

`ClientDB` is a logical locality that is mapped, on the server, to a private physical locality which is known only to the server. It is used to register all the clients and to store information about them (e.g., the documents they subscribed to and



SCREENSHOT 6.1. *A document update*

their current version numbers). Since this locality is not known to the other clients, the server is sure that a client is not able to know the documents installed in another client, and that clients cannot interfere with each other. The secrecy of this locality is obtained by exploiting the locality evaluation mechanism provided by KLAIM. Indeed, that locality will only be mapped to a physical locality dynamically (at run time), through the allocation environment of the server, which is unaccessible by the other nodes.

Each document stored in the server is wrapped inside an agent (`Document` in Listing 6.2). This agent, when executed at a locality, first stores the document into a temp file, hence looks (at `self`) for a tuple of the form:

```
( "OPEN" , app , name )
```

where `app` is the application to use for viewing the document, while `name` is the actual name of the document. For instance, to view a document named `scpe.ps` with `gv`, the following tuple has to be used:

```
( "OPEN" , "gv" , "scpe.ps" )
```

An agent wrapping `document` can receive an update through the tuple

```
("UPDATE",DocName, CurrentVersion, NewVersion)
```

at the private locality. At this point, if the current document is `opened`, the user is notified that a new version of the document is available. Finally, the agent containing the old version of the document produces tuple (`"UPDATE_OK"`, `DocumentName`) and then terminates its execution. The update agent can so store the latest version of the document.

6.2. The update agents. When a new release of a document is installed on the server, by inspecting `ClientDB`, the server will be able to know all the clients that have to be updated, and an update agent is spawned on every such client's site (Listing 6.3).

Upon arrival on the client's site, the update agent (Listing 6.4) first of all verifies that the its version is really new with respect to the one stored locally. If so, it notifies its presence, so that it can be granted permission to update the document. When this update is completed the agent also records that a new version is installed in this node, and then notifies the server that this client has the new version.

LISTING 6.3
The process for spawning an agent on every registered agent.

```
rec CheckUpdate[ ]
 declare
  var DocumentName : str ;
  var Version, ClientVersion : int ;
  var ClientLoc : loc;
  var Document: process;
  var OldDocument: process;
  locname updateKeyb;
  locname ClientDB;
  locname screen
 begin
  while ( true ) do
   in( "STORE" , !DocumentName , !Document )@ClientDB ;
   if inp( DocumentName, !OldDocument, ! Version )@ClientDB then
    Version := Version + 1
   else
    Version := 1
   endif ;
   out( DocumentName, Document , Version )@ClientDB ;
   forall readp( !ClientLoc , DocumentName , !ClientVersion )@ClientDB do
    if Version > ClientVersion then
     eval( UpdateAgent( DocumentName, Version, Document , *self ) )@ClientLoc
    endif
   enddo
  enddo
 end
```

LISTING 6.4
The update agent.

```
rec UpdateAgent[ DocumentName : str, Version : int, Document : process, server : loc ]
 declare
  var CurrentVersion : int;
  var flag: bool
 begin
  in( DocumentName, ! CurrentVersion )@self;
  if ( CurrentVersion < Version ) then
     out( "UPDATE", DocumentName , Version )@self ;
     in( "UPDATE_OK", DocumentName )@self ;
     eval( Document )@self ;
     out( DocumentName, Version )@self ;
     out( "UPDATED", DocumentName, Version, self )@server
  else
     out( DocumentName , CurrentVersion )@self
  endif
 end
```

6.3. Handling client disconnections. Each client can disconnect from the server for work off-line (for instance the client can use a dial-up connection). When off-line, a client will use the local version of the document. Client disconnections are handled by process `DisconnectionManager` (Listing 6.5) that takes care of changing the client status in `clientlist`.

Obviously, when a client works off-line, it cannot receive new updates. However, when a client reconnects to the server, process `AcceptAgent` (Listing 6.1) will execute agent `CheckForClientUpdate` (Listing 6.6) that, by inspecting `ClientDB`, sends to the clients all the documents that have been updated.

7. Implementing Cluedo in X-KLAIM: XKLUEDO. In this section we show how X-KLAIM can be used for developing an agent based implementation of the Cluedo game.

7.1. The game. Cluedo is a crime fiction board game where a set of players have to solve a murder. The board represents a mansion (Figure 7.1 (a)) composed of nine rooms: *Kitchen*, *Ball Room*, *Conservatory*, *Dining Room*, *Billiard*

LISTING 6.5
The agent that handles clients disconnection.

```
rec nodecoord DisconnectionManager[ ]
 declare
  var client : phyloc;
  locname screen, clientlist
 begin
  while (true) do
   disconnected( client );
   out( "Client "+client+" is now disconnected.\n" )@screen;
   in( client , "ON-LINE" )@clientlist;
   out( client , "OFF-LINE" )@clientlist;
  enddo
 end
```

LISTING 6.6
The update procedure after a client connection.

```
rec CheckForClientUpdate[ ClientLoc : loc ]
  declare
    var DocumentName : str ;
    var Version, ClientNum, ClientVersion : int ;
    var Document : process;
    locname ClientDB,screen
  begin
    forall readp( ClientLoc,  !DocumentName, !ClientVersion )@ClientDB do
       read(DocumentName, !Document , !Version )@ClientDB ;
       if (ClientVersion<Version) then
         eval( UpdateAgent( DocumentName, Version, Document , *self ) )@ClientLoc
       endif
    enddo
  end
```

Room, *Library*, *Lounge*, *Hall* and *Study*. Each player represents a character (Miss Scarlet, Professor Plum, Colonel Mustard, Rev. Green, Mrs. White and Mrs. Peacock) that has to discover who killed *Mr. Brown*, the weapon used and the room where murder has taken place. Possible murder weapons are *The Rope*, *The Lead Pipe*, *The Knife*, *The Spanner*, *The Candlestick* and *The Revolver*.

To play the game a pack of cards is used. This consists of 21 cards, each for each room, weapon and character. At the beginning, three cards (one character, one weapon, and one room) are randomly chosen and put into a *special envelope*, so that no-one knows the content of the envelope. The selected cards represent the true facts of the case. The remainder of the cards are distributed among the players.

The aim of the game is to deduce the details of the murder, i. e., the cards in the envelope. To do that, each player announces suggestions to other players, for instance "I suggest it was Mrs. White, in the Library, with the Rope." The other players must then disprove the suggestion, if they can, by showing a card containing one of the suggestion components.

Once a player thinks to know the solution, he/she can make an accusation. The accusing player checks the validity of the accusation by verifying the cards. If the player made a correct accusation, the solution cards are shown to the other players and the game ends. Conversely, if the accusation is incorrect, the player can continue the game but can not submit new accusations (hence he/she cannot win).

7.2. X-KLAIM **implementation.** The idea is to develop an agent-based game where each player executes an agent which migrates among the rooms for implementing a specific game strategy. Please notice that in the standard Cluedo players use dices for moving from a room to another. For the sake of simplicity, we let agent free to move among different rooms.

7.2.1. The pack of cards. Each card is modelled as a tuple composed of two fields $(type, name)$. The former indicates the kind of the card and can assume a value among `"ROOM"`, `"CHARACTER"` and `"WEAPON"`. The latter (*name*) contains the name of the card, e.g., "Conservatory", "Miss Scarlet", "The Revolver", etc. For instance, (`"ROOM"`, `"Kitchen"`) denotes the card corresponding to the *room Kitchen*.

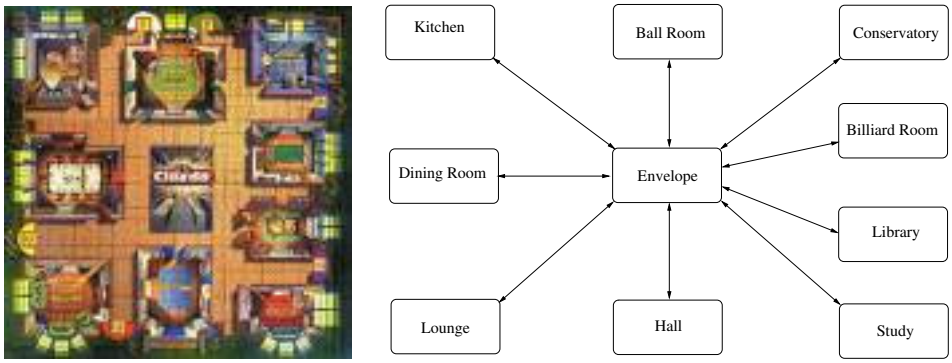


FIG. 7.1. *Cluedo board and* X-KLAIM *topology*

The pack of cards is implemented by means of a tuple space referenced by a local variable of type **ts**. To choose the fact of the case, three cards are retrieved by using X-KLAIM actions and pattern matching. Let `pack` be the tuple space containing all the cards, the following are the instructions initializing variables `room`, `character` and `weapon` that will respectively refer to the place of the murder, the killer and the used weapon:

```
in( "WEAPON" , !weapon )@pack;
in( "ROOM" , !room )@pack;
in( "CHARACTER" , !character )@pack;
```

Each player will receive his/her cards inside a tuple space. Indeed, after a player joins the game, six cards are retrieved from the main `pack` and inserted into a `new` tuple space (`player_pack`). At the end this is sent to the player location (`player_loc`):

```
newloc( player_pack );
out( "COUNT" , 0 )@player_pack;
while (not inp( "COUNT" , 6 )@player_pack) do
  in( !c_type , !c_name )@pack;
  out( c_type , c_name )@player_pack;
  in( "COUNT" , !count )@player_pack;
  out( "COUNT" , count+1 )@player_pack
enddo;
out( "CARDS" , new_board )@player_loc
```

7.2.2. The board. The game board is implemented as an X-KLAIM net containing a node for each room in the mansion. An extra node (named `Envelope`) is used as a central server to which other nodes get connected to take part of the game. The rooms architecture is presented in Figure 7.1 (b).

Incoming connections at `Envelope` are managed by agent `Accept_agent` (Listing 7.1). This agent takes as parameters the cards in the envelope (*room*, *character* and *weapon*) and the main pack of cards.

When a remote node (located at `l`) gets connected with `Envelope`, this agent looks for a tuple indicating the kind of the node. Indeed, two kinds of nodes can get connected with the main node: room nodes, which contain a tuple of the form `("ROOM", name)`, and character nodes, which contain a tuple of the form `("CHARACTER", room)`. If the connected node corresponds to a room, the allocation environment of `Envelope` is updated in order to consider the new binding between the room name with the node location. Please notice that, for the sake of simplicity, we do not consider the case where more nodes try to get connected with the same name.

Agents migrate over the different rooms to acquire the information needed for resolving the case. Agents interact each other by means the tuple spaces located at room nodes by announcing and disproving suggestions. Each agent can make suggestions concerning the room where it is currently located. A suggestion is a tuple of the form (`"SUGGEST"`, *type*, *name*). For instance, tuple (`"SUGGEST"`, `"WEAPON"`, `"Knife"`) in the tuple space located at `Conservatory` indicates that some one suggests that the murder has taken place in the `Conservatory` and that the "Knife" has been used. An agent can disprove a suggestion by removing the corresponding tuple and adding tuple (`"DISPROVE"`, *type*, *name*).

LISTING 7.1
The agent accepting connections at `Envelope`

```
rec nodecoord Accept_agent[ room : str , character : str , weapon : str , board : ts ]
declare
  var l : phyloc;
  var name: str ;
  locname screen, rooms, characters
begin
  while (true) do
    out( "Waiting for new connections...\n" )@screen;
    accept( l );
    out( "New connection established with "+(l)+"\n" )@screen;
    if (inp( "CHARACTER" , !name )@l) then
      out( "PLAYING" , name )@characters;
      out( "Login: "+name+"\n" )@screen;
      eval( Distribute_cards( board , l ) )@self;
      eval( Envelope_agent( room , character , weapon , l , name ) )@self
    else
      if (inp( "ROOM" , !name )@l ) then
        out( "Login: "+name+"\n" )@screen;
        out( name )@rooms;
        bind( name , l )
      else
        out( "Unknown...\n" )@screen
        disconnect( l )
      endif
    endif
  enddo
end
```

To guarantee that one agent at time accesses the tuple space of a `room`, a token is used. Indeed, an agent has to withdraw tuple (`"LOCK"`) from the local tuple space before announces or disproves a suggestion. The token is then reinserted in the tuple space at the end of the operations.

7.2.3. The characters. An X-KLAIM node is associated to each character. The location of these nodes are known only to the agent that accepts the connections. This permits guaranteeing *private* interactions between any character and the main node.

When a character joins the game, i. e., it gets connected with node `Envelope`, `Accept_agent` (Listing 7.1) sends, by using agent `Distribute_cards`, a tuple space containing the cards distributed to the player. After that, process `Envelope_agent` is activated (Listing 7.2).

This agent, which knows the content of the envelope, is waiting for a tuple

```
( "ACCUSE" , !a_room , !a_character , !a_weapon )
```

in the tuple spaces located at the player node.

When a client makes an accusation, `Envelope_agent` verifies its correctness by comparing formal parameters `room`, `character` and `weapon` with retrieved values `a_room`, `a_character` and `a_weapon`. If a player gives the correct accusation he wins the game and the agent registers the player name in the local tuple space. Conversely, if an incorrect accusation has been provided, the agents notifies the player of the mistake and terminates the execution. This way the player cannot win the game anymore. To guarantee that only one accusation for time is considered, tuple `("LOCK")` is used to coordinate the different instances of the agent.

7.2.4. Implementing a player strategy. After a character has received the cards from the `Envelope`, an agent, which implements a specific game strategy, is executed. In Listing 7.3 we present a possible implementation for a simple strategy. This agent migrates over the rooms until it is able to give an accusation. When the agent arrives at a node, it first tries to disprove one of the available suggestions and then formulate a new hypothesis.

This agent has four parameters: `home`, that is a location referring to the player's node, `my_cards`, that is a tuple space containing the player's cards, `suspects` that is a tuple space containing all the suspects (rooms, characters and weapons) that are not yet disproved, and `rooms`, that is a tuple space containing a list of all the rooms. At the beginning `suspects` contains all the cards but those in `my_cards`. Please notice that, an agent can give an accusation when only one card for each type is available in `suspects` tuple space.

LISTING 7.2
The agent that verifies player accusations

```
rec Envelope_agent[ room : str , character : str , weapon : str , l : phyloc , name : str]
 declare
  var a_room : str;
  var a_character : str;
  var a_weapon : str;
  var winner: str;
  locname screen
 begin
  in( "ACCUSE" , !a_room , !a_character , !a_weapon )@l;
  in( "LOCK" )@self;
  if( read( "WINNER" , !winner )@self ) then
    out( "WINNER" , winner )@l
  else
    if ( a_room = room ) and (a_character = character) and (a_weapon = weapon)
    then
     out( "OK" )@l;
     in( "RUNNING" )@self;
     out( "WINNER" , name )@self;
     out( "The winner is "+name )@screen
    else
     out ( "FAIL" )@l
    endif
  endif;
  out( "LOCK" )@self
 end;
```

The agent first retrieves a room, a character and a weapon from `suspects`. Then tests whether other rooms, characters or weapons can be suspected. If there are no other suspects the agent migrates at `home` and makes the accusation. Otherwise, it migrates over the rooms where it makes its suggestions and rejects the ones of other agents.

To migrate to another room each agent first migrates at `Envelope` and then migrates to the locality referenced by name `room`. Indeed, the allocation environment of node `Envelope` stores the binding between the room names and the physical addresses of the corresponding nodes.

When an agent reaches the remote rooms, it first updates the suspect list by removing each card that has been disproved:

```
forall readp( "DISPROVE" , !c_type , !c_name )@self do
  inp( c_type , c_name )@suspects
enddo;
```

After that, the agent tries to disprove one of the available suggestions:

```
flag := true;
while (flag and inp( "SUGGEST" , !c_type , !c_name )@self) do
  if (readp( c_type , c_name )@my_cards) then
    out( "DISPROVE" , c_type , c_name )@self;
    flag := false
  else
    out( "SUGGEST" , c_type , c_name )@self
  endif
enddo;
```

Finally, the agent makes its suggestions by retrieving the corresponding cards from the suspects

```
read( "WEAPON" , !weapon )@suspects;
read( "CHARACTER" , !character )@suspects;
Suggest( weapon , character );
```

Agent `Suspect` is invoked for announcing a suggestion and permits guaranteeing that only a tuple for each suggestion is available in room tuple spaces.

LISTING 7.3
An agent implementing a game strategy

```
rec Player_One[ home : loc , my_cards : ts , suspects : ts ]
 declare
  var room, weapon, character, c_type, c_name : str;
  var flag, try_again : bool;
  var next_room: logloc
 begin
   room := ""; weapon := ""; character := "";
   try_again := true;
   while (try_again) do
    in( "ROOM" , !room )@suspects;
    in( "WEAPON" , !weapon )@suspects;
    in( "CHARACTER" , !character )@suspects;
    # the test below permits verifying if other
    # suspects are available
    if (readp( "ROOM" , !c_name )@suspects or
     readp( "WEAPON" , !c_name )@suspects or
     readp( "CHARACTER" , !c_name )@suspects )
    then
     forall readp( !room )@rooms do
      go@envelope;
      next_room := (logloc) room;
      go@next_room;
      in( "LOCK" )@self;
      flag := true;
      forall readp( "DISPROVE" , !c_type , !c_name )@self do
       inp( c_type , c_name )@suspects
      enddo;
      while (flag and inp( "SUGGEST" , !c_type , !c_name )@self) do
       if (readp( c_type , c_name )@my_cards) then
        out( "DISPROVE" , c_type , c_name )@self;
        flag := false
       else
        out( "SUGGEST" , c_type , c_name )@self
       endif
      enddo;
      read( "WEAPON" , !weapon )@suspects;
      read( "CHARACTER" , !character )@suspects;
      Suggest( weapon , character );
      out( "LOCK" )@self
     enddo
    else
     # In this case the agent can go at home
     # and give the accusation
     try_again := false
    endif
   enddo;
   go@home;
   out( room , character , weapon )@self
 end;
```

8. Conclusions and Related Works. We have presented X-KLAIM, a programming language for implementing distributed applications that can exploit mobile code and run over an heterogeneous network environment. X-KLAIM provides support for moving processes (with strong mobility) and all the code they will need for execution at remote sites. An interesting spin-off of our approach is that, since X-KLAIM is based upon the KLAIM formal model [8], some properties of systems can be formally proved (e.g., in [13] we prove some formal properties of a chat system similar to the one presented in Section 5 and of a mobile agent based software update system). Indeed, a modal logic for KLAIM is being studied [25] and a system to automatically prove KLAIM system properties is under development.

There are currently a number of Java packages, libraries and frameworks that implement functionalities for programming distributed and mobile systems, and that are based on the Linda communication model. In the rest of this section, we review some of them and discuss their relationships with our system.

Jada [22] is a coordination toolkit for Java where coordination and communication among distributed objects is achieved via shared *ObjectSpaces* that are implementations of tuple spaces. Remote access to ObjectSpaces is achieved by specifying the complete IP address and port number, i. e., no locality abstraction is used. Private ObjectSpaces can be dynamically created. No code mobility is supplied by Jada that aims at providing a coordination kernel for implementing more complex Internet languages and architectures.

MARS [18] is a coordination tool for Java-based mobile agents that defines Linda-like tuple spaces programmable to react when accessed by agents. Such a mechanism can be used to control accesses to specific tuples. In X-KLAIM, this is obtained either by using dynamically created private tuple spaces or by adding to the language the capability-based type system presented in [24].

Jini [6] is a connection technology that enables many devices to be plugged together to form a community on a network in a scalable way and without any planning, installation, or human intervention. Each device defines services that other devices in the community may use and drivers that can be downloaded when needed. Jini is developed on top of the *JavaSpaces* [4] technologies, a framework for using Linda-like communication. JavaSpaces introduces some extensions of the Linda original paradigm, such as *event notification*, which allows a process to register its interest in future occurrences of some event and then to receive communication when the event occurs, and *blocking operations with timeouts* and *leasing*, which allows the presence of a tuple in a tuple space, or a notification request, to be granted only for a period of time. Leasing can be obtained also in our language by means of timeouts: a process can sleep for some time (using timeout), and then can take a tuple away from the tuple space (if it is still available). JavaSpaces *transactions* can be programmed in X-KLAIM, by means of dedicated tuples, which represent transaction life time.

IBM *T Spaces* [27] is a network middleware package that supplies tuple space-based network communication with database capabilities; it is implemented in Java by relying on its portability. T Spaces is basically a message processor, in fact a client's view of T Spaces is that of a message center and a message database. A DBMS could be implemented in X-KLAIM by means of a process listening for requests (e.g., SQL strings) passed via tuples, to obtain a similar behavior.

Lime [31] exploits the multiple tuple spaces paradigm [29] to coordinate mobile agents and adds mobility to tuple spaces: it allows processes to have private tuple spaces and to transiently share them. Although in X-KLAIM tuple spaces are bound to nodes and nodes cannot move, processes can have objects of the class `TupleSpace` as data members and, hence, when processes move, `TupleSpace` objects move as well. However, `TupleSpace` objects are never shared and merged automatically.

Systems such as [16, 30, 32, 1], implement strong mobility in Java, by modifying the Java Virtual Machine, to access, save and restore the execution state of threads. However, this solution can jeopardize 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. This is the reason why we preferred not to include strong mobility in KLAVA; however, this feature is available in X-KLAIM and it is implemented on top of KLAVA by means of an appropriate precompilation phase [9].

A feature that is present in systems such as *MARS*, *Lime*, *Sumatra* and *T Spaces*, but not in X-KLAIM, is the ability to react to events such as the insertion of a tuple. This could be programmed by means of a process waiting for a certain tuple, but this does not exactly implement reactions due to the non-determinism in the selection of the process waiting for a tuple.

REFERENCES

[1] A. ACHARYA, M. RANGANATHAN, AND J. SALTZ, *Sumatra: A Language for Resource-aware Mobile Programs*, in Vitek and Tschudin [33], pp. 111–130.

[2] M. ADAMS, J. COPLIEN, R. GAMOKE, R. HANMER, F. KEEVE, AND K. NICODEMUS, *Fault-tolerant telecommunication system patterns*, in Pattern Languages of Program Design 2, J. Vlissides and J. Coplien, eds., Addison-Wesley, 1996, pp. 549–562.

[3] B. G. ANDERSON AND D. SHASHA, *Persistent Linda: Linda + Transactions + Query Processing*, in Proc. of Research Directions in High–Level Parallel Programming Languages, J. P. Banatre and D. Le Metayer, eds., vol. 574 of LNCS, Springer, 1992, pp. 93–109.

[4] K. ARNOLD, E. FREEMAN, AND S. HUPFER, *JavaSpaces Principles, Patterns and Practice*, Addison-Wesley, 1999.

[5] K. ARNOLD, J. GOSLING, AND D. HOLMES, *The Java Programming Language*, Addison-Wesley, 3rd ed., 2000.

[6] K. ARNOLD, B. O'SULLIVAN, R. SCHEIFLER, J. WALDO, AND A. WOLLRATH, *The Jini Specification*, Addison-Wesley, 1999.

[7] L. BETTINI, *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*, PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`

[8] L. BETTINI, V. BONO, R. DE NICOLA, G. FERRARI, D. GORLA, M. LORETI, E. MOGGI, R. PUGLIESE, E. TUOSTO, AND B. VENNERI, *The* KLAIM *Project: Theory and Practice*, in Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Revised Papers, C. Priami, ed., vol. 2874 of LNCS, Springer, 2003, pp. 88–150.

[9] L. BETTINI AND R. DE NICOLA, *Translating Strong Mobility into Weak Mobility*, in Mobile Agents, G. P. Picco, ed., no. 2240 in LNCS, Springer, 2001, pp. 182–197.

[10] ———, *Mobile Distributed Programming in* X-KLAIM, in Formal Methods for Mobile Computing, Advanced Lectures, M. Bernardo and A. Bogliolo, eds., vol. 3465 of LNCS, Springer, 2005, pp. 29–68.
[11] L. BETTINI, R. DE NICOLA, G. FERRARI, AND R. PUGLIESE, *Interactive Mobile Agents in* X-KLAIM, in Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), P. Ciancarini and R. Tolksdorf, eds., IEEE Computer Society Press, 1998, pp. 110–115.
[12] L. BETTINI, R. DE NICOLA, AND M. LORETI, *Software Update via Mobile Agent Based Programming*, in Proc. of ACM SAC 2002, Special Track on Agents, Interactions, Mobility, and Systems, ACM Press, 2002, pp. 32–36.
[13] ———, *Formulae Meet Programs Over the Net: A Framework for Correct Network Aware Programming*, Automated Software Engineering, 11 (2004), pp. 245–288. Special Issue on Distributed and Mobile Software Engineering.
[14] L. BETTINI, R. DE NICOLA, AND R. PUGLIESE, KLAVA*: a Java package for distributed and mobile applications*, Software—Practice and Experience, 32 (2002), pp. 1365–1394.
[15] L. BETTINI, M. LORETI, AND R. PUGLIESE, *An Infrastructure Language for Open Nets*, in Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications, ACM, 2002, pp. 373–377.
[16] S. BOUCHENAK AND D. HAGIMONT, *Pickling Threads State in the Java System*, in Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS), 2000.
[17] P. BUTCHER, A. WOOD, AND M. ATKINS, *Global Synchronisation in Linda*, Concurrency: Practice and Experience, 6 (1994), pp. 505–516.
[18] G. CABRI, L. LEONARDI, AND F. ZAMBONELLI, *Reactive Tuple Spaces for Mobile Agent Coordination*, in Proc. of the 2nd Int. Workshop on Mobile Agents, K. Rothermel and F. Hohl, eds., vol. 1477 of LNCS, Springer, 1998, pp. 237–248.
[19] L. CARDELLI, *Global computation*, in ACM Computing Surveys, 1996. 28(4es), Article 163.
[20] L. CARDELLI, *Abstractions for Mobile Computation*, in Secure Internet Programming: Security Issues for Mobile and Distributed Objects, J. Vitek and C. Jensen, eds., no. 1603 in LNCS, Springer, 1999, pp. 51–94.
[21] N. CARRIERO AND D. GELERNTER, *How to Write Parallel Programs: A Guide to the Perplexed*, ACM Computing Surveys, 21 (1989), pp. 323–357.
[22] P. CIANCARINI AND D. ROSSI, *Jada - Coordination and Communication for Java Agents*, in Vitek and Tschudin [33], pp. 213–228.
[23] R. DE NICOLA, G. FERRARI, AND R. PUGLIESE, KLAIM*: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering, 24 (1998), pp. 315–330.
[24] R. DE NICOLA, G. FERRARI, R. PUGLIESE, AND B. VENNERI, *Types for Access Control*, Theoretical Computer Science, 240 (2000), pp. 215–254.
[25] R. DE NICOLA AND M. LORETI, *A Modal Logic for Mobile Agents*, ACM Transactions on Computational Logic, 5 (2004), pp. 79—128.
[26] D. DEUGO, *Choosing a Mobile Agent Messaging Model*, in Proc. of ISADS 2001, IEEE, 2001, pp. 278–286.
[27] D. FORD, T. LEHMAN, S. MCLAUGHRY, AND P. WYCKOFF, *T Spaces*, IBM Systems Journal, (1998), pp. 454–474.
[28] D. GELERNTER, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, 7 (1985), pp. 80–112.
[29] D. GELERNTER, *Multiple Tuple Spaces in Linda*, in Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89), E. Odijk, M. Rem, and J. Syre, eds., vol. 365 of LNCS, Springer, 1989, pp. 20–27.
[30] H. PEINE AND T. STOLPMANN, *The Architecture of the Ara Platform for Mobile Agents*, in Proc. of the 1st International Workshop on Mobile Agents (MA '97), K. Rothermel and R. Popescu-Zeletin, eds., no. 1219 in LNCS, Springer, 1997, pp. 50–61.
[31] G. PICCO, A. MURPHY, AND G.-C. ROMAN, LIME*: Linda Meets Mobility*, in Proc. of the 21st Int. Conference on Software Engineering (ICSE'99), D. Garlan, ed., ACM Press, 1999, pp. 368–377.
[32] M. RANGANATHAN, A. ACHARYA, S. SHARMA, AND J. SALTZ, *Network-aware Mobile Programs*, in Proc. of the USENIX Annual Technical Conf., USENIX, 1997, pp. 91–103.
[33] J. VITEK AND C. TSCHUDIN, eds., *Mobile Object Systems - Towards the Programmable Internet*, no. 1222 in LNCS, Springer, 1997.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 11, 2006

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 37–51. http://www.scpe.org
ISSN 1895-1767

LEVERAGING STRONG AGENT MOBILITY FOR AGLETS WITH THE *MOBILE JIKESRVM* FRAMEWORK

RAFFAELE QUITADAMO*, LETIZIA LEONARDI* , AND GIACOMO CABRI*

Abstract. Mobility enables agents to migrate among several hosts, becoming active entities of networks. Java is today one of the most exploited languages to build mobile agent systems, thanks to its object-oriented support, portability and network facilities. Nevertheless, Java does not support *strong mobility*, i. e. the possibility of relocating running threads along with their execution state; challenges arising from implementing strong mobility upon the JVM has led to the choice of a weaker form of agent mobility (i. e. *weak mobility*): although in many agent scenarios (e.g. in simple reactive agents) weak mobility could be enough, it usually complicates programming parallel and distributed applications, as it forces developers to structure their agent-based programs as sort of FSMs (Finite State Machine). In this paper, we present our *Mobile JikesRVM* framework to enable strong Java thread migration, based on the IBM Jikes Research Virtual Machine. Moreover, we show how it is possible (and often desirable) to exploit such a framework to enrich a Mobile Agent Platform, like the IBM Aglets, with strong agent mobility and to leverage software agents potential in parallel and distributed computing.

Key words. Java Threads, distributed system, thread migration, strong mobility, IBM JikesRVM

1. Introduction. Agents are autonomous, proactive, active and social entities able to perform their task without requiring a continue user interaction [22]; thanks to the above features, the agent-oriented paradigm is emerging as a feasible approach to the development of today's complex software systems [16].

Moreover, agents can be *mobile*. The concept is simple and elegant: an agent that resides in one node migrates to another node where execution is continued. Code mobility [13] is reshaping the logical structure of modern distributed systems as it enriches software components (in particular, agents) with the capability to dynamically reconfigure their bindings with the underlying execution environment. The main advantages of mobile computations, be they agent-based or not, are as follows:

1. *Load balancing*: distributing agent-based computations among many processors as opposed to one processor gives faster performance for those tasks that can be fragmented.
2. *Communication performance*:agents which interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction.
3. *Availability*: agents can be moved to different nodes to improve the service and provide better failure coverage or to mitigate against lost or broken connections.
4. *Reconfiguration*: migrating agents permits continued service during upgrade or node failure.
5. *Location independence*: an agent visiting a node can rebind to generic services without needing to specifically locate them. Agents can also move to take advantage of services or capabilities of particular nodes.

With regard to mobility, we have to distinguish between *strong mobility*, which enables the migration of code, data and execution state of execution units (for instance, *threads*), from *weak mobility*, which migrates only code and data [13]. From the complexity point of view, *weak mobility* is quite simple to implement using well-established techniques like network class loading or object serialization [29]. However, weakly mobile systems, by definition, discard the execution state across migration and hence, if the application requires the ability to retain the thread of control, extra programming effort is required in order to manually save the execution state. The migration transparency offered by *strong mobility* systems has instead a twofold advantage: it allows a more natural sequential programming style, without the need to awkwardly structure the code with recovery points or flags; moreover, it is more suited to the requirements of many distributed and parallel applications, in which complex computations (e.g. scientific calculations) make manual state capturing (and recovering) somehow unfeasible or, at least, tedious.

Thanks to its portability and network facilities, Java is today the most exploited language to develop mobile agents, and in fact several Java-based Mobile Agent Platforms (MAP) exist [15, 2, 30]. Unfortunately, current standard Java Virtual Machines (JVMs) do not support strong thread migration natively. Thus, despite the advantages above, most mobile agent systems support only weak mobility and the reason lies mainly in the complexity issues of strong mobility and in the insufficient support of existing JVMs to deal with the execution state. Therefore, in order to concretely touch the advantages of strong mobility in mobile agent applications, a suitable framework or "JVM enhancement" is advisable. In this paper, after introducing the motivations for this research work and surveying the related work on this topic (Section 2), we introduce our Java framework (called *Mobile JikesRVM* [23]) to enable thread migration (Section 3), based on the

*Dipartimento di Ingegneria dell'Informazione, University of Modena and Reggio Emilia, Via Vignolese, 905, 41100 Modena, Italy {quitadamo.raffale, leonardi.letizia, cabri.giacomo}@unimore.it

IBM JikesRVM [4]. We found that this JVM offers great support for hosting a strongly mobile agent platform, and we prove this by showing (in Section 4) how the IBM Aglets platform has been successfully adapted to run on top of our mobility framework. Finally, (in Section 5) some first performance evaluation tests are reported. Section 6 concludes the paper and illustrates future research to be done on this framework.

2. Motivations and related work. This section introduces some motivations for our research on strong thread migration and its adoption in some mobile agent scenarios. It sketches some real applications that would benefit from the work explained later and provides a brief overview of proposed approaches in literature.

2.1. Motivations. The choice of strong thread mobility, when designing distributed Java applications, has to be carefully motivated, since it is not always the best one in most simple cases. Distributed and parallel computations can be considered perhaps the "killer application" of such technique.

For instance, complex elaborations, possibly with a high degree of parallelism, carried out on a cluster of servers would certainly benefit from a strong thread migration facility in the JVM. Well-know cases of such applications are mathematical computations, which are often recursive by their own nature (e.g. fractal calculations) and can be parallelized to achieve better elaboration times.

Another field of application for strong mobile threads is load balancing in distributed systems (e.g. in the Grid Computing field), where a number of worker nodes have several tasks appointed to them. In order to avoid overloading some nodes while leaving some others idle (for a better exploitation of the available resources and an increased throughput), these systems need to constantly monitor the execution of their tasks and possibly re-assign them, according to an established load-balancing algorithm. As we will see later, a particular kind of strong thread migration (called *reactive migration*), that we provide in our framework, fits very well the requirements of these systems.

2.2. Related work. Several approaches have been proposed so far to overcome the limitations of the JVM as concerns the execution state management. The main decision that each approach has to take into account is how to capture the internal state of threads, providing a fair trade-off between performances and portability. In literature, we can typically find two categories of approaches:

- modifying or extending the source code of existing JVMs to introduce APIs for enabling migration (*JVM-level approach*);
- translating somehow the application's source code in order to trace constantly the state of each thread and using the gathered information to rebuild the state remotely (*application-level approach*).

JVM-level approach. The former approach is, with no doubt, more intuitive because it provides the user with an advanced version of the JVM, which can completely externalize the state of Java threads (for thread serialization) and can, furthermore, initialize a thread with a particular state (for thread de-serialization). The kind of manipulations made upon the JVM can be several. The first proposed projects following the JVM-level approach like Sumatra [1], Merpati [32], JavaThread [7] and NOMADS [33], extend the Java interpreter to precisely monitor the execution state evolution. They, usually, face the problem of stack references collection modifying the interpreter in such a way that each time a bytecode instruction pushes a value on the stack, the type of this value is determined and stored "somewhere" (e.g., in a parallel stack). The drawback of this solution is that it introduces a significant performance overhead on thread execution, since additional computation has to be performed in parallel with bytecode interpretation. Other projects tried to reduce this penalization avoiding interpreter extension, but rather using JIT (Just In Time) re-compilation (such as Jessica2 [38]) or performing type inference only at serialization time (and not during thread normal execution). In ITS [8], the bytecode of each method in the call stack is analyzed with one pass at serialization time: the type of stacked data is retrieved and used to build a portable data structure representing the state. The main drawback of every JVM-level solution is that they implement special modified JVM versions that users have often to download; therefore they are forced to run their applications on a prototypal and possibly unreliable JVM.

Application-level approach. In order to address the issue of non-portability on multiple Java environments, some projects propose a solution at the application level. In these approaches, the application code is filtered by a pre-processor, prior to execution, and new statements are inserted, with the purpose of managing state capturing and restoration. In fact, the idea of these approaches is to transparently place a few control instructions, similar to recovery-points, that allow a thread to deactivate itself once it has reached one of them. Recovery-points are quite similar to entry points used in most Java MAPs (i. e., methods that are executed when an agent is reactivated at the destination host), even if the former ones enable a finer grain control than entry points. Unluckily, a thread cannot deactivate (or reactivate) itself outside of these recovery-points, which are also not customizable, thus a thread cannot really suspend itself in an arbitrary point of the computation. Some of these solutions rely on a bytecode pre-processor (e.g. JavaGoX [27] or Brakes [36]), while others provide source code translation (e.g. Wasp [12], JavaGo [28], Wang's proposal [37]). Two of them [28, 37] hide a weak

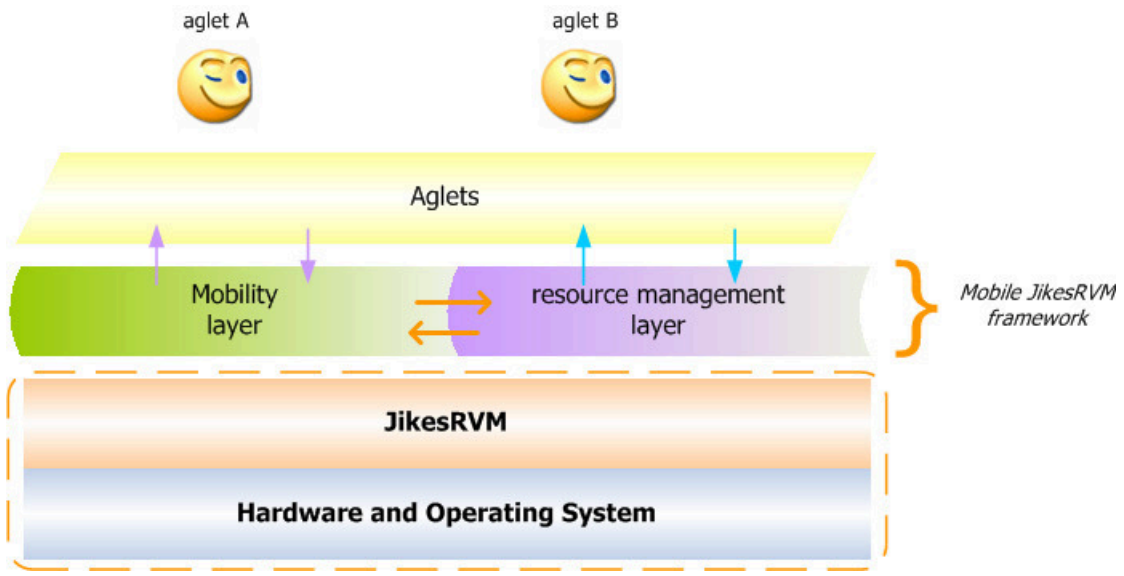


FIG. 3.1. *A layered view of Mobile JikesRVM*

mobility system behind the appearance of a strong mobility one: they, in fact, re-organize "strongly-mobile" written code into a "weakly-mobile" style, so that weak mobility can be used instead. Portability is achieved at the price of a slowdown, due to the many added statements.

Discussion. Starting from the above considerations, we have decided to design and implement a strong thread migration system able to overcome many of the problems of the above-explained approaches. In particular, our framework is written entirely in Java and it does neither suffer performance overheads, due to bytecode instrumentations, nor reliability problems, because the user does not have to download a new, possibly untrustworthy, version of JikesRVM. The framework is capable of dynamically installing itself on several recent versions of JikesRVM (we carried out successful tests starting from release 2.3.2). In fact, every single component of the migration system has been designed and developed to be used as a normal Java library, without requiring rebuilding or changing the VM source code. Therefore, our JikesRVM-based approach can be classified as a midway approach between the above-mentioned JVM-level and Application-level approaches. Other midway approaches [14] exploit the JPDA (Java Platform Debugger Architecture) that allows debuggers to access and modify runtime information of running Java applications. The JPDA can be used to capture and restore the state of a running program, obtaining a transparent migration of mobile agents in Java, although it suffers from some performance degradation due to the debugger intrusion.

3. A Layered View of our Framework. As already stated, in order to successfully exploit the benefits of mobile agents, an efficient and well-designed software support is needed on top of the bare JVM. Such a middleware should provide a precise, though flexible and customizable, answer to the questions of mobile applications developers. Following the seminal work of Fuggetta et al. [13], we can identify three main parts conceptually comprising a mobile code application:

- the *code segment* (i. e. the set of compiled methods of the application);
- the *data space*, a collection of all the resources accessed by the execution unit. In an object-oriented system, these resources are represented by objects in the heap;
- an *execution state*, containing private data as well as control information, such as the call stack and the instruction pointer.

From a mere technological standpoint, the capability to move code and regular objects is already a consolidated matter: the Java language provides very powerful tools to this purpose, like *object serialization* (used to migrate data in the heap) and *bytecode and dynamic class-loaders* (which facilitate the task of moving the code across distant JVMs, hosted by heterogeneous hardware platforms and operating systems). The main problem to tackle here is how to detach the execution state of a Java thread from its native environment and then re-install it at some other site. This requires diving into the internals of the JVM core and externalizing a complete representation of the running thread. Such functionality is provided in our framework by the *mobility layer* in Figure 3.1, which is built just upon JikesRVM. As we stressed earlier, this layer is installed dynamically into the runtime simply importing the *mobility* package, without requiring a

dedicated version of JikesRVM. Further details on this layer and its interactions with JikesRVM are the subject of the next subsection. Shifting to a more application-level point of view, every mobility system (both weak and strong) will sooner or later run across the non-negligible issue of *data space management* [13]: every thread has a set of referenced objects into the heap (i. e. the data space) and, when it migrates to the destination site, the set of bindings to passive (i. e. *resources*) and active objects (i. e. other threads) has to be rearranged. The way this set is rearranged depends on the nature of the resources (whether they can be migrated or not over the network), the type of the binding to such resources, as well as requirements posed by the application. The very fact that it eventually depends on application specific requirements makes it impossible to fully automate the choice of the adequate strategy, entailing the need for its programmatic specification. The *resource management* layer in Figure 3.1 is responsible for handling references to resources and relocating them according to such programmatic specifications. Some ongoing research ideas on this topic are outlined later in subsection 3.2. On top of the *Mobile JikesRVM* framework, it is possible to develop different distributed applications, which can benefit from the provided strong mobility and data space management support. IBM Aglets [2] is a well-known MAP, completely written in Java and open-source project. In Section 4 of this paper, we report on our effort to port this platform on Mobile JikesRVM, so that agents (i. e. aglets) are able to strongly migrate among network nodes: in our opinion, the possibility for agents to exploit the benefits of strong mobility, without many of its well-known drawbacks, can open new applicative scenarios for this paradigm.

3.1. The Mobility Layer. This layer contains a package of classes required to extend the runtime of JikesRVM and enable thread migration on top of it. JikesRVM [4] is now an open-source project, whose innovative and ambitious features are drawing researchers interest from all over the world. JikesRVM began life in 1997 at IBM T. J. Watson Research Center as a project with two main design goals: supporting high performance Java servers and providing a flexible research platform where novel VM ideas can be explored, tested and evaluated [3]. JikesRVM is almost totally written in the Java language, but with great care to achieving maximum performance and scalability exploiting as much as possible the target architectures peculiarities. The all-in-Java philosophy of this VM makes it very easy for researchers to manipulate or extend its functionalities. Furthermore, JikesRVM source code can be built, with a prior custom compilation, both on IA32 and on PPC platforms [17], but the bulk of the runtime is made up of Java objects portable across different architectures. For the sake of brevity, we will focus on those aspects that make JikesRVM an ideal execution environment for strongly mobile agents, overcoming the drawbacks and the limitations of many existing solutions. Further details can be obtained from its users guide [17].

As depicted in the UML excerpt diagram of Figure 3.2, the `MobileThread` class is the basic abstract class, through which thread migration services could be accessed. Users threads have just to subclass `MobileThread` and use some of the inherited methods to extract the execution state (i. e. `collectFrames()`) or to re-install it (i. e. `installFrames()`) at the destination site/host. It must be pointed out that in JikesRVM threads are full-fledged Java objects and are designed explicitly to be as lightweight as possible [3]. As well as many server applications need to create new threads for each incoming request, a Mobile Agent Platform has similar requirements since thousands of agents may request to execute within it.

While some JVMs adopted the so-called *native-thread model* (i. e. the threads are scheduled by the operating system that is hosting the virtual machine), JikesRVM designers chose the *green-thread model* [25]: Java threads are hosted by the same operating-system POSIX thread, implemented by a so-called *virtual processor*, through an object of class `VM_Processor` [5]. Each virtual processor manages the scheduling of its virtual threads (i. e., Java threads), represented by internal objects of the class `VM_Thread`. Moreover, each `java.lang.Thread` has a protected `vmdata` field, pointing to the corresponding instance of `VM_Thread`. When a `MobileThread` is instantiated by the application, it initially points to a standard internal `VM_Thread` object (see the dashed UML composition link between `Thread` and `VM_Thread` in Figure 3.2). This thread becomes truly mobile only when its `enableMobileThread()` method is invoked, since this method changes the reference to the original `VM_Thread` object to an instance of our special `VM_MobileThread` (see the UML composition link between `MobileThread` and `VM_MobileThread` in Figure 3.2).

Before going deeper into the details of the *mobility layer*, we report in Figure 3.3 a possible `migrate()` method, implemented by the user to perform migration of her threads. This method simply opens a socket towards a destination host, captures the execution state of that thread (in a chain of frames, as explained later) and serializes it through the socket stream. Please note that a migration/serialization unit in the example is composed of the thread instance and the chain of frames of its execution state, packed into a dedicated `Transport` object.

Let us suppose we have at destination another service thread, listening on a certain TCP port, whose task is to read incoming threads from the network and resume them locally. A possible method to perform this has been depicted in the excerpt of Figure 3.4. Deserializing the `Transport` object into memory implicitly creates a local instance of the

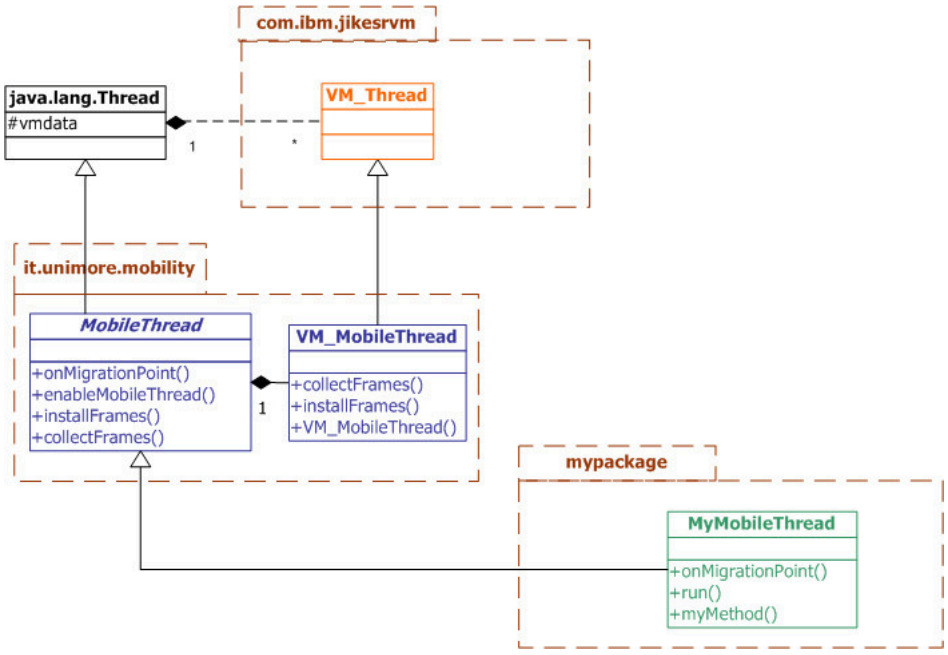


FIG. 3.2. *A simplified view of the mobility package and its dependencies (excerpt)*

```
private void migrate(String hostname, int port)
{
try {
      Socket s = new Socket(hostname, port);

      TransportObject t = new TransportObject();
      t.thread=this;
      t.framesChain=collectFrames(...);

      ObjectOutputStream oos =
              new ObjectOutputStream(s.getOutputStream());
      oos.writeObject(t);
      oos.flush();

      s.close();
      } catch (Exception e) {
      ...
      }
}
```

FIG. 3.3. *Building a mobile thread application with mobility (source machine)*

`MobileThread` object, which has to be manipulated in order to accept the received execution state. The task simply boils down to

- starting the deserialized thread and waiting for its auto suspension,
- installing the received frames in the chain into the suspended thread,
- resuming the thread locally.

If such phases are successfully carried out, the outcome will be that the thread will continue its execution from the next instruction following the `migrate()` method call of Figure 3.3.

```
void handleTransportObject(ObjectInputStream ois, ObjectOutputStream oos)
{
    /* Read the object from the socket */
    Object o = ois.readObject();

    /* Cast the deserialized object to a Transport object */
    TransportObject t = (TransportObject) o;

    /* Create a new autosuspending MobileThread */
    MobileThread newThread = t.thread;

    /* Make this thread autosuspended...  */
    newThread.enableMobileThread(true);
    newThread.start();

    /*... and wait for its suspension */
    while(!newThread.isAutoSuspended())
       Thread.yield();

    /* Install frames into the new thread */
    newThread.installFrames(t.framesChain);

    /* Resume the thread locally */
    newThread.resume();

}
```

FIG. 3.4. *Building a mobile thread application with mobility (destination machine)*

Capturing the execution state of a thread. When the above `collectFrames()` method is called on a `MobileThread` object, the framework starts a walk back through its call stack, from the last frame to the `run()` method of the thread. This jumping is shown schematically in Figure 3.5, where the stack is logically partitioned into three areas: (i) *internal preamble frames*, which are always present and do not need to be migrated; (ii) *user-pushed frames*, to be fully captured as explained later; (iii) *thread-switch internal frames*, which can be safely replaced at the destination and, thus, not captured at all. A special utility class, called `FrameExtractor`, has been implemented in the *mobility* framework, with the precise goal of capturing all the frames in the user area in a portable bytecode-level form. This class uses an *OSR extractor* to capture the frame state representation and returns it to the caller, ready to be serialized and sent to destination or to be check-pointed on disk.

The JikesRVM OSR Extractor. The OSR (On-Stack Replacement) extractor is another fundamental component of the framework: it draws inspiration from the OSR extractors provided by JikesRVM [11], though it has been re-written for the purposes of our project. The OSR technique was introduced in JikesRVM, with a completely different objective: enabling adaptive re-compilation of hot methods. In fact, JikesRVM can rely not only on a baseline compiler but also on an optimized one [9]. Every bytecode method is initially compiled with the baseline compiler, but when the Adaptive Optimization System (AOS) [6] decides that the current executing method is worth being optimized, the thread is drawn from the ready queue and the previous less-optimized frame is replaced by a new more-optimized frame. The thread is then rescheduled and continues its execution in that method. This technique was first pioneered by the Self programming language [10]. An innovative implementation of the OSR was integrated into the JikesRVM [11], which uses source code specialization to set up the new stack frame and continue execution at the desired program counter. The transition between different kinds of frames required the definition of the so-called *JVM scope descriptor* that is the compiler-independent state of a running activation of a method based on the stack model of the JVM [21]. When an OSR is triggered by JikesRVM, the scope descriptor for the current method is retrieved and is used to construct a method, in bytecode, that sets up the new stack frame and continues execution, preserving semantics.

Our modified OSR Extractor. The JikesRVM OSR frame extractor has been rewritten for the purpose of our mobility framework (i. e. the `OSR_MobilityExtractor`) to produce a frame representation, suitable for a thread migration context.

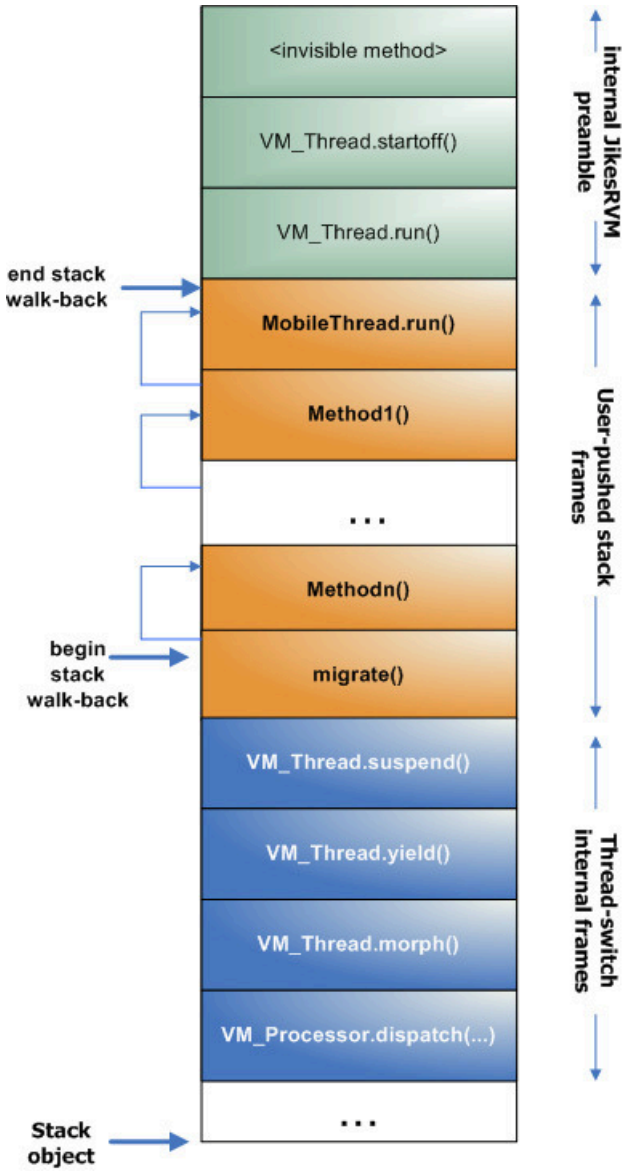


FIG. 3.5. *The stack walk-back of a suspended* `MobileThread`

The scenario we are talking about is a wide-opened one, where different machines running JikesRVM mutually exchange their `MobileThread` objects without sharing the main memory. We introduced, therefore, a portable version of the scope descriptor, called `MobileFrame`, whose structure is reported in Figure 3.6. While the OSR implementation in JikesRVM uses an internal object of class `VM_NormalMethod` to identify the method of the frame, we cannot make such an assumption; the only way to identify that method is through the triplet [method name, method descriptor, full class name] that is universally valid. This triplet (represented by the three fields `methodName`, `methodDescriptor` and `methodClass` in Figure 3.6) is used to refer the method at the destination (e.g. its bytecode must be downloaded if not locally available yet), maybe after a local compilation. The bytecode index (i. e. the `bcIndex` field) is the most portable form to represent the return address of each method body and it is already provided in JikesRVM by default OSR. Finally, we have two arrays (i. e. the `locals` and `stack_operands` fields) that, respectively, contain the values of local variables (including parameters) and stack operands in that frame. These values are extracted from the physical frame at the specified bytecode index and converted into their corresponding Java types (`int`, `float`, `Object` references and so on). In addition, it must be pointed out that the `OSR_MobilityExtractor` class fixes up some problems that we run across during our implementation: here, we think it is worthwhile mentioning the problem of uninitialized local

```
class MobileFrame {
    /** Name of the method which adds this frame*/
    public String methodName;

    /** Method descriptor
    e.g. (I)V for a method
    getting an integer and returning void */
    public String methodDescriptor;

    /** Fully qualified method class
    (e.g.mypackage.myClass)*/
    public String methodClass;

    /** The bytecode index (i.˜e. return address)
    within this method*/
    public int bcIndex;

    /** The local bytecode-level local variable
    including parameters */
    public MobileFrameElement[] locals;

    /** The value of the stack operands at the
    specified bytecode index */
    public MobileFrameElement[] stack_operands;

    // methods and static fields omitted
}
```

FIG. 3.6. *The main fields of the* `MobileFrame` *class*

variables. Default OSR extractor does not consider, in the JVM scope descriptor, those variables that are not active at the specified bytecode index. Nevertheless, these local variables have their space allocated in the stack and this fact should be taken into account when that frame is re-established at the destination.

To summarize, in our mobility framework, threads are serialized in a strong fashion: the `MobileThread` object is serialized as a regular object, while the execution state is transferred as a chain of fully serializable `MobileFrame` objects.

Resuming a migrated thread. The symmetrical part of the migration process is the creation, at the destination host, of a local instance of the migrated thread. This task should be appointed to some user listener thread like the one mentioned above, while in this section we are going to see how the thread is rebuilt in the *mobility layer*. This phase assumes that the target thread is suspended: this allows the infrastructure to safely reshape the current stack object of this thread, injecting one by one all the frames, belonging to the arrived thread. In more details, a new stack is allocated and it is filled in with the thread-switch internal frames, taken from the auto-suspended thread. Then, every `MobileFrame` object is installed, in the same order as they were read from the socket stream (i. e. from the `Methodn()` to `run()`, looking at Figure 3.5). The brand-new stack is closed with the remaining preamble frames, again borrowed from the auto-suspended thread. Now, the new stack has been prepared and the context registers are properly adjusted (pointers are updated to refer to the new stack memory). This stack takes the place of the old stack belonging to the auto-suspended thread (the old one is discarded and becomes garbage). The new `MobileThread` object, with its execution state completely re-established, can be transparently resumed and continues from the next instruction.

Proactive migration vs. reactive migration. In the previous code example, we have shown a kind of migration that has been defined [13] as *proactive migration*: i. e. the mobile thread autonomously determines the time and destination for its migration, explicitly calling a `migrate(URL destination)` method; another interesting, though quite tricky, kind of thread migration is *reactive migration*, where the threads movement is triggered by a different thread that can have some kind of relationship with the thread to be migrated, e.g. acting as a manager of roaming threads. Exploiting JikesRVM features, we successfully implemented both migration types, in particular the *reactive migration*. As anticipated in Subsection 2, an application, in which reactive migration can be essential, is a load-balancing facility in a distributed system.

If the virtual machine provides such functionality to authorized threads, a load monitor thread may want to suspend the execution of a worker thread A, assign it to the least overloaded machine and resume its execution from the next instruction in As code. This form of transparent externally-requested migration is harder to implement with respect to the proactive case, mainly because of its asynchronous nature. Proactive migration raises, in fact, less semantic issues than the reactive one, though identical to the latter from the technological/implementation point of view: in both cases we have to walk back the call stack of the thread, extract the meaningful frames and send the entire thread data to destination (as explained earlier). The fundamental difference is that proactive migration is synchronized by its own very nature (the thread invokes `migrate()` when it means to migrate), while for reactive migration the time when the thread has to be interrupted could be unpredictable (the requester thread notifies the migration request to the destination thread, but the operation is not supposed to be instantaneous). Therefore, in the latter case, the critical design-level decision is about the degree of asynchronism to provide. In a few words, the question is: should the designated thread be interruptible anywhere in its code or just in specific safe migration points? We chose to provide a coarse-grained migration in the reactive case. Our choice has a twofold motivation: (i) designing the migration facility is simpler; (ii) decreasing migration granularity reduces inconsistency risks. Although these motivations can be considered general rules-of-thumb, they are indeed related to the VM we adopted. In fact, the scheduling of the threads in JikesRVM has been defined as *quasi-preemptive* [5], since it is driven by JikesRVM compilers. As mentioned, JikesRVM threads are objects that can be executed and scheduled by several kernel-level threads, called *virtual processors*, each one running on a physical processor. What happens is that the compiler introduces, within each compiled method body, special code (*yieldpoints*) that causes the thread to request its virtual processor if it can continue the execution or not. If the virtual processor grants the execution, the virtual thread continues until a new yieldpoint is reached, otherwise it suspends itself so that the virtual processor can execute another virtual thread. In particular, when the thread reaches a certain yieldpoint (e.g. because its time slice is expired), it prepares itself to dismiss the scheduler and let a context switch occur. If we allow a reactive migration with a too fine granularity (i. e. potentially at any yieldpoint in threads life), inconsistency problems will almost surely occur. The thread can potentially lose control in any methods, from its own user-implemented methods to internal Java library methods (e.g. `System.out.println()`, `Object.wait()` and so forth). It may occur that a critical I/O operation is being carried out and a blind thread migration would result in possible inconsistency errors. We are currently tackling the reactive migration issues thanks to JikesRVM yieldpoints and the JIT compiler. In order to make mobile threads interruptible with the mentioned coarse granularity, we introduced the *migration point* concept: migration points are always a subset of yieldpoints, because they are reached only if a yieldpoint is taken. The only difference is that migration points are inserted only:

1. in the methods of the `MobileThread` class (*by default*);
2. in all user-defined class implementing the special `Dispatchable` interface (*class-level granularity*);
3. in those user-methods throwing `DispatchablePragmaException` (*method-level granularity*).

The introduction of a migration point forces the thread to check also for a possibly pending migration request. If the mobile thread takes the migration point, it invokes a special abstract handler method (i. e. the `onMigrationPoint()` of Figure 3.2) of the `MobileThread` class and this method is responsible for carrying out user-specific migration, as we exemplified in Figure 3.3 and Figure 3.4. This approach has several advantages: firstly, it rids us of the problem of unpredictable interruptions in internal Java library methods (not interested by migration points at all); then, it also gives the programmer more control over the migration, by letting her select those safely interruptible methods; last but not least, it leaves the stack of the suspended thread in a well-defined state, making the state capturing phase simpler. We achieved the insertion of migration points, simply substituting at runtime the method of the JIT compiler object, responsible for inserting yield points, with our migration points insertion method (the source code of the VM is left untouched and one can use every OSR-enabled version of the JikesRVM). We must point out that JikesRVMs compiler does not allow unauthorized users code to access and patch internal runtime structures. Users code, compiled with a standard JDK implementation, will not have any visibility of such low-level JikesRVM-specific details.

3.2. The resource management layer. The set of all referenced objects of a thread has been previously defined as its *data space* [13] and, at any point during the execution, is composed of all the objects that can be reached by the thread through the call stack or its fields. As concerns the stack, the space that the thread is supposed to bring with itself comprises all the objects pointed by the parameters and local variables of methods, together with those objects pushed on the operand stack of each frame in the stack. In the previous section, it was explained how the problem of collecting object references in stack frames has been easily tackled by means of the JikesRVM OSR extractor. The next step, as concerns the data space, will be dealing with objects relocation and reference rebinding. Although such issue pertains more to the application than to the thread migration middleware, we claim that its importance demands some kind of tool

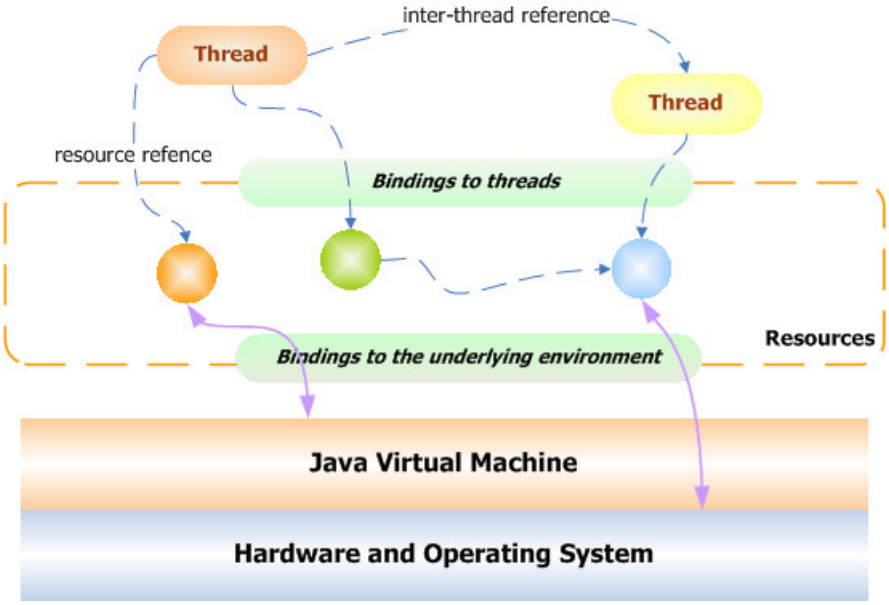


FIG. 3.7. *A conceptual view of resources and threads*

or support from a framework layer, in order to present a coherent set of mobile computing abstractions. In this section, we briefly sketch our vision of the problem and propose some ideas that are nonetheless part of our future research work. Our conceptual view of resources is depicted in Figure 3.7: Java threads can have references to either active (i. e. other threads) or passive resources (i. e. regular Java objects in the heap). The bindings to needed resources must be properly rearranged to maintain accessibility and consistency when the computation migrates to new locations. This poses two kinds of problems:

1. *handling the bindings of resources to their underlying execution environment*. This is not a problem if we consider only resources, like pure Java objects, which are not bound to any OS physical entity; on the contrary, resources, such as files, sockets or database objects, cannot be barely serialized without carefully managing their binding to the underlying environment.
2. *handling the binding of resources to migratory threads*. Fuggetta et al. [13] identified three typologies for this bindings (by identifier, by value or by type) and proposed some relocation strategies for each of them (by move, by copy, network reference, rebinding).

As for the first point, it must be pointed out that moving some resources (e.g. a centralized database) may not be technically (e.g. the bandwidth is not enough for its size) or semantically (e.g. it is already in use for queries by other threads) possible. We think that such issues should be coped with by explicitly introducing the *Resource* concept in our programming model and letting the programmer specify the right policy for her resources. Introducing the `Resource` entity as an interface, the programmer will be asked to make its resource objects implement such interface, together with a set of useful methods for:

- extracting the resource from its environment in a portable/serializable format (if the resource is fixed an exception will be raised and caught by the framework);
- attaching the resource to the destination environment;
- performing a correct cleanup of the resource, if it is detached from the source JVM (see the proposal by Park and Rice [26]).

A simple example of a resource can be a `java.io.File` object. A mere serialization of such an object will not produce the actual movement of the underlying file system object. To accomplish this task, the programmer has to introduce its `MovableFile` object, inheriting from `File` and implementing the `Resource` interface, with some of the methods listed above: in particular, calling the extraction method will likely return a `byte[]` filled in with the file content; calling the attach method will recreate that file in the file system at destination, with its previous content. Moreover, this part of the resource management layer is responsible for properly handling other non negligible issues, pertaining to *dependencies among resources*, *protection* and *concurrency* (e.g. the resource is shared among threads, it is synchronized with a lock and so forth), *inter-thread references*. Focusing on the second point above, the problem of the *bindings between resources*

```
public class MyAgent extends Aglet{
    protected boolean migrated = false; //indicates if the agent has moved yet
    public void run(){
        if( ! migrated ){
            // things to do before the migration
            migrated = true;
            try{ dispatch(new URL(atp://nexthost.unimore.it);}
            catch(Exception e){ migrated = false; }
        }
        else{ // things to do on the destination host }
    }
}
```

FIG. 4.1. *An example of* `Aglet` *with a single migration*

and migratory threads should be addressed [34]. The choice of the right re-binding strategy depends on several factors, from runtime conditions and access-device properties to management requirements and user properties. For instance, a fixed server with no strict constraints on network bandwidth or memory could copy or move the needed resources and work on them locally, whereas a wireless-enabled laptop might want to access that resource remotely without moving it. However, the programming language adopted usually determines the binding strategy (apart from heavily restricted cases, like in Java RMI). Moreover, the strategy is typically embedded within the mobile application code, thus limiting *binding-management flexibility*. We envision that the resource management layer ought to give the programmers the means to specify which reference management policy [24] to use, on a per-instance basis. Furthermore, since the semantics of a given strategy is the same whatever the resource is (e.g. network reference, rebinding, etc.), strategies should be implemented as *basic blocks* that can be reused and programmatically attached to any object, thus achieving a clear and beneficial *separation of concerns* [18] (i. e. between application/functional and non functional/rebinding concerns). Providing an effective and clear support for such abstractions on top of the JikesRVM is part of our future work on this topic.

4. Strong Mobility in Aglets. This section offers an example of a testbed application that we have implemented on top of the *Mobile JikesRVM* framework. It consists of the well-known IBM Aglets MAP, which provides only weak mobility support to mobile agent applications. Simply modifying some parts of its Java source code, we succeeded in implementing a porting of this MAP endowed with strong agent mobility.

4.1. Overview of the Aglets Workbench. The Aglets Workbench [2] is a project originally developed by the IBM Tokyo Research Laboratory with the aim of producing a platform for the development of mobile agent based applications by means of a 100% Java library. The Aglets Workbench provides developer with applet-like APIs [19], thus creating a mobile agent (called *aglet*) is a quite straightforward task. It suffices to inherit from the base class Aglet and to override some methods transparently invoked by the platform during the agent life. Weak mobility is provided through the Java serialization mechanism, and a specific agent transfer protocol (ATP) has been built on top of such mechanism [20]. Each `Aglet` can exploit the special method `dispatch()` to move to another host.

As many other Java MAPs, Aglets exploits weak mobility, that means, from a programming point of view, that each time an agent is resumed at a destination machine, its execution restarts from a defined entry point, that is the `run()` method call. Due to this, dealing with migrations is not always trivial, and developers have to adopt different techniques to handle the fact an agent will execute several times the same code but on different machines. Even if the Aglets library provides a set of classes that helps dealing with migrations, the code will appear like the one shown in the simple example of Figure 4.1. There, in case of a single migration, the `migrated` flag is used to select a code branch for the execution either on the source or on the destination machine. The code of Figure 4.1 is just a simple example, but more complex agents follow the same programming style. In all such cases the point is that with weak mobility it is as the code routinely performs rollbacks. In fact, looking at the code in Figure 4.1, it is clear how, after a successful `dispatch(..)` method call that causes the agent migration, the code does not continue its execution in the `run()` method from that point. Instead, the code restarts from the beginning of the run method (on the destination machine, of course), and thus there is a code rollback. The fact that an agent restarts its execution always from a defined entry point, could produce awkward solutions, forcing the developer to use flags and other indicators to take care of the host the agent is currently running on.

```
public class MyAgent extends Aglet{
    public void run(){
        // things to do before the migration
        try{
            migrate(new URL(atp://nexthost.unimore.it);
        }catch(Exception e){  }
        // things to do after migration
    }
}
```

FIG. 4.2. *An example of* `Aglet` *code using Mobile JikesRVM*

4.2. Implementing Strong Mobility. Our major aim here has been to realize the idea of an Aglet that is to be executed by a strong migratory thread and this required some modifications to the underlying Aglets execution model. In particular, instead of using one of the pre-created threads (i. e., thread pool) to execute the methods of the aglets, JikesRVM makes feasible to have a single independent thread for each aglet. As already mentioned, this is possible because of the lightweight implementation of Java threads in that JVM, being targeted to server architectures, where scalability and performance are key requirements. Furthermore, having a separate thread for each aglet ensures a high level of *isolation* between agents: consider, for example, the case where an agent wants to sleep for some time, without being deactivated (i. e. serialized on the hard disk). Using the classical `sleep()` method on the `java.lang.Thread` object will produce strange effects on the current Aglets implementation platform (such as locking the message passing mechanism). These shortcomings are due to the thread sharing among multiple agents through the pool of threads. Instead, potentially dangerous actions by malicious (or bugged) aglets do not affect the stability of our platform, allowing possibly a clean removal of the dangerous agent without the need of a MAP reboot. In our prototypal implementation, there is only one thread responsible for handling the messages posted to the aglet and this thread will invoke the appropriate handler function to perform the necessary actions in response to the delivered message. In the official Aglets framework, the thread running into the handler function cannot be interrupted asynchronously by a migration request, notified by another thread by means of the *dispatch* message. In our prototype, the dispatch message has the effect of interrupting/preempting the execution of the function (in particular, the handler of the run message, i. e. the `run()` method) and migrating the aglet to the designated host. The `OnDispatching()` handler method is executed to allow preparatory actions to be done, but the current execution stack is preserved, together with local variables, stack operands and method parameters. There is no more need for saving intermediate results into serializable fields or structuring the code with entry points from which the agent execution is restarted each time it arrives at a new host. Referring to the code example of Figure 4.1, the adoption of strong thread mobility overtakes the mentioned drawbacks, since the code restarts at the destination machine from the same point it has stopped at the source one. Thus the code shown in Figure 4.1 becomes the one of Figure 4.2. As readers can see, the code is simpler (no flags and branches are required) and shorter than the previous one.

This kind of message-driven strong mobility is achieved serializing the aglet object and its fields but also appending the sequence of stack frames (as we have explained in Section 3.1) representing the state of the execution at the time of the suspension. Reactive migration has been achieved exploiting the migration point concept provided by the mobility layer (see subsection 3.1) underneath. On the other hand, the de-serialization process involves

1. reading the aglet object from the network stream into the memory;
2. creating a new thread for this aglet or acquiring an existing one, if available;
3. notifying this thread of the arrival event and suspending its execution;
4. injecting on the fly all the migrated frames into its stack;
5. resuming the execution of the thread/aglet transparently.

The migrated aglet will be, by default, destroyed in the source JVM and its associated thread added back to the thread pool, if available. Nevertheless, the *dispose* message can be explicitly intercepted by the programmer so that the aglet can continue executing, thus realizing a form of agent cloning. In summary, the new Aglets implementation tries to overcome the drawbacks of weak agent mobility, using the thread migration facilities of the underlying *mobility layer* and hopefully the resource abstractions of the *resource management layer*.

5. Performance and discussions. At the current stage of our research, the mobility layer of our framework has been successfully tested, focusing mainly on the state capturing and restoring of the threads executing the aglets. First of all, we made some first performance tests (running some simple agent applications) to discover possible bottlenecks and evaluate

the cost of each migration phase. The times measured are expressed in seconds and are average values computed across multiple runs, on a Pentium IV 3.4Ghz with 1GB RAM on JikesRVM release 2.4.5. Thanks to a Fibonacci recursive algorithm we were able to test thread serialization with increasing stack sizes (5, 15 and 25 frames) and found a very graceful time degradation. These times are conceptually divided into two tables, where Table 5.1 refers to the thread serialization process, while Table 5.2 refers to the symmetrical de-serialization process at the arrival host.

TABLE 5.1
Evaluated times for thread serialization (sec.)

	5 frames	**10 frames**	**25 frames**
OSR capturing	1.78E-5	1.89E-5	1.96E-5
State building	3.44E-5	3.75E-5	3.43E-5
Pure serialization	2.49E-3	7.32E-3	1.50E-2
Overall times	2.54E-3	7.38E-3	1.51E-2

Considering how these times are partitioned among the different phases of the thread serialization, we can see that the bulk of the time is wasted in the pure Java serialization of the captured state, while the frame extraction mechanism (i. e. the core of our entire facility, comprising OSR extraction and state building) has very short times instead. The same bottleneck due the Java serialization may be observed in the de-serialization of the thread. In the latter case, however, we have an additional overhead in the stack installation phase, since the system has often to create a new thread and compile (if not yet compiled) the methods for the injected frames. These performance bottlenecks can be further minimized, perhaps using externalization to speed up the serialization of the thread state [35]. Moreover, we had to modify the size of JikesRVM LOS (Large Object Space) to allow the instantiation of a bigger number of thread objects into the runtime image. Nevertheless, the developed prototype has some limitations that will be dealt with in the future: the first one is about the kind of supported compilers. JikesRVM basically provides two compilers, designed to achieve different levels of code optimization: a *baseline* and an *optimizing compiler* [5] (a third *quick compiler* is, at the time of writing, still in a prototypal phase). Our prototype can actually migrate baseline compiled methods JikesRVM, mainly because of an OSR mechanism limitation: it can actually capture method scope descriptors for those methods compiled by optimized compilers, but this requires maintaining additional structures to cope with parameters allocated into registers, inlined methods and other challenging optimization techniques [11]. Currently, JikesRVM designers allows OSR to occur only at yield points (i. e. thread pre-emption points in the code) and this implies that not all the optimized frames in the stack have their maps updated. Nonetheless, we are aware of a project by Krintz et al. [31] trying to present a more general-purpose version of OSR that is more amenable to optimizations than the current one. The improvement descending from this work will be exploited to perform a more complete thread state capturing, even in presence of code optimizations.

TABLE 5.2
Evaluated times for thread rebuilding (sec.)

	5 frames	**10 frames**	**25 frames**
Pure de-serialization	4.46E-3	5.33E-3	7.06E-3
State rebuilding	5.45E-4	5.27E-4	5.06E-4
Stack installation	1.53E-3	1.60E-3	1.71E-3
Overall times	6.54E-3	7.46E-3	9.28E-3

6. Conclusions and Future Work. This paper has introduced our *Mobile JikesRVM* framework to support Java thread strong mobility based on the IBM JikesRVM virtual machine, and has shown how its migration services can be effectively exploited to build mobile computing applications, such as the presented Aglets Mobile Agent Platform. Thanks to the support to thread serialization, agents will be simpler in terms of code, and, at the same time, the code will be easier to read. Our approach represents an extension of JikesRVM, which is pluggable at runtime in any OSR-enabled version of that JVM. It exploits, in fact, some interesting JikesRVM built-in facilities to avoid many of the drawbacks of past solutions. In particular, OSR facility allowed us to capture the execution state (i. e. method frames) in a very portable (i. e. bytecode-level) format. Thanks to the scheduling policy of the JikesRVM, which enables the support of thousands of Java threads, our approach keeps the thread management more lightweight, experimenting the possibility of having one thread for each agent, which is not possible in the current implementation of Aglets. Our JikesRVM-based migration library enriches the Aglets framework with strong mobility benefits. Additional features will be, of course, implemented

to extend our thread mobility framework in the future. Future work includes a comparison with other proposed thread migration systems, to improve our performance evaluation understanding and identify possible undetected bottlenecks. Finally we are currently working to port the implemented code also on PPC architectures (JikesRVM is available also for this architecture), allowing the migration of a thread among heterogeneous platforms as well.

Acknowledgments. Work supported by the European Community within the IST FET project "CASCADAS".

REFERENCES

[1] A. ACHARYA, M. RANGANATHAN, J. SALTZ, *Sumatra: A Language for Resource-aware Mobile Programs*, in 2nd International Workshop on Mobile Object Systems (MOS'96), Linz, Austria, 1996.
[2] THE AGLETS MOBILE AGENT PLATFORM WEBSITE, `http://aglets.sourceforge.net`
[3] B.ALPERN, C.R. ATTANASIO, D. GROVE AND OTHERS, *The Jalapeno virtual machine*, IBM System Journal, Vol. 39, N1, 2000.
[4] B. ALPERN, S. AUGART, S.M. BLACKBURN, M. BUTRICO, A. COCCHI, P. CHENG, J. DOLBY, S. FINK, D. GROVE, M. HIND AND OTHERS, *The Jikes Research Virtual Machine project: Building an open-source research community*, IBM Systems Journal, Vol. 44, No. 2, 2005.
[5] B. ALPERN, D. ATTANASIO, J. J. BARTON, A. COCCHI, S. F. HUMMEL, D. LIEBER, M. MERGEN, T. NGO, J. SHEPHERD, S. SMITH, *Implementing Jalapeño in Java*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), Denver, Colorado, November 1, 1999.
[6] M. ARNOLD, S. FINK, D. GROVE, M. HIND, P. F. SWEENEY, *Adaptive Optimization in the Jalapeño JVM*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 15-19, 2000.
[7] S. BOUCHENAK, D. HAGIMONT, S. KRAKOWIAK, N. DE PALMA AND F. BOYER, *Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence*, I.N.R.I.A., Research report n° 4662, December 2002.
[8] S. BOUCHENAK, D. HAGIMOT, *Pickling Threads State in the Java System*, Technology of Object-Oriented Languages and Systems Europe (TOOLS Europe'2000) Mont-Saint-Michel/Saint-Malo, France, Jun. 2000.
[9] G. BURKE, J.CHOI, S. FINK, D.GROVE, M. HIND, V. SARKAR, M.J. SERRANO, V.C. SREEDHAR, H. SRINIVASAN, The Jalapeno Dynamic Optimizing Compiler for Java, ACM Java Grande Conference, June 1999.
[10] C. CHAMBERS, *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, PhD thesis, Stanford University, Mar. 1992. Published as technical report STAN-CS-92-1420.
[11] S. FINK, AND F. QIAN, *Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement*, International Symposium on Code Generation and Optimization San Francisco, California, March 2003.
[12] S. FNFROCKEN, *Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs*, In K. Rothermel and F. Hohl (Ed.), Mobile Agents: Proceedings of the Second International Workshop (MA 98), Stuttgart, Germany. (pp. 26-37). Berlin, Germany: Springer-Verlag.
[13] A. FUGGETTA, G. P. PICCO, G. VIGNA, *Understanding Code Mobility*, IEEE Transactions on Software Engineering, Vol 24, 1998.
[14] T. ILLMANN, T. KRUEGER, F. KARGL, M. WEBER, *Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture*, Proceedings of the 5th International Conference on Mobile Agents, December 02 - 04, 2001, Atlanta, Georgia, USA.
[15] F. BELLIFEMINE, G. CAIRE, A. POGGI, G. RIMASSA, *JADE—A White Paper*, EXP in Search of Innovation, TILAB, vol. 3, 2003.
[16] N. R. JENNINGS, *An agent-based approach for building complex software systems*, Communications of the ACM, Vol. 44, No. 4, pp. 35-41 (2001).
[17] THE JIKESRVM PROJECT SITE, `http://jikesrvm.sourceforge.net`
[18] G. KICZALES, J. IRWIN, J. LAMPING, J. LOINGTIER, C. LOPES, C. MAEDA, AND A. MENDHEKAR, *Aspect Oriented Programming*, in Special Issues in Object-Oriented Programming, Max Muehlhaeuser (general editor) et al., 1996.
[19] D. B. LANGE, M. OSHIMA, G. KARJOTH, K. KOSAKA, *Aglets: Programming Mobile Agents in Java*, in the Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA), 1997.
[20] D. B. LANGE, Y. ARIDOR, *Agent Transfer Protocol (ATP)*, IBM=TRL, draft number 4, 19 March 1997.
[21] T. LINDHOLM, F. YELLIN, *The Java Virtual Machine Specification, second edition*, SUN Microsystem.
[22] M. LUCK, P. MCBURNEY, C. PREIST, *Agent Technology: Enabling Next Generation Computing A Roadmap for Agent Based Computing*, AgentLink, `http://www.agentlink.org/roadmap`
[23] Mobile JikesRVM is available at the projects website, `http://www.agentgroup.unimore.it/didattica/curriculum/raffaele`
[24] R. MONTANARI, E. LUPU AND C. STEFANELLI, *Policy-Based Dynamic Reconfiguration of Mobile-Code Applications*, in IEEE Computer, July 2004
[25] SCOTT OAKS AND HENRY WONG, *Java Threads, 2nd edition*, Oreilly, 1999
[26] D. PARKA AND S. RICE, *A Framework for Unified Resource Management in Java*, in the Proceedings of the International Conference on Principles and Practices of Programming In Java (PPPJ 2006), Mannheim, Germany, August 30 – September 1, 2006
[27] T. SAKAMOTO, T. SEKIGUCHI, A. YONEZAWA, *A bytecode transformation for Portable Thread Migration in Java*, 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Sep. 2000.
[28] T. SEKIGUCHI, A. YONEZAWA, H. MASUHARA, *A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation*, 3rd International Conference on Coordination Models and Languages, Amsterdam, The Netherlands, Apr. 1999.
[29] *The Java Object Serialization Specification*, Sun Microsystems, 1997.
[30] D. SISLAK, M. ROLLO, M. PECHOUCEK, *A-globe: Agent Platform with Inaccessibility and Mobility Support*, in Cooperative Information Agents VIII , n. 3191, Springer-Verlag Heidelberg, 2004.
[31] S. SOMAN, C. KRINTZ, Efficient, General-Purpose, On-Stack Replacement for Aggressive Program Specialization, University of California, Santa Barbara Technical Report 2004-24.
[32] T. SUEZAWA, *Persistent Execution State of a Java Virtual Machine*, ACM Java Grande 2000 Conference, San Francisco, CA, USA, Jun. 2000.
[33] N. SURI ET AL., *An Overview of the NOMADS Mobile Agent System*, Workshop On Mobile Object Systems in association with the 14th European Conference on Object-Oriented Programming (ECOOP 2000), Cannes, France.

[34] É. TANTER, M. VERNAILLEN AND J. PIQUER, *Towards Transparent Adaptation of Migration Policies*, in the 8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002), in conjunction with the 16th European Conference on Object-Oriented Programming (ECOOP 2002), June 2001, Malaga, Spain.

[35] SUN MICROSYSTEMS, *Improving Serialization Performance with Externalizable*, `http://java.sun.com/developer/TechTips/txtarchive/2000/Apr00_StuH.txt`

[36] E. TRUYEN, B. ROBBEN, B. VANHAUTE, T. CONINX, W. JOOSEN, P. VERBAETEN, *Portable support for Transparent Thread Migration in Java*, in 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Switzerland, Sep. 2000.

[37] X. WANG, *Translation from Strong Mobility to Weak Mobility for Java*, Master's thesis, The Ohio State University, 2001.

[38] W. ZHU, C. WANG, F. C. M. LAU, *JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*, in IEEE Fourth International Conference on Cluster Computing, Chicago, USA, September 2002.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 14, 2006

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 53–87. http://www.scpe.org
ISSN 1895-1767

A FAULT-TOLERANT DIRECTORY SERVICE FOR MOBILE AGENTS BASED ON FORWARDING POINTERS

LUC MOREAU*

Key words. mobile agents, distributed directory service, fault tolerance

Abstract. A reliable communication layer is an essential component of a mobile agent system. We present a new fault-tolerant directory service for mobile agents, which can be used to route messages reliably to them, even in the presence of failures of intermediary nodes between senders and receivers. The directory service, based on a technique of forwarding pointers, introduces some redundancy in order to ensure resilience to stopping failures of nodes containing forwarding pointers; in addition, it avoids cyclic routing of messages, and it supports a technique to collapse chains of pointers that allows direct communication between agents. We have formalised the algorithm and derived a *fully mechanical proof* of its correctness using the proof assistant Coq; we report on our experience of designing the algorithm and deriving its proof of correctness. The complete source code of the proof is made available from the WWW.

1. Introduction. While mobile agents have been touted as a major programming paradigm for structuring distributed applications [3, 5], several important issues remain to be addressed before mobile agents can become a mainstream technology for such applications: among them, a *communication system* and a *security infrastructure* are needed respectively for facilitating communications between mobile agents and for protecting agents and their hosts.

In this article, we focus solely on the problem of communications, whereas security issues are the focus of other publications, such as [20, 21]. Various authors have previously investigated a communication layer for mobile agents based on *forwarding pointers* [17, 10]. In such an approach, when mobile agents migrate, they leave forwarding pointers that are used to route messages. This simple approach raises some concerns: first, one needs to avoid cyclic routing when agents migrate to previously visited sites; second, chains of pointers can become arbitrarily long and increase the cost of communication. The first problem can be addressed by using timestamps [10], whereas the second can be solved by techniques such as lazy updates and piggy-backing of information on messages [14]. For structuring and clarity purposes, a communication layer for mobile agents is usually defined in terms of a message router and a directory service; the latter tracks mobile agents' locations, whereas the former forwards messages using the information provided by the latter.

Directory services based on forwarding pointers are currently ***not*** *tolerant* to failures [17, 10]: the failure of a node containing a forwarding pointer may prevent finding agents' positions. The purpose of this article is to present a directory service, fully distributed and resilient to failures exhibited by intermediary nodes that possibly contain forwarding pointers. This algorithm may be used to specify fault-tolerant message routers.

We consider stopping failures according to which processes are allowed to stop during the course or their execution [7]. The essence of our fault-tolerant distributed directory service is to introduce *redundancy* of forwarding pointers, typically by making N copies of agents' location information. This type of redundancy ensures the resilience of the algorithm to a maximum of $N-1$ failures of intermediary nodes. We show that the complexity of the algorithm remains linear in N. Our specific contributions are:

1. A *new directory service* based on forwarding pointers that is fault-tolerant, preventing cyclic routing, and not involving any static location;
2. A *full mechanical proof* of its correctness, using the proof assistant Coq [1]; the complete source code of the proof (involving some 25000 tactic invocations) may be downloaded from the following URL [9].

This article is an extended version of a paper originally published at the AIMS track (Agents, interactions, mobility, and systems) at the ACM Symposium on Applied Computing 2002 [11]. It extends the original paper with a series of graphical animations that illustrate the behaviour of the algorithm.

We begin this paper by a survey of background work (Section 2) and follow by a summary of a routing algorithm based on forwarding pointers (Section 3). We present our new directory service (Section 4) and its formalisation as an abstract machine (Section 5). The purpose of Section 6 is to summarise the correctness properties of the algorithm: its safety states that the distributed directory service correctly and uniquely identifies agents' positions, whereas the liveness property shows that the algorithm reaches a stable state after a finite number of transitions, once agents stop migrating. Then, in Section 7, we report on our experience of designing the algorithm and deriving its proof of correctness, and we suggest possible variants or extensions, before discussing further related work (Section 8) and concluding the paper.

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

2. Background. The topic of mobile agent tracking and communication has been researched extensively by the mobile agent community. Very early on, location-aware communications were proposed: they consist of sending messages to locations where agents are believed to be, but typically result in failure if the receiver agent has migrated [16, 22].

For a number of applications, such a kind of communication layer is not satisfactory when the expectation is to deliver messages *reliably* to a recipient, wherever its location and whatever the route adopted; for instance, it would not be acceptable to lose a message in a negotiation between two mobile agents, simply because one of them changed location. To combat this problem, location-transparent communication services were introduced as a means to route and deliver messages automatically to mobile agents, independently of their migration. Such services have been shown to be implementable on top of a location-aware communication layer [22].

In the category of location-transparent communication layers, there are essentially two distinct approaches, respectively based on *home agents* and *forwarding pointers*. In systems based on home agents, such as Aglets [5], each mobile agent is associated with a non-mobile home agent. In order to communicate with a mobile agent, a message has to be sent to its associated home agent, which forwards it to the mobile one; when a mobile agent migrates, it informs its home agent of its new position. Alternatively, in mobile agent systems such as Voyager [17], agents that migrate leave trails of forwarding pointers, which are used to route messages.

In pervasive computing environments, for example, the mechanism of a home agent may defeat the purpose of using mobile agents by re-introducing centralisation: the home agent approach puts a burden on the infrastructure, which may hamper its scalability, in particular, in massively distributed systems. A typical illustration of the difficulty consists of two mobile agents with respective home bases in the US and Europe having to communicate with each other, while located at a host in Australia. In such a scenario, routing via home agents is not desirable, and may not be possible when the Australian host is temporarily disconnected from the network. If we introduce a mechanism by which home agents change location dynamically according to the task at hand, we face the problem of how to communicate reliably with a home agent, which is itself mobile. Alternatively, we could only use the home agent to bootstrap communication, and then shortcut the route, but this approach becomes unreliable once agents migrate. Finally, the home agent is also as a *single point of failure*: when it exhibits a failure, it becomes impossible to track the mobile agent or to route messages to it.

A naive forwarding pointer implementation causes communications to become more expensive as agents migrate, because chains of pointers increase. Chains of pointers need to be collapsed promptly so that mobile agents become independent of the hosts they previously visited. Once the chain has collapsed direct communications become possible and avoid the awkward scenario discussed above. As far as tolerance to failures is concerned, the failure of an intermediary node with a forwarding pointer prevents upstream nodes from forwarding messages. Hence, collapsing chains of pointers is crucial to reduce the system's exposure to failures.

Coordination models offer a more asynchronous form of communication, typically involving a tuple space [4]. As tuple spaces are non-mobile, they may suffer from the same problem as the home agent. To improve locality, tuple spaces may be distributed, and coordinated by replication protocols, but maintaining consistency in the presence of updates is a non-trivial problem. A further inconvenience of the coordination approach is that it requires coordinated processes to poll tuple spaces, which may be an inefficient approach in terms of both communication and computation. To combat this problem, tuple spaces generally provide a mechanism by which registered clients can be notified of the arrival of a new tuple: when clients are mobile, we are back to the problem of delivering notifications reliably to a potentially mobile recipient. Alternatively, if the tuple space itself is mobile [18], the problem is then to deliver messages to the tuple space.

This discussion shows that reliable delivery of messages to mobile agents without using static locations to route messages is essential, even if peer-to-peer communications are not adopted as the high-level interaction paradigm between agents. Previous work has focused on formalisation [10] and implementation [17] of forwarding pointers, but such solutions were not fault-tolerant. We summarise such an approach in Section 3 before extending it with support for failures in Section 5.

3. Original Algorithm. In this section, we summarise the principles of a communication layer based on forwarding pointers [10] without any fault-tolerance. The algorithm comprises two components: a distributed directory service and a message router, which we describe below; we then describe why the algorithm is not tolerant to failures.

3.1. Distributed Directory Service. Each mobile agent is associated with a timestamp that is increased every time the agent migrates. When an agent has decided to migrate to a new location, it requests the communication layer to transport it to its new destination. When the agent arrives at a new location, an acknowledgement message containing both its new position and its newly-incremented timestamp is sent to its previous location. As a result, for each site, one of the following three cases is valid for each agent A: *(i)* the agent A is local, *(ii)* the agent A is in transit but has not acknowledged its new position yet, or *(iii)* the agent A is known to have been at a remote location with a given timestamp.

Figures 3.1 to 3.8 present an animation of the algorithm by showing various agent locations and associated site states. (*Note: such animation is best viewed in colour directly in a pdf viewer.*) Each site contains a local state composed of four components, which we intuitively introduce now.

1. *mob*: the lastest known mobility counter, i. e. timestamp, of the agent;
2. *loc*: the lastest known location of the agent;
3. *pres*: a boolean flag indicating if the agent is present locally or not;
4. *ack*: whether an acknowledgement message has to be sent or not.

As an illustration, in Figure 3.1, at site s_2, agent is known to have timestamp $t+1$ and location s_2, and to be present locally. No acknowledgement needs to be sent by s_2.

The mobile agent located at site s_2 decides to migrate to a new location (Figure 3.2). As it makes the request to the transport layer to be transported and leaves s_2 (Figure 3.3), the agent state is packaged up with its previous location's name s_2 and the timestamp it would have at its next location $t+2$. The latest known timestamp and location remain unchanged at s_2 since the agent's presence elsewhere has not been confirmed yet. However, the agent is now known to be absent from s_2.

As the agent arrives at s_3 (Figure 3.4), s_3 acquires the knowledge that the agent is present locally (hence, reflected in the states *mob*, *loc* and *pres*). Furthermore, the state *ack* is changed at s_3 to indicate that the new agent's position has to be communicated to its previous location s_2.

The sending of the acknowledgement message by s_3 (Figure 3.5) clears the *ack* state at s_3 and results in the latest known location of the agent to be communicated to s_2 (as reflected by the change of *mob* and *loc* at s_2 in Figure 3.6). As a result, the processing of the acknowledgement message by s_2 results in a *forwarding pointer* being set up from s_2 to s_3.

Timestamps are essential to avoid race conditions between acknowledgement messages: by using timestamps, a site can decide which position information is the most recent, and therefore can avoid creating cycles in the graph of forwarding pointers (see [10] for details).

In order to avoid an increasing cost of communication when the agent migrates, it is useful to reduce the length of forwarding pointers. To this end, we have established [10] that information messages, containing a site's belief about the agent's position and its timestamp, can be communicated by any site to any other site. The processing of such inform messages, like acknowledgement messages, updates the receipient's local state if the knowledge received is more recent than the local one (as per indicated by the timestamp). Figure 3.7 illustrates the sending of such an information message by s_2, sharing its knowledge of the agent's position with s_1. On receipt of the information message (Figure 3.8), s_1 updates its internal tables. By propagating such agent's positions, one can reduces the length of chains of pointers. Different strategies such as eager or lazy propagation are discussed in [14].

3.2. Message Router. Sites rely on the information about agents' positions in order to route messages. For any incoming message aimed at an agent A, the message will be delivered to A if A is known to be local. If A is in transit, the message will be enqueued, until A's location becomes known; otherwise, the message is forwarded to A's latest known location.

3.3. Absence of Fault Tolerance. There is no redundancy in the information concerning an agent's location. Indeed, sites only remember the most recent location of an agent, and only the previous agent's location is informed of the new agent's position after a migration. As a result, a site (transitively) pointing at a site exhibiting a failure has lost its route to the agent.

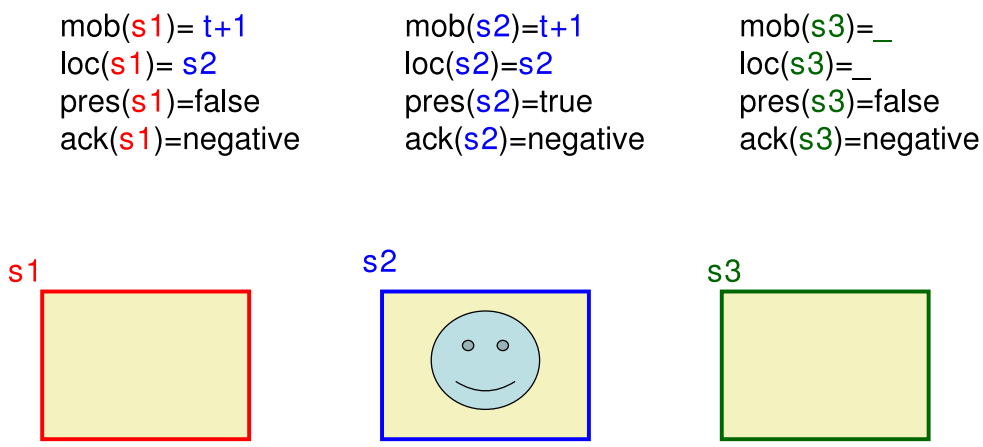


FIG. 3.1. *Original Algorithm (1/8)*

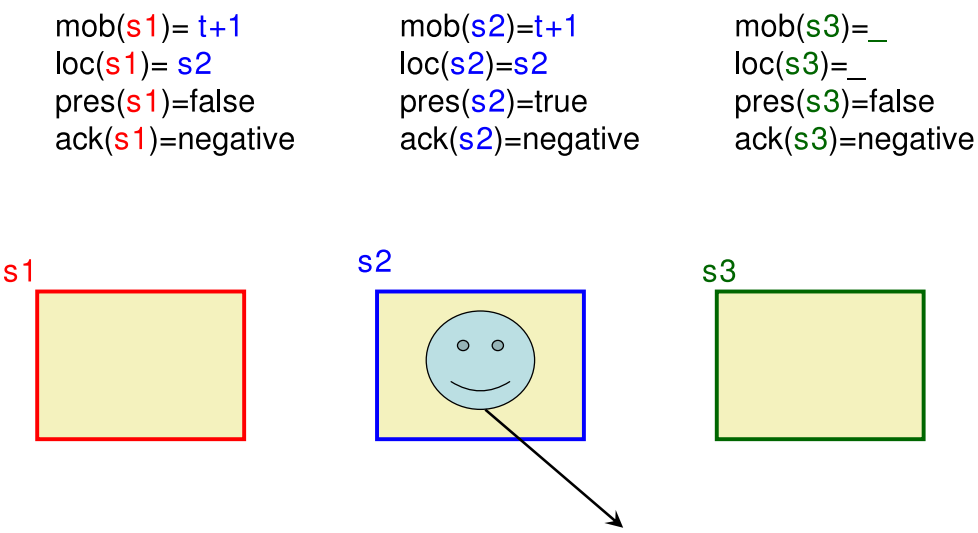


FIG. 3.2. *Original Algorithm (2/8)*

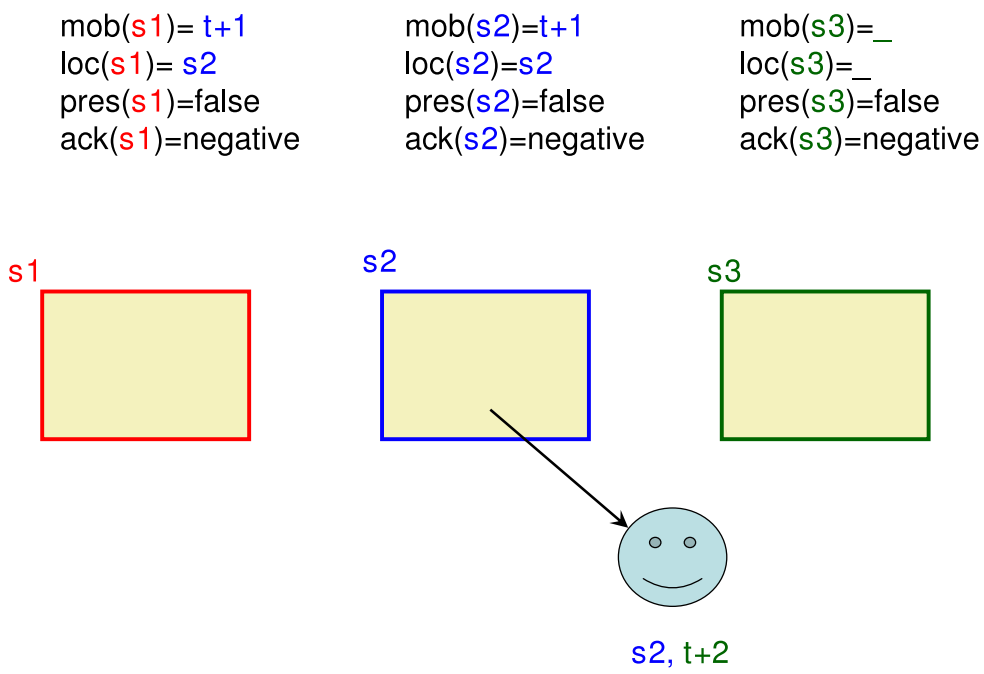


FIG. 3.3. *Original Algorithm (3/8)*

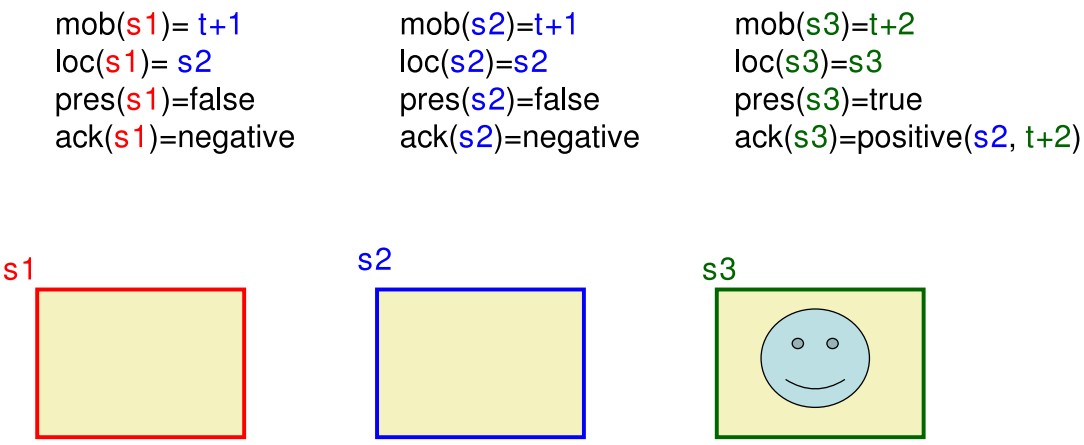


FIG. 3.4. *Original Algorithm (4/8)*

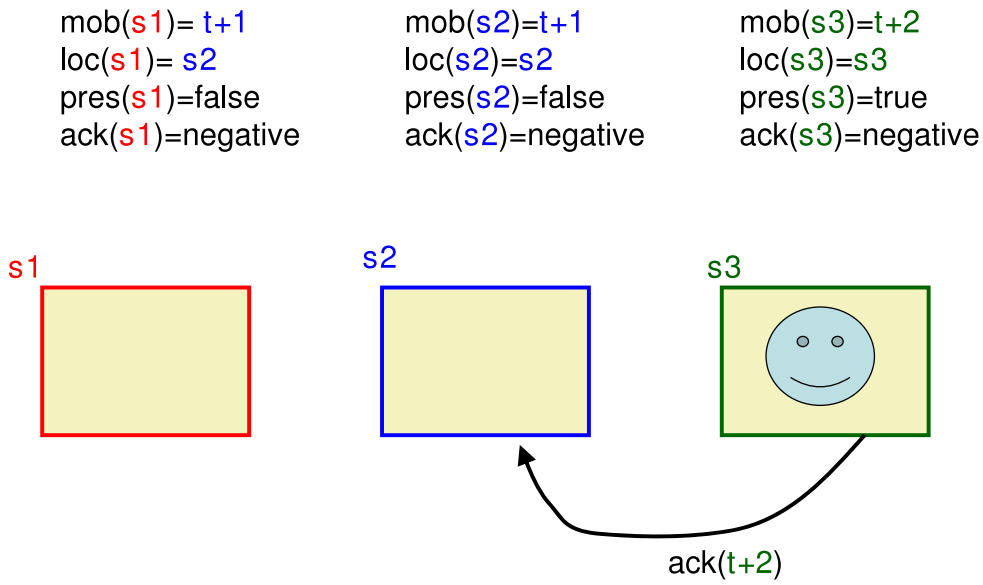


FIG. 3.5. *Original Algorithm (5/8)*

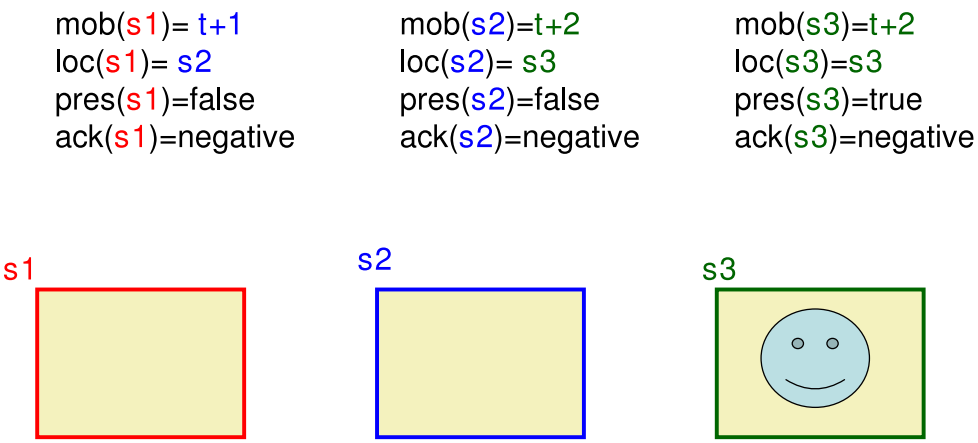


FIG. 3.6. *Original Algorithm (6/8)*

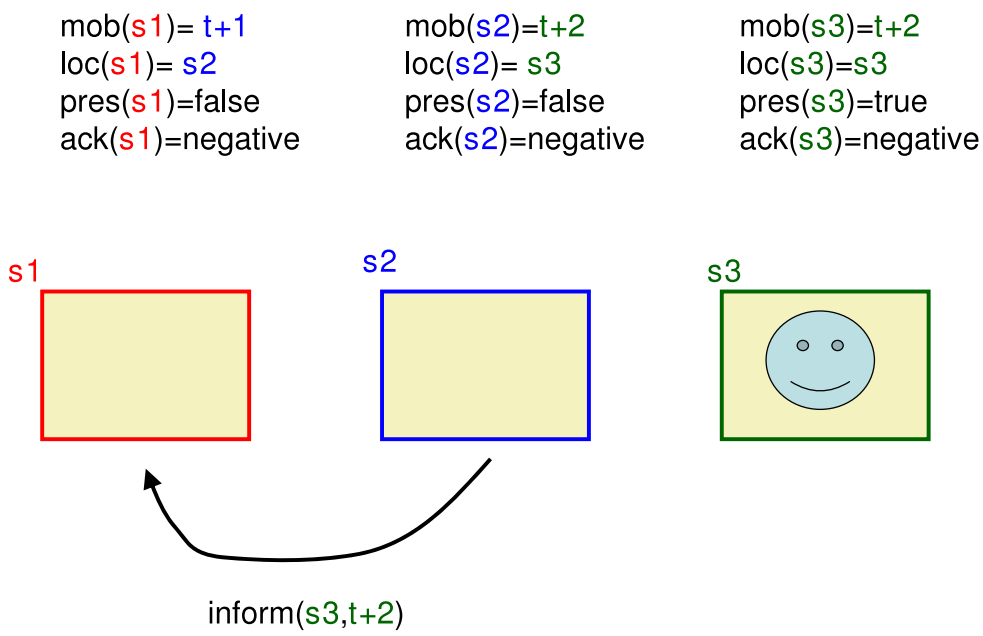


FIG. 3.7. *Original Algorithm (7/8)*

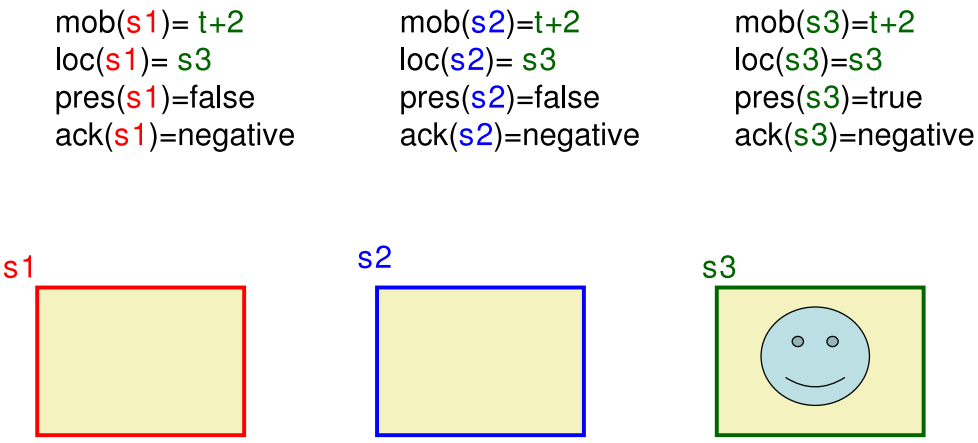


FIG. 3.8. *Original Algorithm (8/8)*

4. Fault-Tolerant Algorithm. The intuition of our solution to the problem of failures is to introduce some *redundancy* in the information that sites maintain about agents' positions. To this end, we bring three novel aspects to the original algorithm. First, agents remember N previous different sites that they have visited. Second, once an agent arrives at a new location, the agent's N previous locations are informed of its new position. Third, sites remember up to N different positions for an agent and their associated timestamps. With these changes in place, we shall establish that the algorithm is able to determine the agent's position correctly, provided that the number of stopping failures remains smaller or equal to $N-1$.

As in the original algorithm, each mobile agent is associated with a timestamp that is increased every time the agent migrates. When an agent has decided to migrate to a new location, it requests the communication layer to transport it to its new destination. When the agent arrives at a new location, an acknowledgement message containing both its new position and its newly-incremented timestamp is sent to N previous location.

We illustrate the modified algorithm by the animation in Figures 4.1 to 4.9. Each site contains a local state composed of four components, which we intuitively explain as follows:

1. *mob*: the lastest known mobility counter, i. e. timestamp, of the agent;
2. *loc*: up to N different most recently known locations visited by the agent after the current site;
3. *pres*: when the agent is present, it indicates up to N most recently visited different locations before the current site;
4. *ack*: indicating whether acknowledgement messages have to be sent or not.

A complete formalisation will follow in Section 5. In Figure 4.1, at site s_3, the agent's timestamp is $t+2$, the agent is local (and hence there is no other site visited after s_3), and the agent was known to be be in s_1 and s_2, previously at timestamps t and $t+1$, respectively.

The mobile agent located at site s_3 decides to migrate to a new location (Figure 4.2). As the agent makes the request to the transport layer to be transported and leaves s_3 (Figure 4.3), the agent state is packaged up with its previous locations s_2, s_1 and associated timestamps it had at these. The latest known timestamp and location remain unchanged at s_3 since the agent's presence elsewhere has not been confirmed yet. However, the agent is now known to be absent from s_3.

As the agent arrives at s_4 (Figure 4.5), s_4 acquires the knowledge that the agent is present locally (hence, reflected in its states *mob*, *loc* and *pres*). Furthermore, the state *ack* is changed to indicate that the new agent's position has to be communicated to its previous locations s_1 to s_3.

The sending of acknowledgement messages by s_4 (Figure 4.6) updates the *ack* state at s_4 and results in the latest known locations of the agent to be communicated to s_1, s_2, s_3 (as reflected by the change of *loc* at these sites in Figures 4.7 to 4.9).

As in the original algorithm, we allow arbitrary sites to communicate their knowledge about the agent's position and timestamp. Instead of introducing a specific message for this purpose, we overload the meaning of the acknolwedgement message (see Figures 4.7 to 4.9). This allows chains of forwarding pointers to be shortened, and hence reduce exposure to failures.

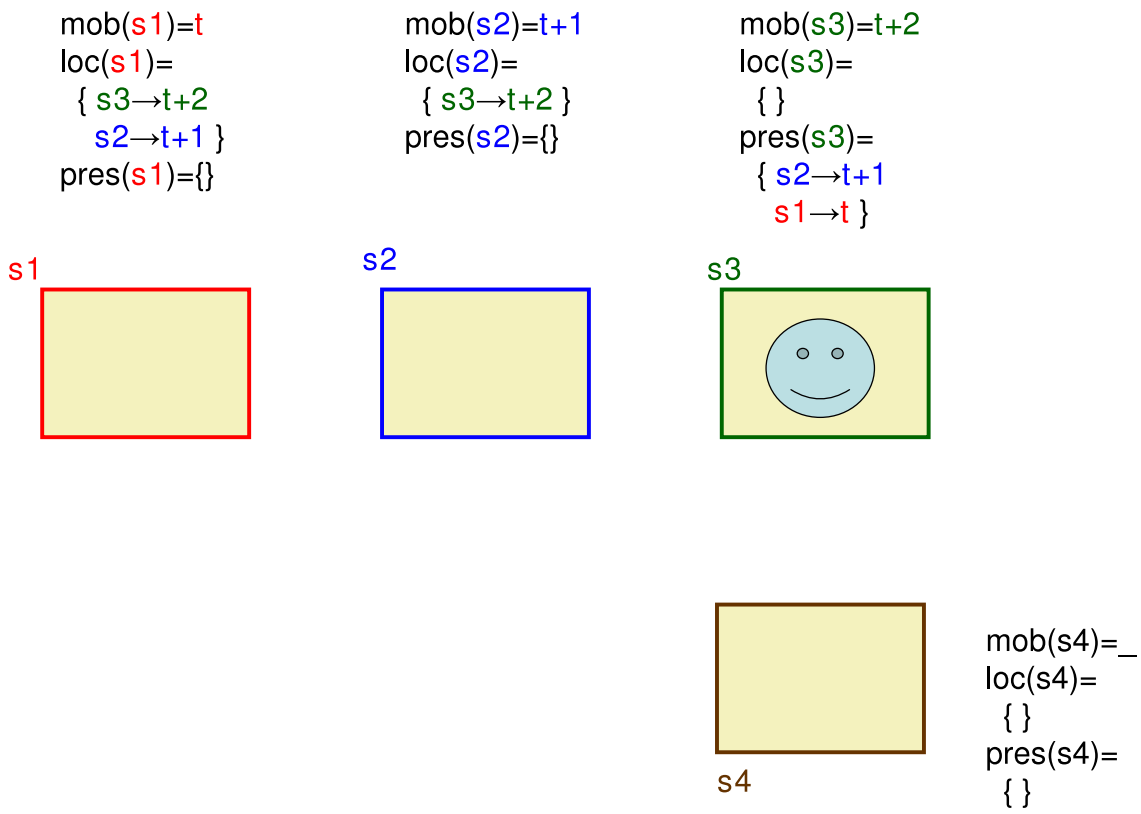


FIG. 4.1. *Agent Migration with 3-Redundancy (1/9)*

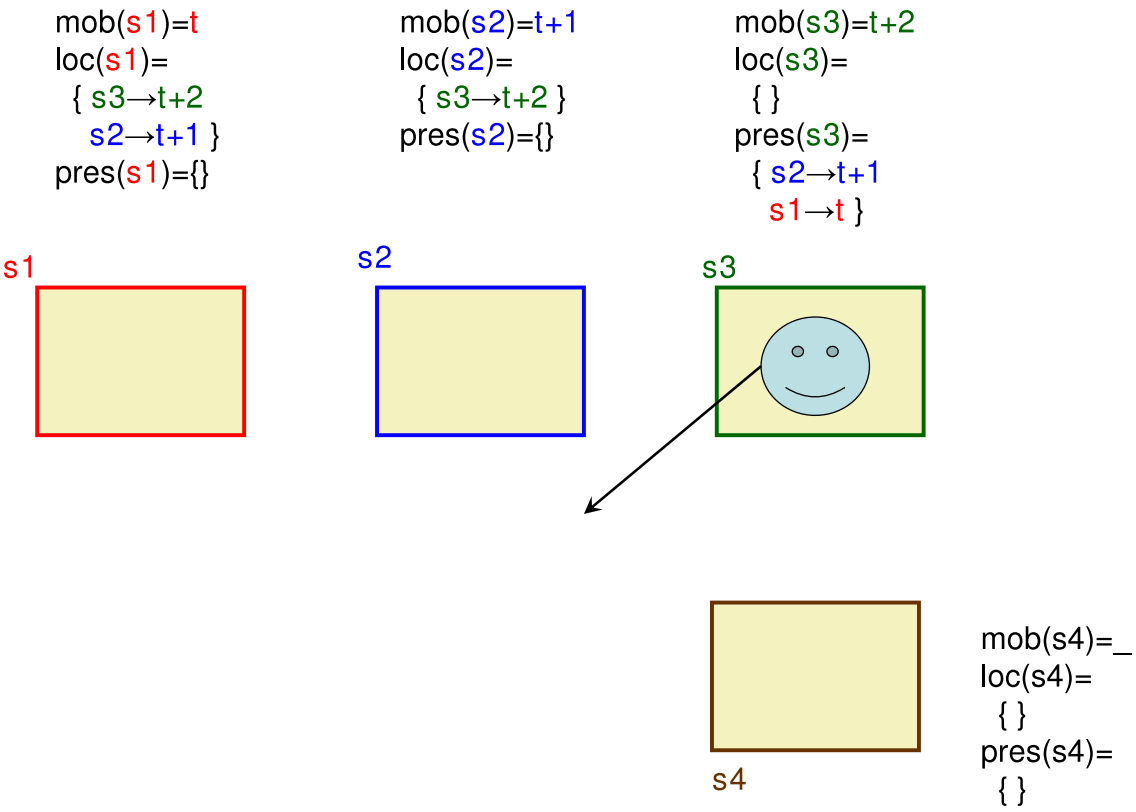


FIG. 4.2. *Agent Migration with 3-Redundancy (2/9)*

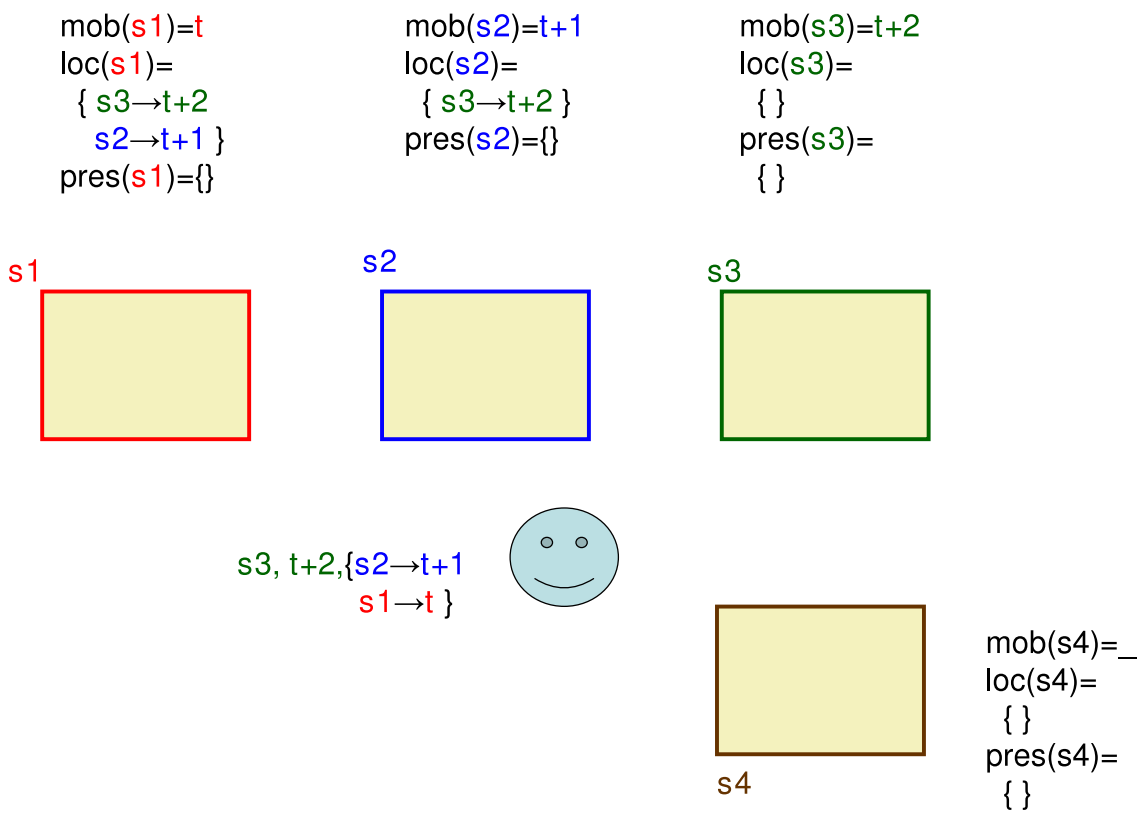


FIG. 4.3. *Agent Migration with 3-Redundancy (3/9)*

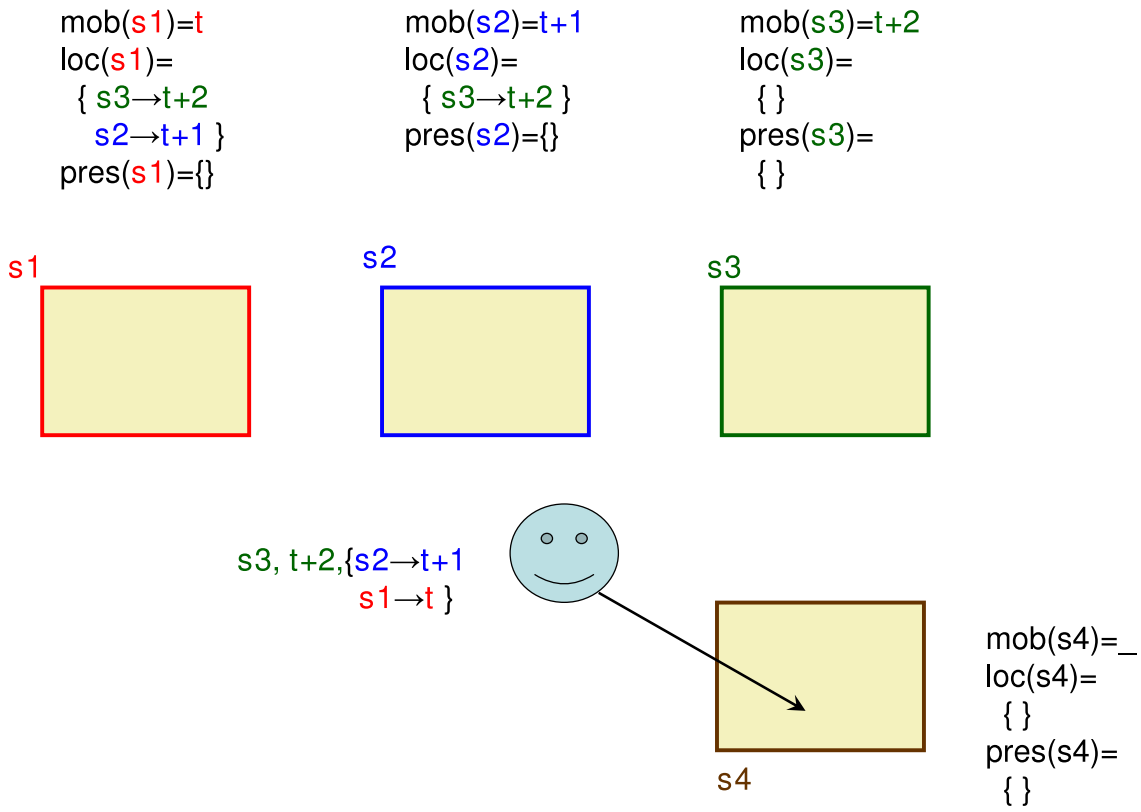


FIG. 4.4. *Agent Migration with 3-Redundancy (4/9)*

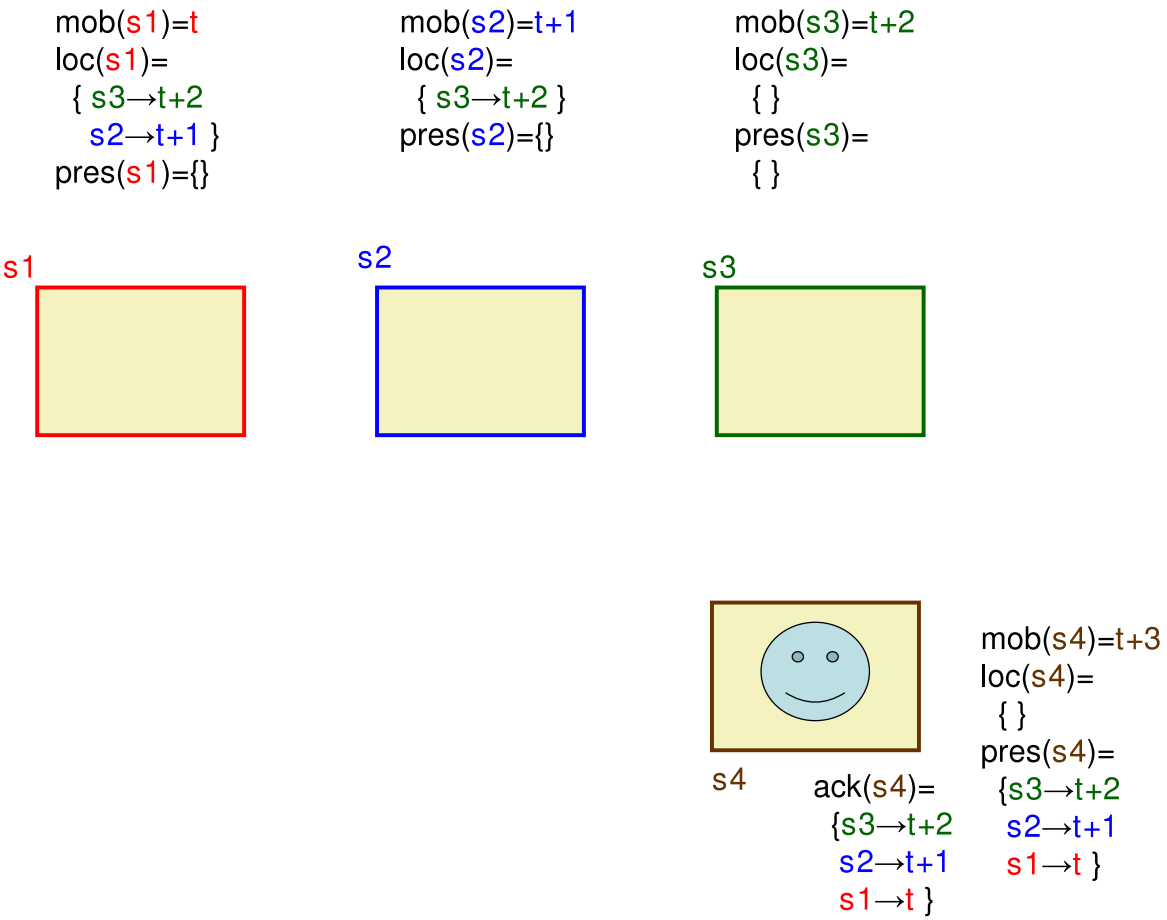


FIG. 4.5. *Agent Migration with 3-Redundancy (5/9)*

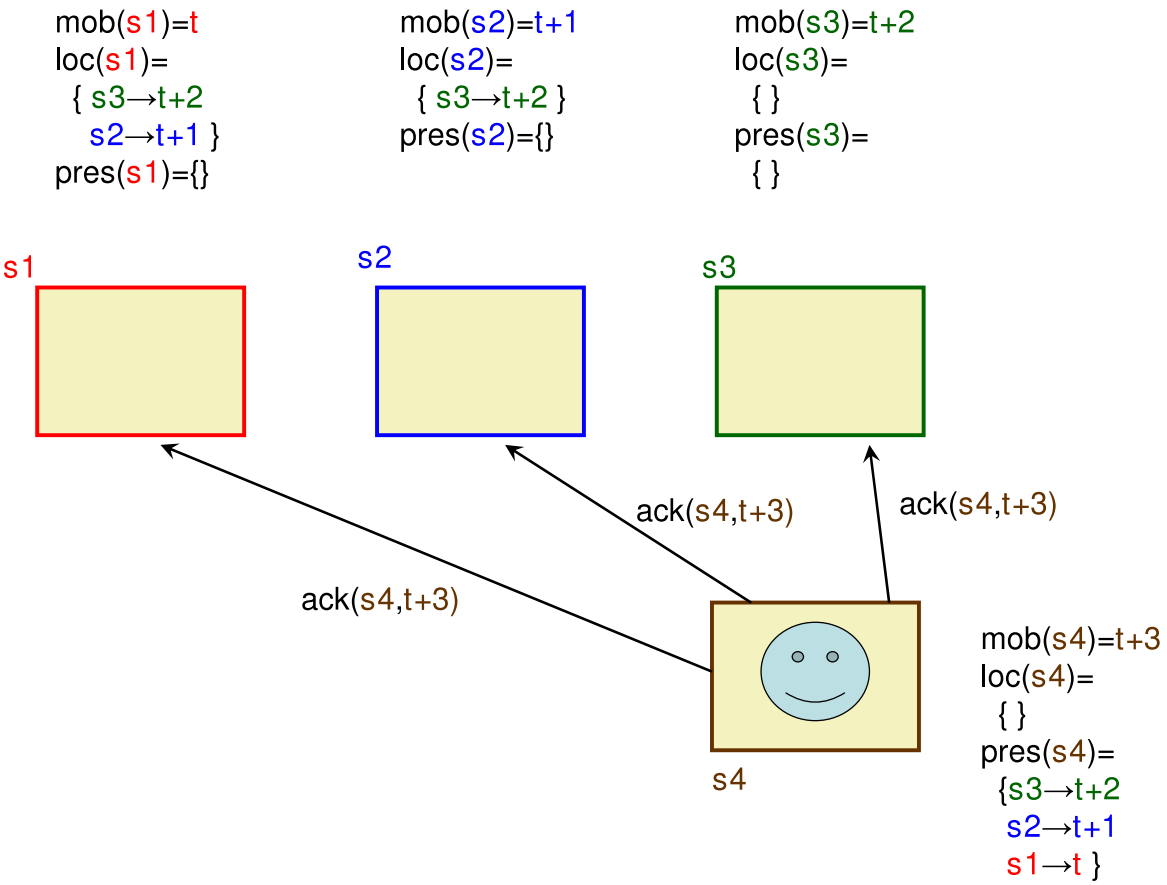


FIG. 4.6. *Agent Migration with 3-Redundancy (6/9)*

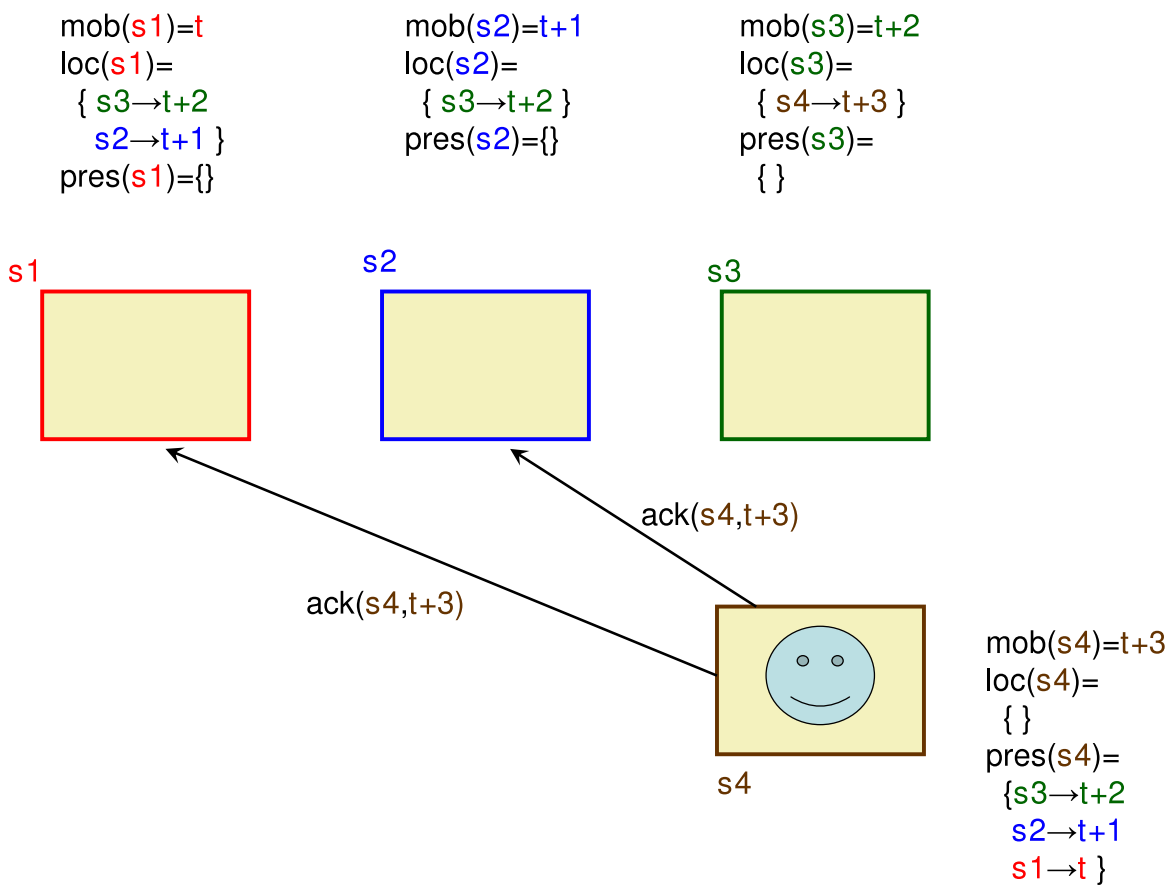


FIG. 4.7. *Agent Migration with 3-Redundancy (7/9)*

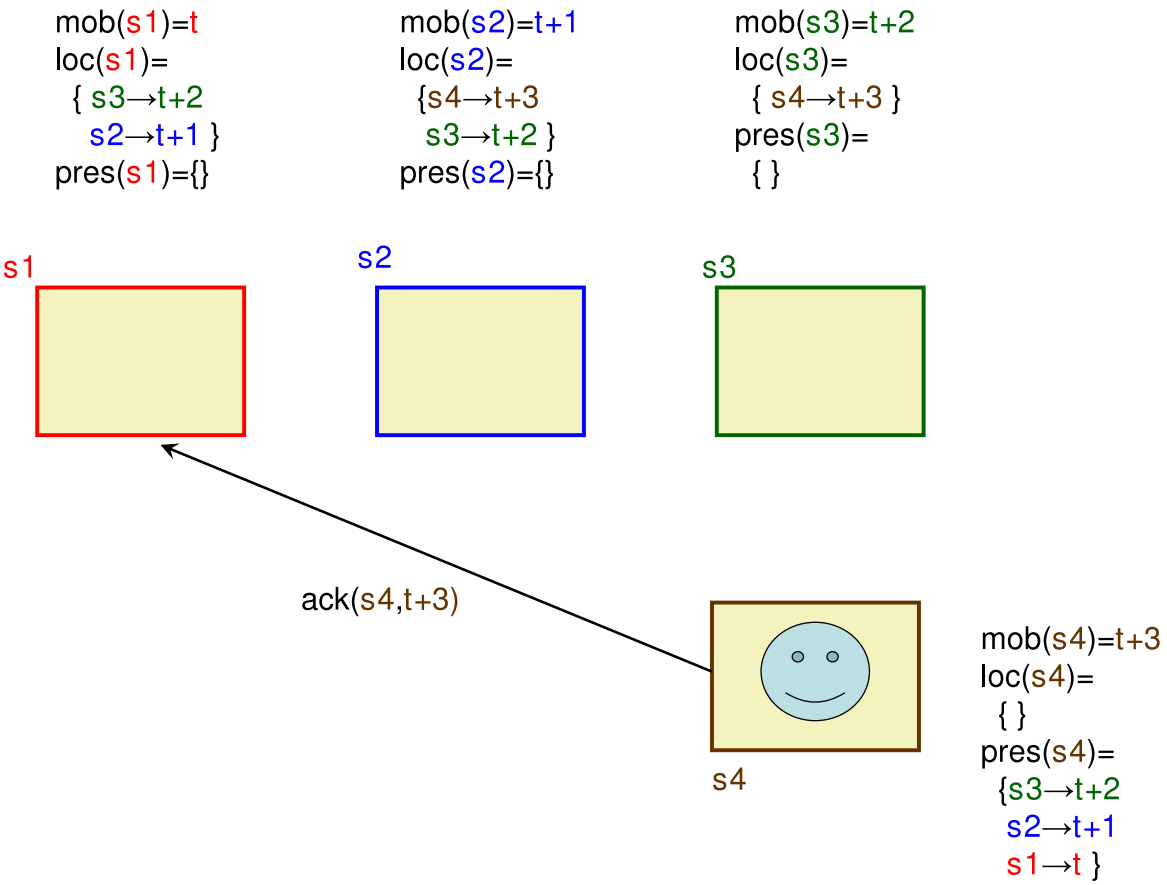


FIG. 4.8. *Agent Migration with 3-Redundancy (8/9)*

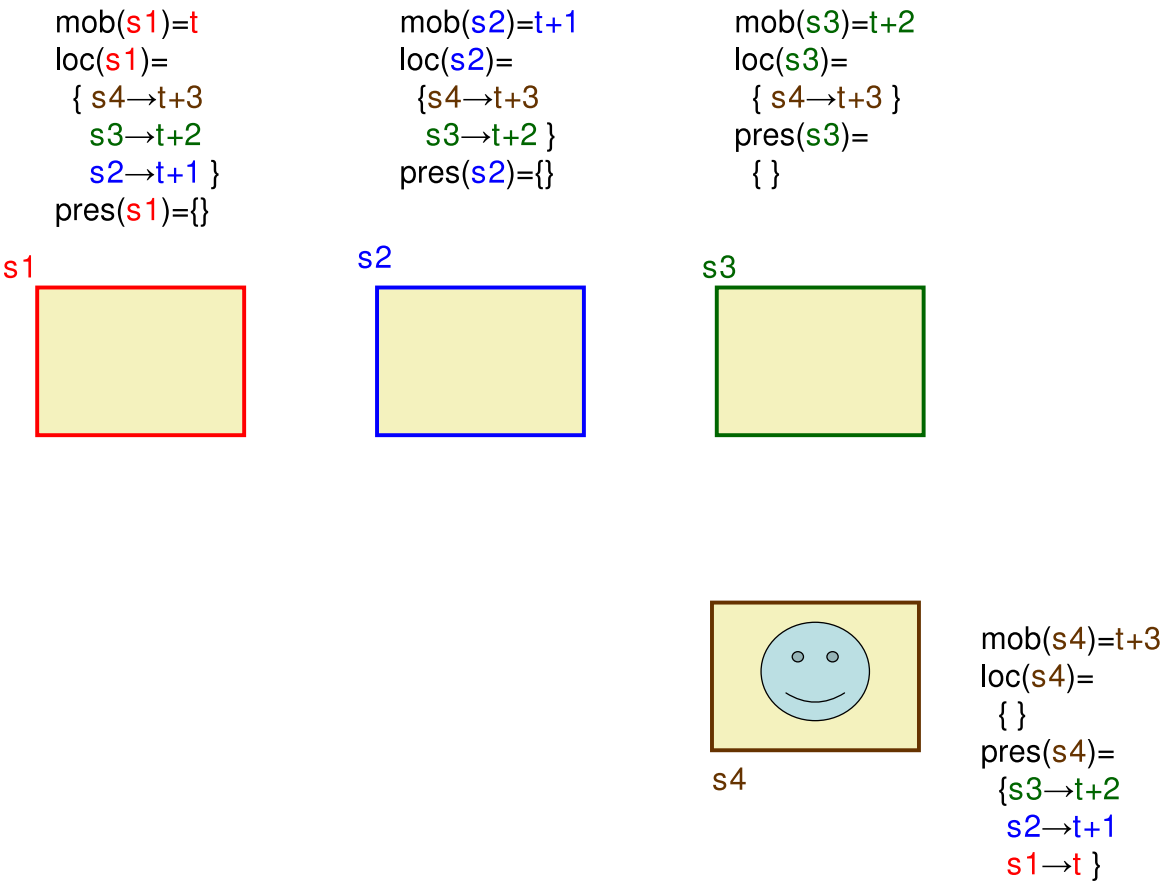


FIG. 4.9. *Agent Migration with 3-Redundancy (9/9)*

Consequently, the processing of each acknowledgement message (Figures 4.6 to 4.9), resulted in a *forwarding pointer* being set up by each recipient of these messages to the agent's latest position. Given that N acknowledgement messages are sent, N such forwarding pointers are being established, hereby introducing a redundancy in the number of pointers to the agent.

The benefit of such a redundancy is itself better illustrated by an animation, with Figures 4.10 to 4.15. Figure 4.10 is simply a replica of Figure 4.9, whereas Figure 4.11 only displays information about the latest known positions of the agent.

Such latest known agent positions are in fact forwarding pointers to locations where the agent is believed to be. Hence, in Figure 4.12, we represent such forwarding pointers graphically. We can see that from site s_1, there are many different routes that lead to s_4.

The following figures consider the presence of failures in the network. In Figure 4.13, s_2 stopped due to a failure, whereas in Figure 4.14, it is s_3 that stopped due to a failure. Among all the possible routes identified in Figure 4.12, we see in Figures 4.13 and 4.14 that there exists routes that do not involve failed nodes.

Figure 4.15 even considers two failures at s_2 and s_3, and the remaining route between s_1 and s_4 that does not involve failed nodes. Figures 4.1 to 4.9 illustrated 3-redundancy since up to three different agent locations are remembered by sites. Figures 4.15 shows that even in the present of $3-1$ failures, there is still a route to find the agent.

Remark Our purpose is to design an algorithm that is resilient to failures of *intermediary* nodes, and therefore we do not consider failures that may occur at the sender node and at the recipient node, i. e. "the first site that sends a message" and "the site where the agent is located". In other words, we are not concerned with reliability of agents themselves. Instead, systems replicating agents and using failure detectors such as [8] may be used for that purpose; they are complementary to our approach. Furthemore, in this paper, we also assume that communications are reliable.

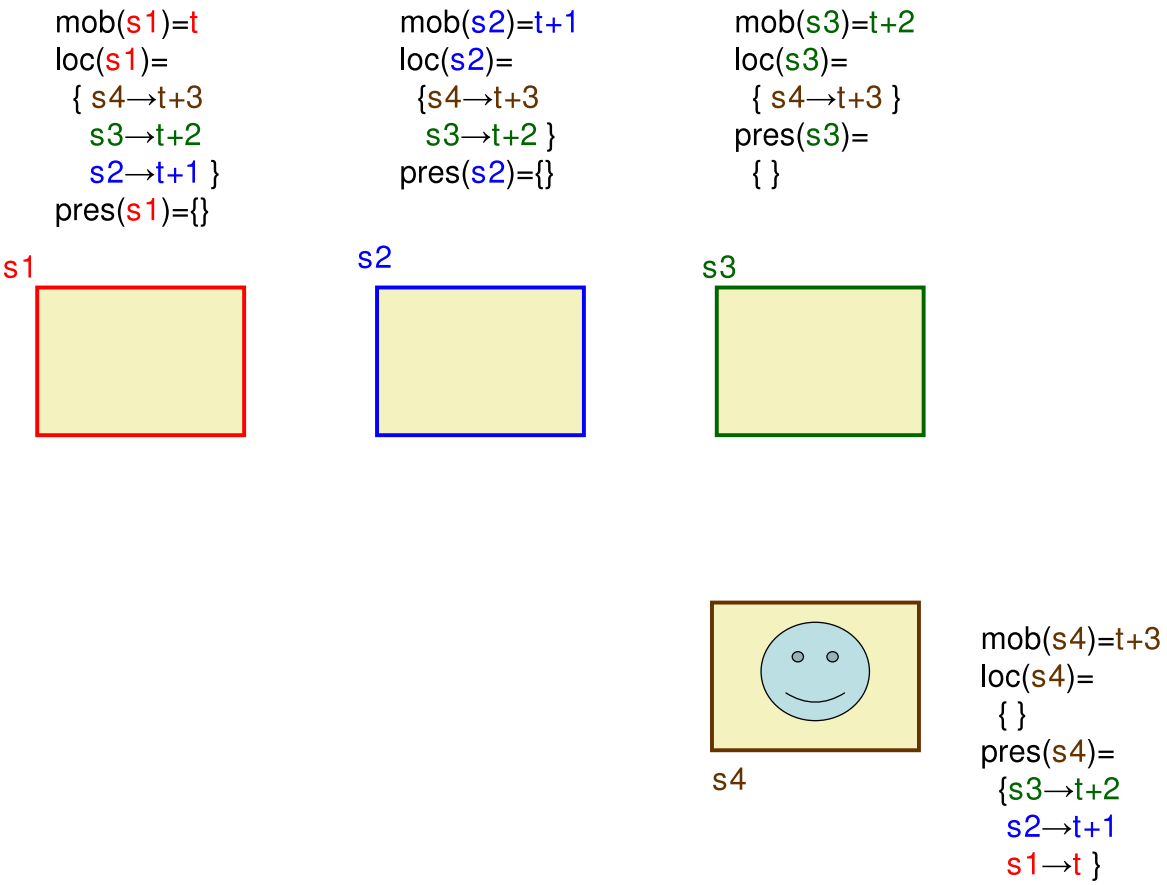


FIG. 4.10. *Alternate Paths in the Presence of Failures (1/6)*

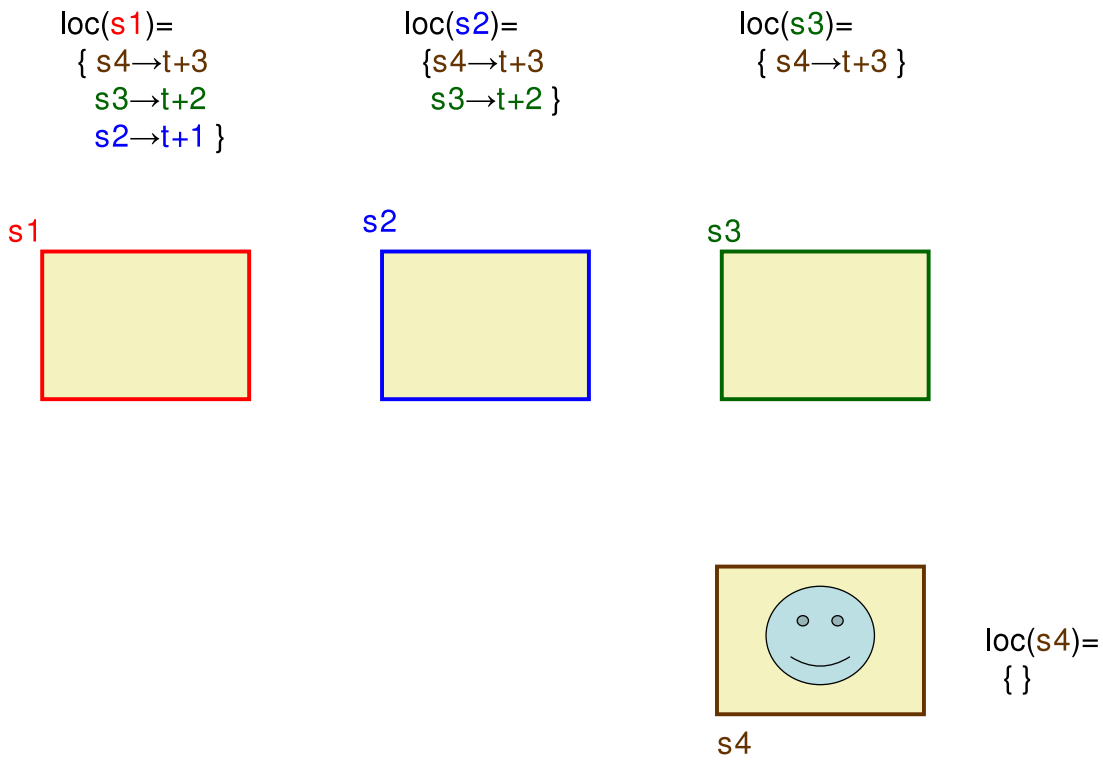


FIG. 4.11. *Alternate Paths in the Presence of Failures (2/6)*

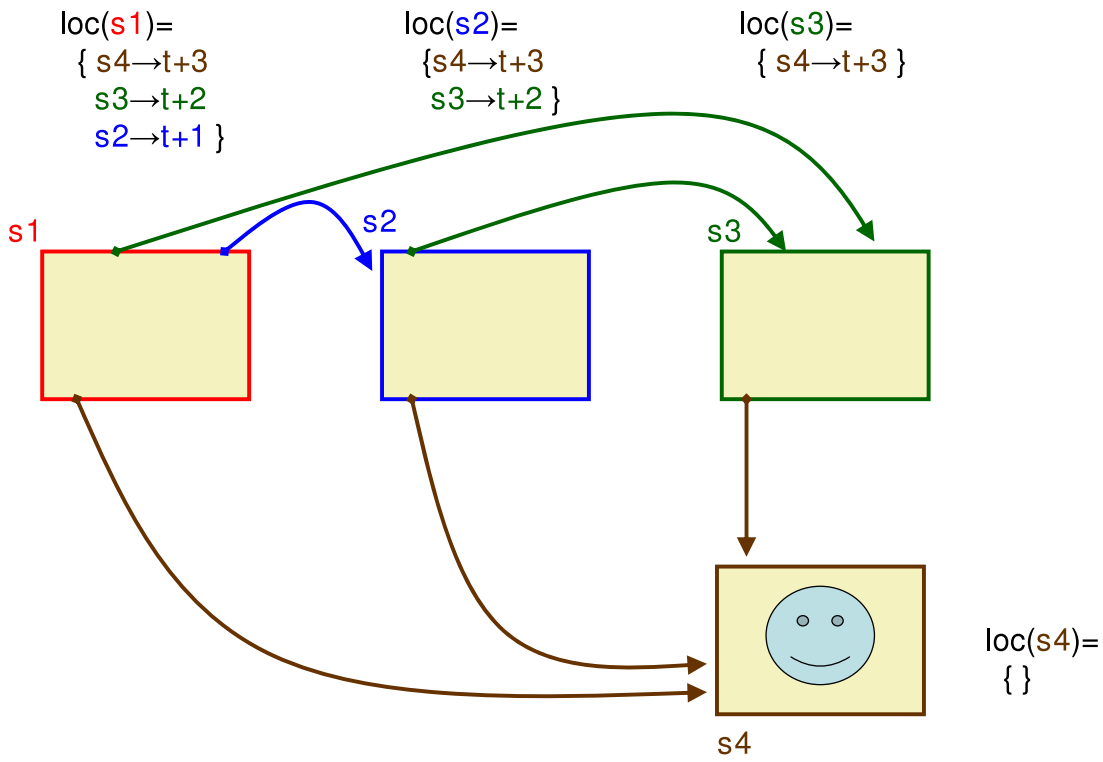


FIG. 4.12. *Alternate Paths in the Presence of Failures (3/6)*

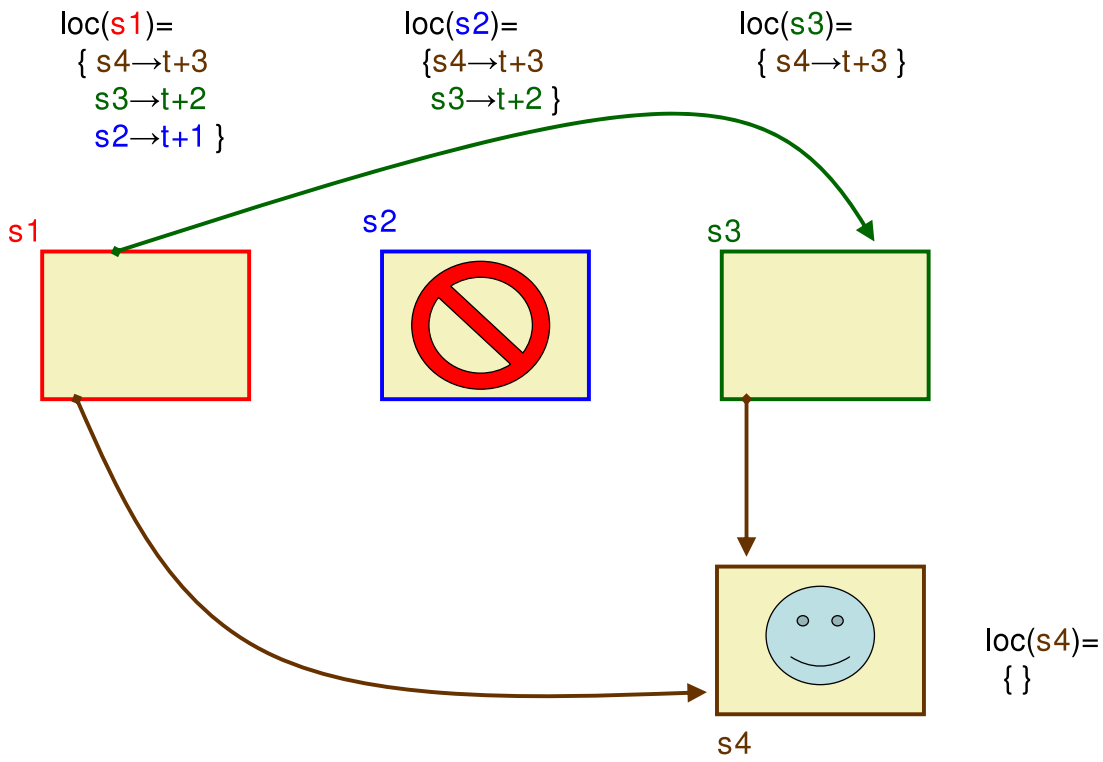


FIG. 4.13. *Alternate Paths in the Presence of Failures (4/6)*

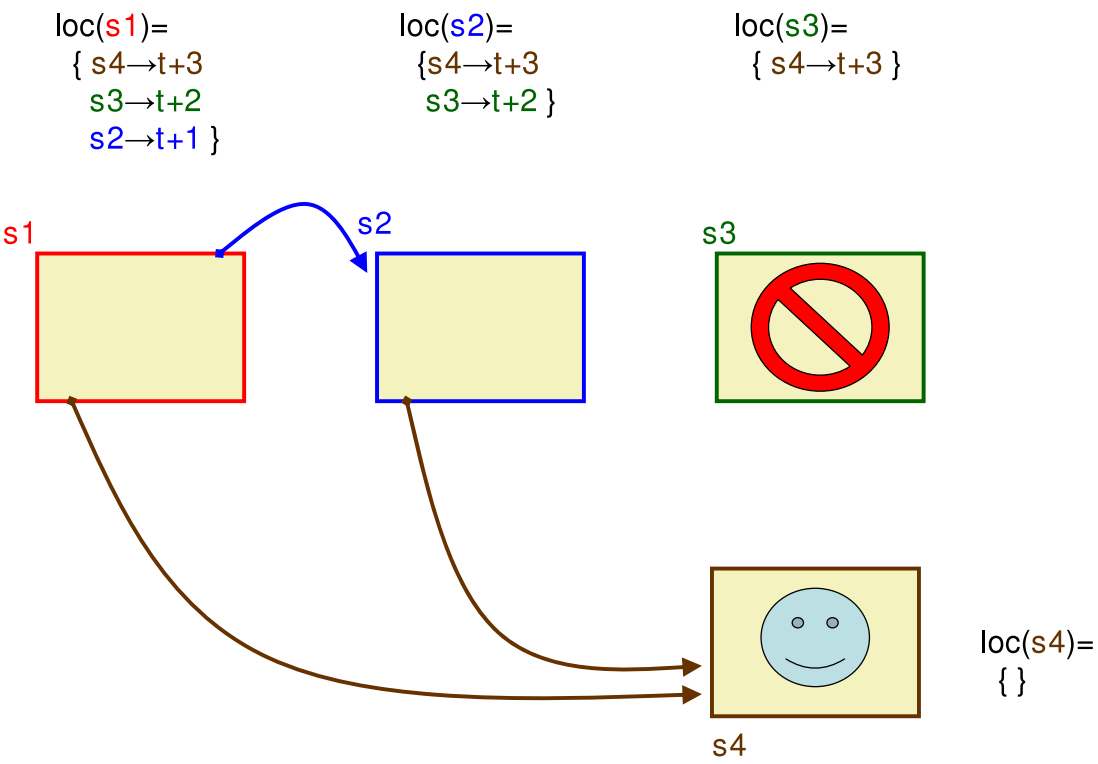


FIG. 4.14. *Alternate Paths in the Presence of Failures (5/6)*

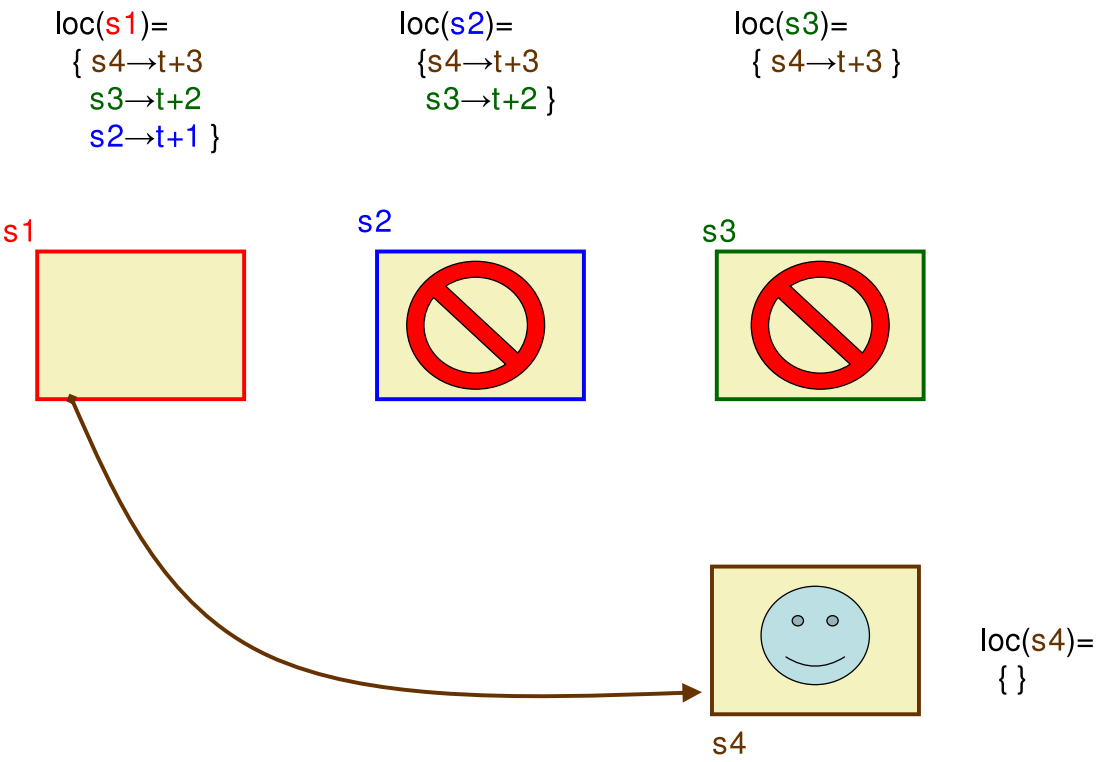


FIG. 4.15. *Alternate Paths in the Presence of Failures (6/6)*

5. Formalisation. We adopt a tried and tested framework to formalise this algorithm, which we have previously applied to several types of distributed algorithms, for mobile agents [10, 11] and for distributed reference counting [13, 12]. This formal framework has lent itself naturally to mechanical proofs in three cases.

Specifically, we model the distributed directory service as an abstract machine, whose state space is summarised in Figure 5.1. For the sake of clarity, we consider a single mobile agent; the formalisation can easily be extended to multiple agents by introducing names by which agents are being referred to. An abstract machine is composed of a set of sites taking part in a computation. Agent timestamps, which we call *mobility counters*, are defined as natural numbers. A *memory* is defined as an association list, associating locations with mobility counters; we represent an empty memory by $\emptyset$. The value N is a parameter of the algorithm. We will show that the agent's memory has a maximum size N and that the algorithm tolerates at most $N-1$ failures.

5.1. Algorithm in the Absence of Failures. The set of messages is inductively defined by two constructors. These constructors are used to construct messages, which respectively represent an agent in transit and an arrival acknowledgement. The message representing an agent in transit, typically of the form $\mathsf{agent}(s,l,\vec{M})$, contains the site s that the agent is leaving, the value l of the mobility counter it had on that site, and the agent's memory $\vec{M}$, i. e. the N previous sites it visited and associated mobility counters. The message representing an arrival acknowledgement, $\mathsf{ack}(s,l)$, contains the site s (and associated mobility counter l) where the agent is.

We assume that the network is fully connected, that communications are reliable, and that the order of messages in transit between pairs of sites is preserved. These communication hypotheses are formalised in the abstract machine by point-to-point communication links, which we define as queues using the following notations: the expression $q_1 \S q_2$ denotes the concatenation of two queues q_1, q_2, whereas $first(q)$ refers to the head of a queue q.

$$
\begin{array}{rcll}
\mathscr{S} & = & \{s_0, s_1, \ldots, s_{n_s}\} & \text{(Set of Sites)} \\
\mathscr{L} & = & \mathbb{N} & \text{(Mobility Counters)} \\
\Psi & = & list(\mathscr{S} \times \mathscr{L}) & \text{(Memory)} \\
N & \in & \mathbb{N} & \text{(Algorithm Parameter)} \\
\mathscr{M} & : & \mathsf{agent} : \mathscr{S} \times \mathscr{L} \times \Psi \to \mathscr{M} \;\mid\; \mathsf{ack} : \mathscr{S} \times \mathscr{L} \to \mathscr{M} & \text{(Messages)} \\
\mathscr{K} & = & \mathscr{S} \times \mathscr{S} \to Queue(\mathscr{M}) & \text{(Message Queues)} \\
\mathscr{LT} & = & \mathscr{S} \to \Psi & \text{(Location Tables)} \\
\mathscr{PT} & = & \mathscr{S} \to \Psi & \text{(Present Tables)} \\
\mathscr{MT} & = & \mathscr{S} \to \mathscr{L} & \text{(Mobility Counter Tables)} \\
\mathscr{AT} & = & \mathscr{S} \to \Psi & \text{(Acknowledgement Tables)} \\
\mathscr{FT} & = & \mathscr{S} \to Bool & \text{(Failure State)} \\
\mathscr{C} & = & \mathscr{LT} \times \mathscr{PT} \times \mathscr{MT} \times \mathscr{AT} \times \mathscr{FT} \times \mathscr{K} & \text{(Configurations)}
\end{array}
$$

Characteristic variables :

$s \in \mathscr{S}$	$present\_T \in \mathscr{PT}$
$m \in \mathscr{M}$	$mob\_T \in \mathscr{MT}$
$k \in \mathscr{K}$	$ack\_T \in \mathscr{AT}$
$c \in \mathscr{C}$	$fail\_T \in \mathscr{FT}$
$\vec{M} \in \Psi$	$q \in Queue(\mathscr{M})$
$loc\_T \in \mathscr{LT}$	

FIG. 5.1. *State Space*

Each site maintains some information, which we abstract as "tables" in the abstract machine. The *location table* maps each site to a memory; for a site s, the location table indicates the sites where s believes the agent has migrated to (with their associated mobility counter). The *present table* is meant to be empty for all sites, except for the site where the agent is currently located, when the agent is not in transit; there, the present table contains the sites previously visited by the agent. The *mobility counter table* associates each site with the mobility counter the agent had when it last visited the site; the value is zero if the agent has never visited the site.

After the agent has reached a new destination, acknowledgement messages have to be sent to the N previous sites it visited. We decouple the agent's arrival from acknowledgement sending, so that transitions that deal with incoming messages are different from those that generate new messages. Consequently, we introduce a further table, the *acknowledgement table*, indicating which acknowledgements still have to be sent.

In our formalisation, we use a variable to indicate whether a machine is up and running. A site's *failure state* is allowed to change from false to true, which indicates that the site is exhibiting a failure. We are modelling *stopping failures* [7] since no transition allows a failure state to change from true to false.

A complete configuration of the abstract machine is defined as the Cartesian product of all tables and message queues. Our formalisation can be regarded as an *asynchronous* distributed system [7]. In a real implementation, tables are not shared resources, but their contents can be distributed at each site.

The behaviour of the algorithm is represented by transitions, which specify how the state of the abstract machine evolves. Figure 5.2 contain all the transitions of the distributed directory service.

For convenience, we use some notations such as *post*, *receive* or table updates, which give an imperative look to the algorithm; their definitions is as follows.

- Given a configuration $\langle loc\_T, present\_T, mob\_T, ack\_T, fail\_T, k\rangle$, $mob\_T(s) := V$ denotes $\langle loc\_T, present\_T, mob\_T', ack\_T, fail\_T, k\rangle$, such that $mob\_T'(s) = V$ and $mob\_T'(s') = mob\_T(s')$, $\forall\, s' \neq s$.
- A similar notation is used for other tables.
- Given a configuration, $post(s_1, s_2, m)$ denotes $\langle loc\_T, present\_T, mob\_T, ack\_T, fail\_T, k'\rangle$, with $k'(s_1, s_2) = k(s_1, s_2)\S\{m\}$, and $k'(s_i, s_j) = k(s_i, s_j)$, $\forall (s_i, s_j) \neq (s_1, s_2)$.
- A similar notation is used for *receive*.

In each rule of Figure 5.2, the conditions that appear to the left-hand side of an arrow are guards that must be satisfied in order to be able to fire a transition. For instance, the first four rules contain a proposition of the form $\neg fail\_T(s)$, which indicates that the rule has to occur for a site s that is up and running. The right-hand side of a rule denotes the configuration that is reached after transition. We assume that guard evaluation and new configuration construction are performed atomically.

The animation we presented in the previous section directly illustrates the applications of these rules. Figure 4.1 illustrates a configuration in which an agent has previously successively visited sites s_1, s_2, s_3 with respective timestamps $t, t+1, t+2$. In this example, we assume that the value of N is 3.

The first transition of Figure 5.2 models the actions to be performed, when an agent decides to migrate from s_1 to s_2. In the guard, we see that the present table at s_1 must be non-empty, which indicates that the agent is present at s_1. After transition, the present table at s_1 is cleared, and an agent message is posted between s_1 and s_2; the message contains the agent's origin s_1, its mobility counter $mob\_T(s_1)$, and the previous content of the present table at s_1. Note that s_2, the destination of the agent, is only used to specify which communication channel the agent message must be enqueued into. The site s_1 does not need to be communicated this information, nor does it have to remember that site. In a real implementation, the agent message would also contain the complete agent code and state to be restarted by the receiver. Figure 4.3 illustrates changes in the system, after the agent has initiated its migration.

The second transition of Figure 5.2 is concerned with s_2 handling a message[1] $\mathsf{agent}(s_3, l, \vec{M})$ coming from s_1. At site s_2, tables are updated to reflect that s_2 is becoming the new agent's location, with $l+1$ its new mobility counter. Our algorithm prescribes the agent to remember N *different* sites it has visited. As s_2 may have been visited recently, we remove s_2 from $\vec{M}$, before adding the site s_3 where it was located before migration. The call $add(N, s, l, \vec{M})$ adds an association (s, l) to the memory $\vec{M}$, keeping at most N different entries with the highest timestamps. (Appendix A contains the complete definition of *add*.) In addition, the acknowledgement table of s_2 is updated, since acknowledgements have to be sent back to those previously visited sites. At this point, a proper implementation would reinstate the agent state and resume its execution. Figure 4.3 illustrates the system as an agent arrives at a new location.

According to the third transition of Figure 5.2, if the acknowledgement table on s_1 contains a pair (s_2, l_2), then an acknowledgement message $\mathsf{ack}(s_1, (mob\_T(s_1)))$ has to be sent from s_1 to s_2; the acknowledgement message indicates that the agent is on s_1 with a mobility counter $mob\_T(s_1)$.

According to transition receive_ack, if a site s_2 receives an acknowledgement message about site s_3 and mobility counter l, its location table has to be updated accordingly. Let us note two properties of this rule. First, we do not require the emitter s_1 of the acknowledgement message to be equal to s_3; this property allows us to overload the meaning of this message and use it for sharing information about the agent's location. Second, we make sure that updating the location

[1] Note that s_3 is not required to be equal to s_1. Indeed, we want the algorithm to be able to support sites that forward incoming agents to other sites.

For a configuration $\langle loc\_T, present\_T, mob\_T, ack\_T, fail\_T, k\rangle$, legal transitions are:

$$
\begin{array}{l}
\mathsf{migrate\_agent}(s_1, s_2): \\
\quad s_1 \neq s_2 \ \wedge\ loc\_T(s_1) = \emptyset \ \wedge\ present\_T(s_1) \neq \emptyset \\
\quad \wedge\ ack\_T(s_1) = \emptyset \ \wedge\ \neg fail\_T(s_1) \\
\quad\quad \rightarrow\ \{\quad let\ \vec{M} = present\_T(s_1) \\
\quad\quad\quad\quad in\quad present\_T(s_1) := \emptyset \\
\quad\quad\quad\quad\quad\quad post(s_1, s_2, \mathsf{agent}(s_1, mob\_T(s_1), \vec{M}))\ \ \} \\
\\
\mathsf{receive\_agent}(s_1, s_2, s_3, l, \vec{M}): \\
\quad first(k(s_1, s_2)) = \mathsf{agent}(s_3, l, \vec{M})\ \wedge\ \neg fail\_T(s_2) \\
\quad\quad \rightarrow\ \{\quad receive(s_1, s_2) \\
\quad\quad\quad\quad let\quad S' = add(N, s_3, l, remove(s_2, \vec{M})) \\
\quad\quad\quad\quad in\quad loc\_T(s_2) := \emptyset \\
\quad\quad\quad\quad\quad\quad present\_T(s_2) := S' \\
\quad\quad\quad\quad\quad\quad mob\_T(s_2) := l + 1 \\
\quad\quad\quad\quad\quad\quad ack\_T(s_2) := S'\ \ \} \\
\\
\mathsf{send\_ack}(s_1, s_2, \vec{M}, l_2): \\
\quad ack\_T(s_1) = (s_2, l_2)\ \S\ \vec{M}\ \wedge\ \neg fail\_T(s_1) \\
\quad\quad \rightarrow\ \{\quad ack\_T(s_1) := \vec{M} \\
\quad\quad\quad\quad post(s_1, s_2, \mathsf{ack}(s_1, mob\_T(s_1)))\ \ \} \\
\\
\mathsf{receive\_ack}(s_1, s_2, s_3, l): \\
\quad first(k(s_1, s_2)) = \mathsf{ack}(s_3, l)\ \wedge\ \neg fail\_T(s_2) \\
\quad\quad \rightarrow\ \{\quad receive(s_1, s_2) \\
\quad\quad\quad\quad loc\_T(s_2) := add(N, s_3, l, loc\_T(s_2))\quad \} \\
\\
\mathsf{inform}(s_1, s_2, s_3, l): \\
\quad (s_3, l) \in loc\_T(s_1)\ \wedge\ \neg fail\_T(s_1) \\
\quad\quad \rightarrow\ \{\quad post(s_1, s_2, \mathsf{ack}(s_3, l))\ \ \} \\
\\
\mathsf{stop\_failure}(s): \\
\quad fail\_T(s) = false \\
\quad\quad \rightarrow\ \{\quad fail\_T(s) = true\ \} \\
\\
\mathsf{msg\_failure}(s_1, s_2, m): \\
\quad first(k(s_1, s_2)) = m\ \wedge\ fail\_T(s_2) \\
\quad\quad \rightarrow\ \{\quad receive(s_1, s_2)\ \}
\end{array}
$$

FIG. 5.2. *Fault-Tolerant Directory Service*

table *(i)* maintains information about *different* locations, *(ii)* does not overwrite existing location information with older one. This functionality is implemented by the function *add*, whose specification required careful design to ensure correctness of the algorithm and can be found in appendix A.

According to rule inform of Figure 5.2, any site s_1 believing that the agent is located at site s_3, with a mobility counter l, may elect to communicate its belief to another site s_2. Such a belief is also communicated by an ack message. (It is here that we overload the meaning of the original ack message to allow information sharing.) It is important to distinguish the roles of the send_ack and inform transitions. The former is mandatory to ensure the correct behaviour of the algorithm, whereas the latter is optional. The purpose of inform is to propagate information about the agent's location in the system, so that the agent may be found in less steps. As opposed to previous rules, the inform rule is non-deterministic in the destination and location information in an acknowledgement message. At this level, our goal is to define a correct *specification* of an algorithm: any implementation strategy will be an instance of this specification; some of them are discussed in Section 7. Figures 4.6 illustrates the states of the system after sending acknowledgement messages, whereas Figures 4.7, 4.8, 4.9 show the effects of such messages.

5.2. Failures. The first five rules of Figure 5.2 require the site s where the transition takes place to be up and running, i. e. $\neg fail\_T(s)$. Our algorithm is designed to be tolerant to *stopping failure*, according to which processes are allowed to stop somewhere in the middle of their execution [7]. We model a stopping failure by the transition stop_failure, changing the failure state of the site that exhibits the failure. Consequently, a site that has stopped will be prevented from performing any of the first five transitions of Figure 5.2.

As far as distributed system modelling is concerned, it is unrealistic to consider that messages that are in transit on a communication link remain present if the destination of the communication link exhibits a failure. Rule msg_failure shows how messages in transit to a stopped site may be lost. A similar argument may also hold for messages that were posted (but not sent yet) at a site that stops. We could add an extra rule handling such a case, but we did not do so in order to keep the number of rules limited. Thus, our communication model can be seen as using buffered inputs and unbuffered outputs.

5.3. Initial and Legal Configurations. In the initial configuration, noted c_i, we assume that the agent is at a given site *origin* with a mobility counter set to $N+1$. Obviously, at creation time, an agent cannot have visited N sites previously. Instead, the creation process elects a set $\mathscr{S}_i$ of different sites that act as "backup routers" for the agent in the initial configuration. Each site is associated with a different mobility counter in the interval $[1,N]$. Such N sites could be chosen non-deterministically by the system or could be configured manually by the user. For each site in $\mathscr{S}_i$, the location table points to the origin and to sites of $\mathscr{S}_i$ with a higher mobility counter; the location table at all other sites contains the origin and the $N-1$ first sites of $\mathscr{S}_i$. The present table at *origin* contains the sites in $\mathscr{S}_i$. A detailed formalisation of the initial configuration is available from [9]. A configuration c is said to be *legal* if there is a sequence of transitions $t_1, t_2, \ldots, t_n$ such that c is reachable from the initial configuration: $c_i \mapsto^{t_1} c_1 \mapsto^{t_2} c_2 \ldots \mapsto^{t_n} c$. We define $\mapsto^*$ as the reflexive, transitive closure of $\mapsto$.

6. Correctness. The correctness of the distributed directory service is based on two properties: safety and liveness. The *safety* of the distributed directory service ensures that it correctly tracks the mobile agent's location, in particular in the presence of failures. The *liveness* guarantees that agent location information eventually gets propagated.

We *intuitively* explain the safety property proof as follows. An acknowledgement message results in the creation of a forwarding pointer that points towards the agent's location. Forwarding pointers may be modelled by a relationship *parent* that defines a directed acyclic graph leading to the agent's location.

In the presence of failures, we show that the relationship parent contains sufficient redundancy in order to guarantee the existence of a path leading to the agent, without involving any failed site: *(i)* Sites that belong to the agent's memory have the agent's location as a parent. *(ii)* Sites that do not belong to the agent's memory have at least N parents. Consequently, if the number of failures is strictly inferior to N, each site has always at least one parent that is closer to the agent's location; by repeating this argument, we can find the agent's location.

We summarise the liveness result similar to the one in [10]. A *finite* amount of transitions can be performed from any legal configuration (if we exclude migrate_agent and inform). Furthermore, we can prove that, if there is a message at the head of a communication channel, there exists a transition of the abstract machine that consumes that message. Consequently, if we assume that message delivery and machine transitions are fair, and if the mobile agent is stationary at a location, then location tables will eventually be updated, which proves the liveness of the algorithm.

All proofs were mechanically derived using the proof assistant Coq [1]. Coq is a theorem prover whose logical foundation is constructive logic. The crucial difference between constructive logic and classical logic is that $\neg\neg p \supset p$ does not hold in constructive logic. The consequence is that the formulation of proofs and properties must make use of constructive and decidable statements. Due to space restriction, we do not include the proofs but they can be downloaded from [9]. The notation adopted in the paper is pretty-printed concise version of the mechanically established one.

7. Algorithm and Proof Discussion. The constructive proof of the initial algorithm without fault-tolerance helped us understand the different invariants that needed to be preserved. In particular, the algorithm maintains a directed acyclic graph leading to the agent's position; interestingly, short-cutting chains of pointers by propagating acknowledgement messages ensures that the graph remains connected and acyclic. Using the same mechanism of timestamp in combination with replication preserves a similar invariant in the presence of failures.

Interestingly, the fault-tolerant algorithm turned out to be simpler than its non-fault-tolerant version, because it uses less rules; furthermore, its correctness proof was easier to derive. When N is equal to 1, the algorithm has the same observable behaviour as [10]. From a practical point of view, generating the mechanical proof still remained a tedious process, though simpler, because it needed some 25000 tactic invocations, of which 5000 for the formalisation of the abstract machine were reused from our initial work.

The complexity of the algorithm is linear in N as far as the number of messages (N acknowledgement messages per migration), message length (size of a memory is $O(N)$), space per site (size of a memory is $O(N)$), and time per migration are concerned. Our proof established the correctness in the worst-case scenario. Indeed, the algorithm may tolerate more than N failures provided that one parent, at least, remains up and running for each site.

For a given application, the designer will have to choose the value of N. If N is chosen to be equal to the number of nodes in the network, the system will be fully realiable but its complexity, even though linear, is too high on an Internet scale. Instead, an engineering decision should be made: in a practical network, from network statistics, one can derive the probability of obtaining $1, 2, \ldots, N$ simultaneous failures. For each application, and for the quality of service it requires, the designer selects the appropriate failure probability, which determines the number of simultaneous failures the system should be able to tolerate.

A remarkable property of the algorithm is that it does not impose any delay upon agents when they initiate a migration. Forwarding pointers are created temporarily until a stable situation is reached and they are removed. This has to be contrasted with the home agent approach, which requires the agent to notify its homebase, before and after each migration. Interestingly, our algorithm does not preclude us also from using other algorithms; we could envision a system where algorithms are selected at runtime according to the network conditions and the quality of service requirements of the application.

Propagating agent location information with rule inform is critical in order to shorten chains of forwarding pointers, because shorter chains reduce the cost of finding an agent's location. The ideal strategy for sending these messages depends on the type of distributed system, and on the applications using the directory service. A range of solutions is possible and two extremes of the spectrum are easily identifiable. In an eager strategy, every time a mobile agent migrates, its new location is broadcasted to all other sites; such a solution is clearly not acceptable for networks such as the Internet. Alternatively, a lazy strategy could be adopted [14] but it requires cooperation with the message router. The recipient of a message may inform its emitter, when the recipient observes that that the emitter has out-of-date routing information. In such a strategy, tables are only updated when application messages are sent.

In Section 5, communication channels in the abstract machine are defined as queues. We have established that swapping any two messages in a given channel does not change the behaviour of the algorithm; in other words, messages do not need to be delivered in order.

Message Router. This paper studied a distributed directory service, and we can sketch two possible uses for message routing.

Simple Routing. The initial message router [10] can be adopted to the new distributed directory service. A site receiving a message for an agent that is not local forwards the message to the site appearing in its location table with the highest mobility counter; if the location table is empty, messages are accumulated until the table is updated. This simple algorithm does not use the redundancy provided by the directory service and is therefore not tolerant to failure.

Parallel Flooding. A site must endeavour to forward a message to N sites. If required, it has to keep copies of messages until N acknowledgements have been received. By making use of redundancy, this algorithm would guarantee the delivery of messages. We should note that the algorithm needs a mechanism to clear messages that have been delivered and are still held by intermediate nodes.

8. Further Related Work. Murphy and Picco [15] present a reliable communication mechanism for mobile agents. Their study is not concerned with nodes that exhibit failures, but with the problem of guaranteeing delivery in the presence of runaway agents. Whether their approach could be combined with ours remains an open question.

Lazar *et al.* [6] migrate mobile agents along a logical hierarchy of hosts, and also use that topology to propagate messages. As a result, they are able to give a logarithmic bound on the number of hops involved in communication.

Their mechanism does not offer any redundancy: consequently, stopping failures cannot be handled, though they allow reconnections of temporarily disconnected nodes.

Baumann and Rothermel [2] introduce the concept of a shadow as a handle on a mobile agent that allows applications to terminate a mobile agent execution by notifying the termination to its associated shadow. Shadows are also allowed to be mobile. Forwarding pointers are used to route messages to mobile agents and mobile shadows. Some fault-tolerance is provided using a mechanism similar to Jini leases, requiring message to be propagated after some timeout. This differs from our approach that relies on information replication to allow messages to be routed through multiple routes.

Mobile computing devices share with mobile agents the problem of location tracking. Prakash and Singhal [19] propose a distributed location directory management scheme that can adapt to changes in geographical distribution of mobile hosts population in the network and to changes in mobile host location query rate. Location information about mobile hosts is replicated at $O(\sqrt{m})$ base stations, where m is the total number of base stations in the system. Mobile hosts that are queried more often than others have their location information stored at a greater number of base stations. The proposed algorithm uses replication to offer improved performance during lookups and updates, but not to provide any form of fault tolerance.

9. Conclusion. In this paper, we have presented a fault-tolerant distributed directory service for mobile agents. Combined with a message router, it provides a reliable communication layer for mobile agents. The correctness of the algorithm is stated in terms of its safety and liveness.

Our formalisation is encoded in the mechanical proof assistant Coq, which we used for carrying out the proof of correctness. The constructive proof gives an ideal insight of the algorithm that can be exploited to specify a reliable message router. Overall, this would lead to a fully mechanically proven correct mobile agent system. Besides message routing, other problems deserve formalisation and mechanical proofs, such as security and authentication methods for mobile agents.

10. Acknowledgements. Thanks to Paul Groth and Zheng Chen for their feedback on the paper. This work was initiated during the QinetiQ and EPSRC sponsored Magnitude project (reference GR/N35816).

REFERENCES

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. M. noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner, *The Coq Proof Assistant Reference Manual – Version V6.1*, Tech. Report 0203, INRIA, August 1997.

[2] J. Baumann and K. Rothermel, *The shadow approach: An orphan detection protocol for mobile agents*, in Second Int. Workshop on Mobile Agents MA'98, Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 2–13.

[3] K. Bharat and L. Cardelli, *Migratory Applications*, in Mobile Object Systems: Towards the Programmable Internet, C. Tschudin and J. Vitek, eds., Springer-Verlag, Apr. 1997, pp. 131–149. Lecture Notes in Computer Science No. 1222.

[4] G. Cabri, L. Leonardi, and F. Zambonelli, *Reactive Tuple Spaces for Mobile Agent Coordination* , in Proceedings of the 2nd International Workshop on Mobile Agents (MA'98), no. 1477 in LNCS, 1998.

[5] D. B. Lange and M. Ishima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[6] S. Lazar, I. Weerakoon, and D. Sidhu, *A Scalable Location Tracking and Message Delivery Scheme for Mobile Agents*, tech. report, University of Maryland, 1998.

[7] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, Dec. 1995.

[8] S. Mishra, X. Jiang, and B. Yang, *Providing Fault Tolerance to Mobile Intelligent Agents*, in Proceedings of the ISCA 8th International Conference on Intelligent Systems, Denver, June 1999.

[9] L. Moreau, *A Fault-Tolerant Distributed Directory Service for Mobile Agents: the Constructive Proof in Coq*. Available from `http://www.ecs.soton.ac.uk/~lavm/projects/algorithms/coq/failure/`, Sept. 2000.

[10] ———, *Distributed Directory Service and Message Router for Mobile Agents*, Science of Computer Programming, 39 (2001), pp. 249–272.

[11] ———, *A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers*, in The 17th ACM Symposium on Applied Computing (SAC'2002) — Track on Agents, Interactions, Mobility and Systems, Madrid, Spain, Mar. 2002, pp. 93–100.

[12] L. Moreau, P. Dickman, and R. Jones, *Birrell's Distributed Reference Listing Revisited*, ACM Transactions on Programming Languages and Systems (TOPLAS), 27 (2005), p. 52.

[13] L. Moreau and J. Duprat, *A Construction of Distributed Reference Counting*, Acta Informatica, 37 (2001), pp. 563–595.

[14] L. Moreau and D. Ribbens, *Mobile Objects in Java*, Scientific Programming, 10 (2002), pp. 91–100. Special issue of the International Workshop on Performance-oriented Application Development for Distributed Architectures (PADDA'2001).

[15] A. L. Murphy and G. P. Picco, *Reliable Communication for Highly Mobile Agents*, in First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA'99), Oct. 1999.

[16] S. Nog, S. Chawla, and D. Kotz, *An RPC mechanism for transportable agents*, Tech. Report TR96-280, Department of Computer Science, Dartmouth College, Hanover, N.H., 1996.

[17] ObjectSpace, *Voyager*. `http://www.objectspace.com/`.

[18] G. P. Picco, A. L. Murphy, and G.-C. Roman, *LIME: Linda meets mobility*, in International Conference on Software Engineering, 1999, pp. 368–377.

[19] R. PRAKASH AND M. SINGHAL, *A Dynamic Approach to Location Management in Mobile Computing Systems*, in The 8th International Conference on Software Engineering and Knowledge Engineering (SEKE'96), Lake Tahoe, Nevada, June 1996, pp. 488–495.
[20] H. K. TAN, *Interaction tracing for mobile agent security*, PhD thesis, Electronics and computer science, Southampton, UK, Mar. 2004.
[21] G. VIGNA, ed., *Mobile Agents and Security*, vol. 1419 of LNCS State-of-the-Art Survey, Springer-Verlag, June 1998.
[22] P. WOJCIECHOWSKI AND P. SEWELL, *Nomadic Pict: Language and Infrastructure Design for Mobile Agents*, in First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA'99), Oct. 1999.

Appendix A. ADD FUNCTION. The function *add* adds a pair site–timestamp to an association list, making sure that no two entries have a same timestamp or site. A maximum of N entries is kept in the association list, and they are sorted by decreasing timestamp order.

A functional definition of *add* (close to its definition in Coq) appears below, and it uses auxiliary functions `remove` to remove an entry with a specific site from an association list and `firstN` which keeps the first N entries of an association list.

```
fun add (N:int;s1:site;n1:int;q:(Alist site int)) :=
   (firstN N (insert s1 n1 q))

fun insert (s:site;n:int;q:(Alist site int)) :=
  match q
   nil => (cons (s,l) q)
   (cons (s1,n1) q') =>
       if (s=s1)
       then
         if (n <=  n1)
         then q
         else (cons (s,n) q')
       else
         if (n<n1)
         then (cons (s1,n1) (insert s n q'))
         elif (n=n1)
         then q
         else (cons (s,n) (remove s q))
```

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 12, 2006

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 89–99. http://www.scpe.org
ISSN 1895-1767

GENERATIVE MOBILE AGENT MIGRATION IN HETEROGENEOUS ENVIRONMENTS

B. J. OVEREINDER, D. R. A. DE GROOT, N. J. E. WIJNGAARDS, AND F. M. T. BRAZIER*

Abstract. Agent migration, in theory, makes it possible to bring computations to the resources required. In practice, however, homogeneity in programming language and/or agent platform is required. This paper presents an approach that supports heterogeneous agents and platforms: agents written in different languages can migrate between non-identical platforms. Instead of migrating the "code" (including data and state) of an agent, a blueprint of an agent's functionality is transferred (together with its state). An agent factory on the receiving platform generates new code on the basis of this blueprint. This approach of *generative mobility* not only has implications for interoperability but also for security, as discussed in this paper.

Key words. mobile agents, process migration, compositional design

1. Introduction. Agents are used to solve complex problems in distributed environments, in which heterogeneity of protocols, runtime environment, operating system and hardware resources are a fact of life. Agents migrate from one machine to another: moving computations to resources and/or services, overcoming the high latency or limited bandwidth problem of traditional client-server interactions [19], and providing resource owners a means to control access to their resources. The current evolution of intelligent networked systems and Grid management, for example, is based on this technology [7]. A similar tendency is observed in the search and filtering of global information in the electronic marketplaces, e-commerce, and information retrieval on the World Wide Web [20].

To support agent mobility a distributed system needs provisions to physically migrate units of computation at runtime. This migration includes relocation of an agent's code base and state to another platform. Code and state migration is a complex task with technical complications such as incompatibility of platforms. The current solution to migration of active units of computation is to provide homogeneous platforms, either physically such that binary checkpoints can be restarted at another location, or virtually by using virtual machines, e.g., the Java Virtual Machine, providing a machine independent platform. The homogeneity requirement, physical or virtual, is a strong requirement: mobility is otherwise impossible. Another solution is for an agent to carry different versions of its binaries (or URLs) depending on the platforms it may be expected to encounter. The same objections, however, hold: the platforms need to be pre-specified.

Another important issue in mobile agents technology is security. In most current systems trust in the owners and in the machines on which an agent has previously run, are the only basis for a security model. Code signing and certificates are the techniques used to this purpose.

This paper presents an alternative approach to agent mobility and security. Not the code migrates, but an agent's blueprint (implementation independent description) and state. A receiving platform *regenerates* a mobile agent as it migrates to its new location. Homogeneity is no longer required: an agent programmed in Java can be transformed to, for example, a Python implementation of the agent with the same functionality. Trust in an agent coming from another machine increases considerably if the receiving platform uses its own trusted components to reconfigure an agent.

Section 2 discusses mobility of processes and agents in more detail. Section 3 describes the concept of an agent factory. Heterogeneous migration based on this concept is the topic of Section 4. Implications of this approach for heterogeneous migration and security are discussed at more length Section 6. Section 8 sums up the results and proposes future research.

2. Agent Migration.

2.1. Mobility. An agent in a mobile agent system is typically associated with a unit of computation which resides in the lower layers of a virtual machine. A unit of computation is composed of the code describing its behaviour, the associated data, and its execution state. Mobile agent systems allow migration of the whole unit or a part thereof, i. e., one or more of the three constituents mentioned above. The most relevant differences among existing systems lie exactly in what is moved and how [31].

A distinction can be made between systems that migrate execution state along with the unit of computation and those that do not. Systems that do, are said to support *strong mobility*, as opposed to systems that discard the execution state across migration, and are hence said to provide *weak mobility*. In systems supporting strong mobility, migration is completely transparent. In systems supporting weak mobility, if part of an execution state is needed at a later state in an agent's lifetime, it needs to be explicitly saved.

*IIDS Group, Department of Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Strong mobility, as found in NOMADS [37], Ara [29], D'Agents [15], requires that the entire state of an agent, including its execution stack and program counter, is saved before an agent is migrated to a new location. This process of saving the entire state of an executing process is called *checkpointing*. An important quality of strong mobility is transparent migration of the running process. That is, the agent is unaware of the migration, bindings to other agents and objects are transparently resolved, i. e., references to agents and objects are location independent. The checkpoint/migration facility is either implemented at the operating system level [16, 17, 28] or incorporated within the virtual machine of an interpreted language (e.g., within the Java Virtual Machine [37]).

Despite the advantages of strong mobility, many agent systems support weak mobility (like Ajanta [38], Aglets [18], Grasshopper [1], AgentScape [27] and JADE [2]). Most current agent systems are implemented on top of the Java Virtual Machine (JVM), which provides object serialization and basic mechanisms to implement weak mobility. The JVM does not provide mechanisms to deal with execution state transfer.

Agent mobility is relatively straightforward in homogeneous environments. For strong mobility with checkpoint/migration incorporated at the operating system level, agent mobility is limited to identical computer architectures running the same operating system. Agent mobility facilities implemented at the virtual machine level makes the migration of agents machine independent, but is still homogeneous in language, i. e., only migration of agents from Java to Java platforms is possible.

2.2. Heterogeneity & Interoperability. Migration of mobile agents does not need to be constrained by homogeneity of code bases and platforms. *Generative migration* is an option. Blueprints of the functionality of an agent, together with its state, are the basis of this type of migration. A blueprint defines the functional components of which an agent is composed: implementations of these components differ between platforms. When an agent decides to migrate its blueprint and state are transferred to the designated platform. The agent is regenerated on this platform according to its blueprint. The same functional components are used. The components, however, may be coded in a different code base, the code base supported by the agent platform to which the agent migrated. The agent platform may also differ: its the blueprint that matters.

Regeneration relies on an automated facility (*agent factory*) on the receiving platform to reconfigure an agent on the basis of its blueprint and state. The interface with which an agent communicates to the platform on which it runs can differ significantly.

Taking this interface into account, and the specific machines on which an agent runs, i. e., the host, the following migration scenarios can be distinguished.

Homogeneous migration An agent migrates to another host without any changes to the format of its executable code or the interfaces to the agent platform. This form of migration requires that source and destination platform offer the same interfaces, but also that the (virtual) machine that executes the agent is the same on both sides. In practice, this form of migration is most common.

Cross-platform migration An agent migrates to another host with a different agent platform, one that offers the same (virtual) machine architecture. This generally entails changes to the interface to the agent platform, but not necessarily changes to the format of its executable code. This form of migration may occur when, e.g., a Java-agent migrates from a Ajanta platform to a Zeus platform. One commonly applied solution is to offer wrapper interfaces that hide the differences between source and target platform. Another approach, followed in MAF [26] or FIPA, is to enforce platforms to implement a standard interface for interoperability.

Agent-regeneration migration An agent migrates to a host running a different (virtual) machine. This requires an agent to be regenerated, resulting in different executable code. Note that the target agent platform may be the same as that of the source, simplifying regeneration. To regenerate an agent, it is necessary that the target has a blueprint of the agent.

Heterogeneous migration An agent migrates to another host with a different agent platform with a different (virtual) machine. In this case, regeneration of the agent is necessary. As the underlying agent platform is also different the agent's blueprint must be platform independent.

This paper advocates the support of heterogeneous migration as it offers the most flexibility. As distributed systems are gradually required to scale worldwide across different administrative organizations, and to support a myriad of platforms, solutions are needed that anticipate heterogeneity and adaptability. Regeneration of agents for different underlying platforms is a step towards meeting such requirements.

The approach described in this paper combines heterogeneous migration with weak migration. The term proposed for our approach introduced above *generative migration*. Generative migration for agents may open the world of distributed

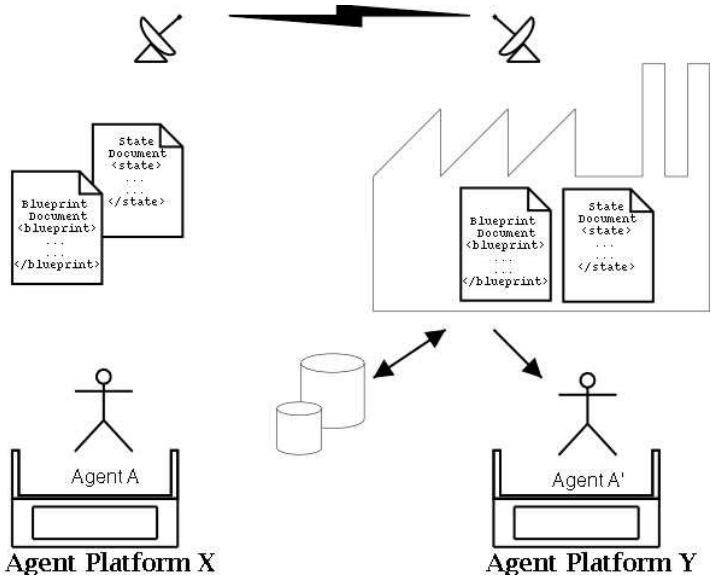


FIG. 3.1. *Principles of generative migration.*

systems to agent-developers. The adage "write once, run everywhere" is achieved while retaining heterogeneity and tackling the problem of interoperability.

Generative migration requires that a target host has access to an agent factory capable of generating an agent for that target. Ideally, this factory is placed on the target host or another machine on the same local-area network. An important issue is that the factory is trusted to generate an agent that the target host/platform can trust. Security and trust are briefly discussed in Section 6.

Our approach has the additional benefit that various optimizations become possible. For example, an agent generated by a factory may be optimized with respect to the target's machine architecture, or the way that local resources such as databases are accessed. In addition, transmission of an agent's blueprint and information on its current state will generally require fewer network resources than migrating an agent using more traditional approaches. On the downside, the agent-generation process may affect overall performance if agents migrate frequently.

3. Generative Agent Migration. Assuming agents have a compositional structure described by their *blueprints*, building an agent is, in fact, a configuration task: a task that can be automated. Automated (re-)design of agents is the task of an agent factory [5]. An agent factory generates from this blueprint an agent for a specific platform (in many ways this functionality is comparable to a compiler). This section describes reconfiguration by an agent factory.

The process of generative migration is illustrated in Figure 3.1. An agent's blueprint describes its structure and functionality. Its state (including private data) is needed for its execution. Both the blueprint and state are described in a format independent of the operating system and agent system, e.g., in XML. The blueprint and state are transferred to the destination platform requested by the agent. The destination's agent factory re-builds the agent on the basis of the blueprint and state it's received. Regeneration of the agent relies on availability of the appropriate building blocks. These building blocks are retrieved from a local repository. Finally, the regenerated agent is initiated with its state, on the platform for which it has been generated.

Section 3.1 discusses characteristics of an agent factory, and Section 3.2 illustrates this concept for a specific type of agent, namely an information retrieval agent. Section 3.3 describes a current prototype of this agent factory.

3.1. Concepts of an Agent Factory. Whether the need for adaptation is identified by an agent itself, or by another agent, is irrelevant in the context of this paper. An agent factory simply constructs new agents and/or modifies existing agents [5]. The (re-)design of agents is fully automated, with very limited interaction with outside parties. The concept of an agent factory requires (i) agents to have a compositional structure, (ii) one or more libraries of re-usable agent components, and (iii) one or more ways to describe the functionality of these agent components.

In the agent factory discussed in this section two additional assumptions hold: (i) two levels of description are distinguished: conceptual and detailed, and (ii) no commitments are made to specific programming languages and/or ontologies.

A conceptual description of (parts of) an agent is an architectural description: a blueprint of the components, interfaces and interactions between components. A detailed description includes code, together with definitions of, e.g., interfaces. A mapping between these descriptions defines the relationship between the elements at one level with elements at the other. This mapping may be structure preserving. (Note that this is not always ideal.) A number of detailed components may exist for each conceptual component (e.g., one in C, another in Java), and vice versa.

The concept of a building block is used to describe the components within an agent factory at both levels. Some building blocks contain open slots, others are fully specified and operational. Both define their functionality on the basis of their interfaces. Open slots define the interfaces of the building blocks to be inserted.

Depending on availability and domain of application libraries of building blocks may include: partial agent designs (cf., generic models/design patterns [13, 30, 33]), knowledge-based models (e.g., problem-solving models [35] or generic task models [4]), agent-wrappers (providing cross platform interfaces) (e.g., AgentScape, Zeus [25], message parsing Ajanta [38]), et cetera. Building blocks may be written in, e.g., UML, Python, C++, CommonKads, etc.

3.2. Agents and Building Blocks. A blueprint of an agent contains descriptions of the interfaces of building blocks and their open slots, and additional information on the relation between the building block configurations at the two levels of detail.

Consider, for example, the architecture of a simple information retrieval agent. This simple agent is only capable of communication with one other agent, e.g., its owner, and interaction with external resources using one protocol. The template for this simple information retrieval agent contains the agent architecture shown in Fig 3.2. A number of the components and data structures contain open slots in this model specifying the interfaces of the required building blocks. These are not depicted in this figure.

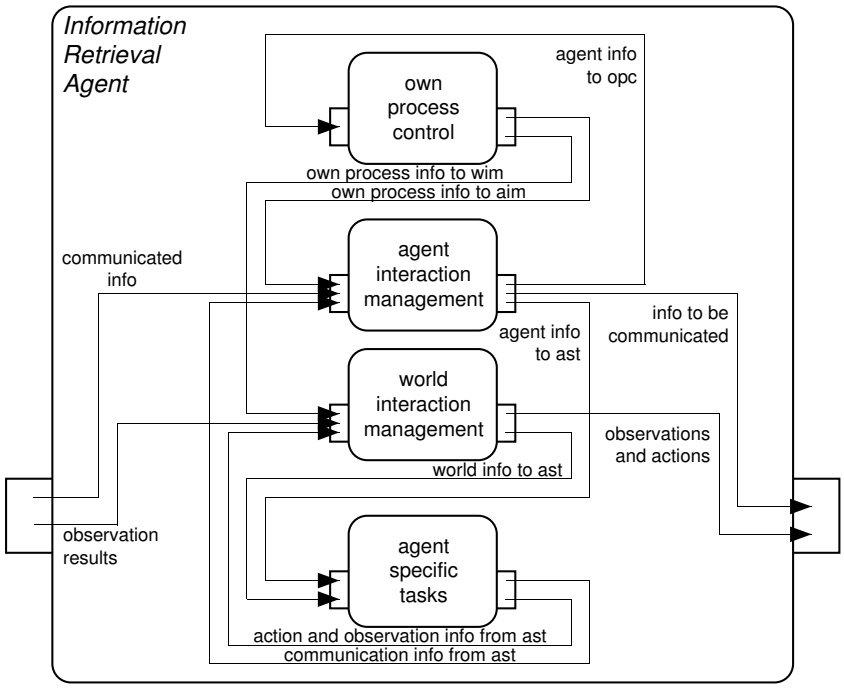


FIG. 3.2. *Architecture of a simple information retrieval agent.*

The component Own Process Control (opc) has an open slot for a building block with knowledge of an agent's identity and the agent's preferences for interactions with resources. The component World Interaction Management (wim) has an open slot for a building block capable of managing interaction with external resources. The component Agent Specific Task (ast) has an open slot for a building block capable of transforming user requests into queries, and query results into a format/language users may understand. The fourth component of the agent, Agent Interaction Management (aim), does not have an open slot.

The control inside this building block is pre-defined, no control slot is available for extension. A number of data-structures used by the agent need to be extended, see Section 4. The library contains a detailed building block for this information retrieval agent template in Java. The structure of the code mirrors the architecture of the agent.

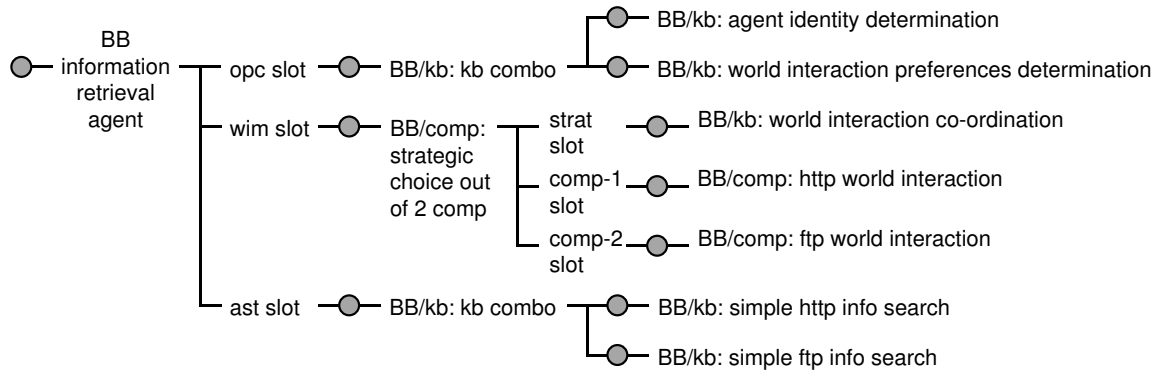


FIG. 3.3. *Building block configuration of a simple information retrieval agent.*

Figure 3.3 illustrates a building block-configuration in which two levels of building blocks were required: each open slot required a building block that itself contained other open slots. Note that the lower level building blocks make a distinction between open slots for data, and open slots for processes.

3.3. Agent Factory Prototype. A prototype of the agent factory has been implemented. This prototype automatically (re-)configures an information retrieval agent depending on the site to which it migrates.

Conceptual components are specified in the DESIRE framework [3, 4]. The compositional nature of DESIRE models, and the separation between processes and knowledge makes it possible to specify the functionality of knowledge intensive systems as the composition of (reusable) components. A structure-preserving mapping exists between the configuration of building blocks at the conceptual level of abstraction (i. e., DESIRE specifications) and the configuration of building blocks at the detailed level of abstraction. The detailed components are in Java.[1]

The prototype agent factory itself is written in Java, and contains the knowledge needed to (re-)design simple information retrieval agents.

4. Migration Using the Agent Factory Service. One of the strengths of the agent factory concept is that it provides a means to support migration of agents in heterogeneous environments that require a high level of security. Section 4.1 discusses pre-conditions for successful migration of agents. Section 4.2 describes the approach in agent-factory-enhanced migration.

4.1. Migration Pre-Conditions. To facilitate the description of agent migration, an agent is assumed to consist of executable code and state. Executable code may contain "code and data," if these can be distinguished, or may be inseparable (as in Prolog). When an agent migrates, it needs to retain sufficient information from its state to resume execution at its destination. Note that this description leans towards weak-mobility: it may not be necessary to transport the entire state of an agent. This, however, depends on the application.

Although it is not necessary for the source and destination host to both have access to an agent factory, it greatly simplifies descriptions of the migration process. An agent needs to be able to store and restore information on its state; this is a requirement for interoperability. An implementation-independent format such as XML, RDF, or OIL can be used to this purpose.

The agent factories on the hosts need to share the same functional components. That is, both have the same libraries of building blocks at the conceptual level of abstraction, but may have different libraries of building blocks at the detailed level, depending on the nature of the destination location. For example, an agent factory may have a mapping from a conceptual agent architecture building block (its functional components) to a detailed building block written in Java, while another agent factory may have a detailed building block written in C++.

4.2. Approach to Migration. In essence, migration entails moving an agent from one machine to another. This usually involves pre-packaging an agent before its move, such that its executable code and state may be restored at the destination host. Migration using an agent factory diverges from standard mobility of agents in that *not* an agent's executable code with state is migrated, but its blueprint together with (parts of) its state. This might seem to be similar

[1] Automated prototype generation within the DESIRE framework on the basis of detailed formal specifications facilitates verification and validation of knowledge intensive systems; this feature is not used within the current prototype of the agent factory.

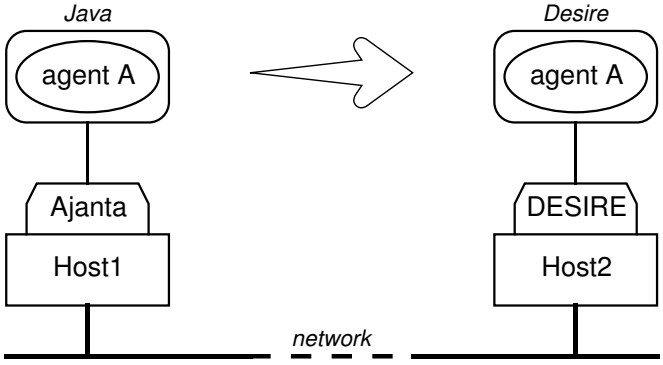


FIG. 4.1. *Example migration scenario in which agent A on host 1 (written in Java, running on Ajanta) migrates to host 2 (where it will be specified in* DESIRE *and running on* DESIRE*).*

to Java agents and their interaction with class loader objects that allows specific implementations of Java classes to be loaded; however, in our approach a blueprint of an agent can be sent to any arbitrary platform like Java, Python, or Prolog.

Consider the scenario for heterogeneous mobility, depicted in Fig. 4.1. An information retrieval agent A resides on a host machine H1. This host runs the Ajanta [38] agent platform, and, as such, supports Java agents. The agent wishes to move to another host: host H2. The host H2 runs the DESIRE platform, and its agents run code generated by the DESIRE platform.

In the process of migrating the agent A from host H1 to host H2, the agent first needs to offload information on its state. Then the agent factory on host H1 sends the blueprint of the agent, together with the state information of the agent to host H2.

Host H2's local agent factory receives the blueprint of the agent and state information. This agent factory designs a DESIRE agent A on the basis of the blueprint of agent A. This DESIRE agent A (i. e., a functionally equivalent incarnation of the Java agent A) runs on DESIRE's virtual machine (the DESIRE interpreter), and is able to incorporate information on its state.

The agent factory on the receiving side regenerates the agent, possible in a different implementation language and in a different environment.

5. Experimental Validation. The experimental setting consists of two agent platforms: JADE (version 3.01b) and FIPA-OS (version 2.10). The goal of our experiment is to connect the platforms to enable the exchange of messages and the migration of agents following the principles on generative migration (i. e., a blueprint and state are transferred).

5.1. Agent Platforms. Agent platforms provide an environment for the execution of agents. JADE [2] the Java Agent Development Environment, is a FIPA-compliant agent platform. The main objective of JADE is to simplify the development of agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. As a secondary objective the JADE agent platform tries to optimize the performance of a distributed agent system implemented in the Java language. JADE consists of two parts. The first part is the runtime environment for FIPA compliant multi-agent systems. The second is a framework for FIPA compliant agent system development. JADE supports weak agent migration using Java serialization.

FIPA-OS [32] is an open source agent platform implemented in the Java programming language. The main purpose of the FIPA-OS agent platform is to provide a reference implementation for the standard specifications of the FIPA organization. A key focus of the platform is that it supports openness through a loose coupling between the platform components and compliance to the FIPA Agent Management reference model. Official releases of FIPA-OS do not support agent migration.

5.2. Assumptions. Given the assumptions for Generative Migration, extra assumptions are necessary for the realization of Cross-Platform Generative Migration implementation:

- Appropriate communication facilities in the agent platforms exist: for example, facilities for sending ACL messages via HTTP are needed. A bidirectional channel for communication is needed for sending messages. The communication channel is used for requesting migration and sending/receiving information about the agents (e.g., blueprint and state documents).

TABLE 5.1
Test results.

	Message Structure	Communication Protocols	Creation Process
JADE	ACL	HTTP, CORBA ORB (IIOP)	simple, can be isolated
FIPA-OS	ACL	HTTP, CORBA ORB (IIOP)	currently part of the GUI, difficult to isolate

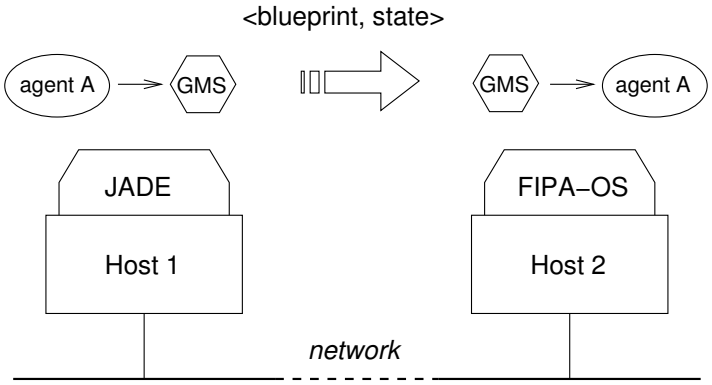


FIG. 5.1. *Generative migration experiment with JADE and FIPA-OS.*

- There is a service for agent creation or regulated control over the agent creation process. By means of an agent creation process a new agent can be created on a target platform. Control over the creation process allows for the implementation of a service that starts and initializes a new agent.

Agent creation, as such, was a challenge in FIPA-OS. The agent creation process is difficult to isolate in FIPA-OS: it is integrated in the User Interface classes and other classes. Implementation of a separate service for automatic agent creation and initialization is not currently supported. JADE does support separate services for this purpose.

5.3. Scenario. The scenario focuses on the technical aspects of generative agent migration. The environmental setting consists of two Agent Platforms: FIPA-OS and JADE. A number of services is assumed to be present on both platforms: a standard Directory Facilitator (DF) and a Generative Migration Service (GMS). The GMS is our own implementation and its primary task is to take care of migration, agent creation and initialization. The GMSs are registered at the local DF and can be found by agents and other services on both platforms if the DFs are cross-registered.

An agent sends a migration request to the local GMS for migration to a remote platform. The GMS contacts the remote GMS and forwards the request. The agent receives an acknowledgement of the migration request, or denial with an indication of the reason for denial (e.g., no GMS on target location, building-blocks not present on target or an agent with (requested) identical name exists). On acknowledgement the agent hands over its blueprint and state to the local GMS. The local GMS contacts the remote GMS for creation of a new agent with a given blueprint, state and agent name (see experiment scenario in Fig 5.1). The remote GMS needs to be able to assemble a new agent based on the blueprint of the requesting agent, it must have capabilities and permission to create a new agent on the platform and the means to initialize the new agent with the state of the original agent. Once a new agent is running the agent in the source platform can be stopped and removed.

5.4. The Implementation. As explained in Section 3, a compositional agent can be described in a blueprint document, written in an independent format that can be exchanged between locations. Additionally, state information can be used to initialize a newly generated agent on the target location.

To implement generative agent migration in agent platforms, two frameworks had been developed: a conceptual framework and an operational framework. The conceptual framework is based on the DESIRE modelling framework [16] and has four kinds of structure elements: components, links, information types, and control. Each component has an input and an output buffer and components define their input and output information types. Links can be placed between components to connect input and output buffers. A link transfers information elements (instances of information types) from an output buffer to an input buffer. Which information types a link transports can be predefined. In this framework only component structure elements can contain other structure elements.

On the operational level, an implementation-language specific framework is needed. For building-block components implemented in the Java programming language a framework had been developed which has a one-to-one mapping with the conceptual framework. The operational framework is used to dynamically load Java components via the provided mechanisms for class loading.

A compositional agent is embedded in a wrapper, the wrapper is an extension of the base-agent of an agent platform, e.g., in JADE it is `jade.core.Agent` and in FIPA-OS `fipaos.agent.FIPAOSAgent`.

6. Security Issues. Migration of an agent involves security from a number of perspectives. Security issues related to authenticating an agent, and deciding whether an agent is allowed to migrate to its destination, are not discussed in this paper. What remains is how to protect an agent against attacks during and after its migration, and how to protect a target against attacks from a malicious agent. Considerable research has already been conducted with respect to both issues which can be applied to our approach. In the following, the role of security is briefly considered. It should be noted, however, that security in our approach is subject of current research.

6.1. Protecting an Agent. A mobile agent may be preyed upon while it is in transit, or while running on a malicious host. It is impossible to protect an agent against modifications during its transfer or execution in an untrusted environment [12]. At best, it can check whether an agent has been maliciously modified and take appropriate measures after the fact. Our approach to migration can help here.

It is important to realize that, in principle, an agent's blueprint does not change during its lifetime. (Except for reconfiguration at an agent factory.) Consequently, by adding an integrity check to a blueprint using standard techniques for digital signatures [34], it is easy to detect whether a blueprint has been changed. When a factory notices that a blueprint has been changed, it can either discard the agent or generate it from the original blueprint. The latter is possible only if that blueprint is locally available, or if it can be retrieved in a secure way. Securely retrieving a blueprint requires that a factory can set up a secure channel to a blueprint repository, that is, a channel that provides authentication and transmission integrity.

Of course, it should be possible to support *evolutionary agents* for which new blueprints are generated. However, blueprint generation should be done only by trusted factories and never as a solution to migration. As such, it falls outside the scope of this paper.

6.2. Protecting a Host. A host that admits foreign mobile agents to its resources takes a risk: some of the agents may be malicious, and may try to subvert (parts of) the host. The problem with traditional approaches to agent migration is that it is impossible to check in advance whether or not imported code does only what it promises. The solution is to construct what are known as sandboxes [39]: a restricted environment in which, effectively, each instruction is monitored and checked before being executed. If access to resources is violated, execution halts. The sandbox model is quite restrictive, and has been extended since its initial introduction (see, for example, [14, 23]).

Regenerating agents from blueprints may considerably help to protect a host against malicious code. Normally, blueprints do not contain code descriptions, but refer only to interfaces and components that should be locally available to an agent factory. The code contained in these components may have been verified by the owner of the factory, or have been obtained from trusted sources. Of course, protection will fail if verification has not been done properly. In effect, a requirement is that trusted code is available before an agent migrates to a target, or that can be retrieved from a trusted repository through a secure channel.

In those cases that blueprints require execution of untrusted code, traditional approaches based on sandboxing techniques or protection domains need to be implemented as part of the target platform.

A mobile agent arriving at the host is regenerated on the basis of its blueprint, using only detailed building blocks of which the host approves. Although the specific configuration of building blocks may be new to the host, a number of security risks can be removed. The mobile agent may still be untrustworthy, but is prevented from executing certain calls on the host.

As an example, consider a bank that wishes to use mobile agents which may transact money from one account to another. The bank offers libraries of building blocks written in Java to its clients. These clients may build mobile agents that can perform transactions at the bank. The bank admits only those mobile agents that can be regenerated on the basis of their blueprint using building blocks written in their programming language of choice, running on their machines. Note that cheating, using other people's passwords and certificates is not necessarily stopped by this approach.

7. Related Research. Related research with respect to the agent factory and other heterogeneous agent migration initiatives is shortly discussed.

In this paper an Agent Factory is a service to design, adapt and (re-) assemble agents from building blocks. The Agent Factory to which this paper refers is the Agent Factory described in [6, 36]: the IIDS Agent Factory. There are other agent factories. The Factory of the Agents [8, 9] currently being developed to provide "a cohesive framework that supports a structured approach to the development and deployment of agent oriented-applications", providing extensive support for the creation of Belief-Desire-Intention (BDI) agents.

The Agent Factory described in [10, 11] is designed to provide designers with a tool for building agents, and a repository of patterns that can be used to add new capabilities. Agents in this project are developed according to the PASSI design methodology, with which models of multi-agent systems and agent interactions are specified and expressed in UML.

The IIDS Agent Factory has been designed for automated design and redesign of single agents, whereas the two other agent factories are being designed to assist human designers designing multi-agent systems. However, some similarities can be found in the approaches to agent design and the assembly process.

Also related to this research are other heterogeneous agent migration approaches. Known implementations are based on wrappers, intermediate interface layers, and agent servers. Especially worth mentioning are the Guest [21, 22] and Monads [24] research efforts.

The developers of Guest [21, 22] propose a middleware-based model that introduces an intermediate layer for the support of their agents on top of an existing agent platform. Regardless of the underlying platform, if it supports the Guest Layer, Guest Agents can be executed. The goals of the project are (i) to allow interoperability between platforms, i. e., allowing agents to run, communicate and move between them; (ii) To provide a uniform view of those platforms, i. e. agents can be written and manipulated without considering the kind of servers they will run on; (iii) To add new adaptive features to the platforms, i. e., plug-in mechanisms for dynamic extensions. Though still an experimental prototype, Guest enables interoperability between Voyager, Aglets, Grasshopper and JADE.

The Monads project [24] concentrates on the needs of nomadic users and adaptability. By adaptability they primarily mean the ways in which services adapt themselves to properties of terminal equipment and to characteristics of communications. This involves both mobile and intelligent agents as well as learning and predicting temporary changes in the available Quality-of-Service along the communications paths. The goal of Monads is to design an efficient and reliable software architecture based on adaptive services and agents, and to develop prototypes based on that architecture. The agents themselves are not adaptive, only used to steer changes in, e.g., quality of service.

The basic approach to mobile agents in Monads is a separation of an agent into a head and a body. The body handles the agent-programming interface of each agent platform, and a head can be placed on top of it. The agent-head can migrate via a service called the Monads Agent Gateway (MAG). The Monads approach has been implemented on JADE, Voyager and Grasshopper agent platforms.

There are some differences and similarities with our approach. The Guest Interface Layer defines a generic agent interface that can be supported on other platforms. This differs with the approach explained in this paper because our agents are assembled (i. e. adapted) before execution on any platform takes place and agents in Guest are not adapted. A similarity with Monads and our approach concerns one of the goals of Guest: to provide a uniform view of the underlying platform. In Monads an agent consists of two parts: a head and a body. In our approach a similar division into agent-head and agent-body can be made. In contrast with the Monads agent-head our agent consists of multiple components and can be migrated using generative migration whereas Monads does not make this distinction and uses Java object serialization for migration. Additionally the functionality of their MAG-service is comparable with our Generative Migration Service.

8. Discussion and Future Work. Agents, and in particular mobile agents, offer a means for application developers to build distributed applications. Mobility of agents is often required for various reasons, notably performance. Current agent platforms offer a wide range of services to agent developers, including mobility. However, mobility of agents is usually limited to hosts running the same agent platform and that have the same (virtual) machine architecture. In other words, it is often restrained to a homogeneous environment.

The approach described in this paper transcends this homogeneity and proposes *generative mobility*. In generative mobility, a blueprint of an agent's functionality is transported, together with information on the agent's state. At its destination, an agent factory regenerates the executable code of the agent on the basis of its blueprint. An agent may then restore its state and resume execution.

With generative mobility an agent may travel to locations that offer a different platform and that require it to adopt a different (virtual) machine architecture. In other words, generative mobility supports true heterogeneous mobility, offering an agent maximum flexibility with respect to where it wants to go. In addition, an agent's executable code can be

optimized for its destination, while retaining its required agent-level functionality. In their own way, agent factories and blueprints offer a language and agent platform independent virtual machine that allows for heterogeneous migration.

Agent factories play an important role in generative mobility as they offer the services needed to generate executable code on the basis of blueprints. Agent factories rely on libraries of building blocks from which agents can be configured. As a consequence, agent factories need to share these (conceptual) building blocks to understand an agent's blueprint and be able to generate its associated executable code. Homogeneity in agent architectures is a likely consequence of this approach.

Research on generative mobility is clearly not finished. In particular, the use of blueprints needs to be investigated to determine to what extent blueprints are flexible enough to describe agents, and how security can be adequately dealt with. Agent factories form an important component within our worldwide distributed AgentScape system that allows agents to be automatically (re-)designed. Currently a new prototype of the agent factory (namely the libraries of components) in AgentScape is being built that supports generative migration, based on the factory described in this article.

The use of generative mobility for relatively closed environments, such as hospitals, is currently being studied. Generative mobility with trusted code libraries on the hospital side may possibly provide a solution to controlled access to medical dossiers. Insurance companies, for example, are allowed limited access to specific types of information and processing. Control over the executable code of an insurance company's agent provides a means for a hospital to control the calls and data an agent may execute inside the hospital.

REFERENCES

[1] C. BÄUMER, M. BREUGST, S. CHOY, AND T. MAGEDANZ, *Grasshopper—A universal agent platform based on OMG MASIF and FIPA standards*, in First International Workshop on Mobile Agents for Telecommunication Applications (MATA'99), Ottawa, Canada, Oct. 1999, pp. 1–18.

[2] F. BELLIFEMINE, A. POGGI, AND G. RIMASSA, *Developing multi-agent systems with a FIPA-compliant agent framework*, Software: Practice and Experience, 31 (2001), pp. 103–128.

[3] F. BRAZIER, C. JONKER, AND J. TREUR, *Compositional design and reuse of a generic agent model*, Applied Artificial Intelligence Journal, 14 (2000), pp. 491–538.

[4] ———, *Principles of component-based design of intelligent agents*, Data and Knowledge Engineering, 41 (2002), pp. 1–28.

[5] F. BRAZIER AND N. WIJNGAARDS, *Automated servicing of agents*, in Proceedings of the AISB-01 Symposium on Adaptive Agents and Multi-Agent Systems, Mar. 2001, pp. 54–64.

[6] ———, *Designing self-modifying agents*, in Proceedings of Computational and Cognitive Models of Creative Design, the Fifth International Roundtable Conference, Dec. 2001, pp. 93–112.

[7] A. CHAKRAVARTI, G. BAUMGARTNER, AND M. LAURIA, *The organic grid: Self-organizing computation on a peer-to-peer network*, in Proceedings of the First International Conference on Autonomic Computing (ICAC 2004), New York, NY, May 2004, pp. 96–103.

[8] R. COLLIER AND G. O'HARE, *Agent factory: A revised agent prototyping environment*, in Proceedings of the 10th Irish Artificial Intelligence and Cognative Science Conference (AICS'99), Cork, Ireland, Sept. 1999.

[9] R. COLLIER, G. O'HARE, T. LOWEN, AND C. ROONEY, *Beyond prototyping in the factory of the agents*, in Proceeding of the 3rd Central and Easern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic, June 2003.

[10] M. COSSENTINO, *Different perspectives in designing multi-agent systems*, in Proceedings of the AGES'02 Workshop at NODe02, Erfurt, Germany, Oct. 2002.

[11] M. COSSENTINO, P. BURRAFATO, S. LOMBARDO, AND L. SABATUCCI, *Introducing pattern reuse in the design of multi-agent systems*, in Proceedings of the AITA'02 workshop at NODe02, Erfurt, Germany, Oct. 2002.

[12] W. FARMER, J. GUTTMAN, AND V. SWARUP, *Security for mobile agents: Issues and requirements*, in Proceedings of the 19th National Information Systems Security Conference, Baltimore, MD, Oct. 1996, pp. 591–597.

[13] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

[14] L. GONG AND R. SCHEMERS, *Implementing protection domains in the Java Development Kit 1.2*, in Proceedings of the Symposium on Network and Distributed System Security, San Diego, CA, Mar. 1998, pp. 125–134.

[15] R. GRAY, G. CYBENKO, D. KOTZ, R. PETERSON, AND D. RUS, *D'Agents: Applications and performance of a mobile-agent system*, Software: Practice and Experience, (2001). In press.

[16] K. ISKRA, F. VAN DER LINDEN, Z. HENDRIKSE, B. OVEREINDER, G. VAN ALBADA, AND P. SLOOT, *The implementation of Dynamite: An environment for migrating PVM tasks*, Operating Systems Review, 34 (2000), pp. 40–55.

[17] D. JOHANSEN, R. VAN RENESSE, AND F. SCHNEIDER, *Operating system support for mobile agents*, in Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), Orcas Island, WA, May 1995, pp. 42–45.

[18] D. LANGE, M. OSHIMA, G. KARJOTH, AND K. KOSAKA, *Aglets: Programming mobile agents in Java*, in Worldwide Computing and Its Applications, vol. 1274 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1997, pp. 253–266.

[19] D. B. LANGE AND M. OSHIMA, *Seven good reasons for mobile agents*, Communications of the ACM, 42 (1999), pp. 88–89.

[20] M. LUCK, P. MCBURNEY, O. SHEHORY, AND S. WILLMOTT, *Agent technology: Computing as interaction—A roadmap for agent-based computing*, tech. rep., AgentLink III, Sept. 2005. `http://www.agentlink.org/roadmap/`

[21] L. MAGNIN, T. V. PHAM, A. DURY, N. BESSON, AND A. THIEFAINE, *Our guest agents are welcome to your agent platforms*, in Proceedings of the ACM Symposium on Applied Computing (SAC 2002), Madrid, Spain, Mar. 2002, pp. 107–114.

[22] L. MAGNIN, H. SNOUSSI, V. T. PHAM, A. DURY, AND J.-Y. NIE, *Agents need to become welcome*, in Proceedings of the 3rd International Symposium on Multi-Agent Systems, Large Complex Systems, and E-Businesses (MALCEB'2002), Erfurt, Germany, Oct. 2002.

[23] D. MALKHI AND M. REITER, *Secure execution of Java applets using a remote playground*, IEEE Transactions on Software Engineering, 26 (2000), pp. 1197–1209.
[24] P. MISIKANGAS AND K. RAATIKAINEN, *Agent migration between incompatible platform*, in Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, Republic of China, Apr. 2000.
[25] H. NWANA, D. NDUMU, L. LEE, AND J. COLLIS, *ZEUS: A tool-kit for building distributed multi-agent systems*, Applied Artifical Intelligence Journal, 13 (1999), pp. 129–186.
[26] OMG, *Mobile agent facility specification*, OMG Document formal/00-01-02, Object Management Group, Framingham, MA, Jan. 2000.
[27] B. OVEREINDER AND F. BRAZIER, *Scalable middleware environment for agent-based Internet applications*, in Proceedings of the Workshop on State-of-the-Art in Scientific Computing (PARA'04), Copenhagen, Denmark, June 2004, pp. 675–679. Published in Applied Parallel Computing, LNCS 3732, Springer, Berlin, 2006.
[28] B. OVEREINDER, P. SLOOT, R. HEEDERIK, AND L. HERTZBERGER, *A dynamic load balancing system for parallel cluster computing*, Future Generation Computer Systems, 12 (1996), pp. 101–115.
[29] H. PEINE AND T. STOLPMANN, *The architecture of the Ara platform for mobile agents*, in Proceedings of the First International Workshop on Mobile Agents (MA'97), vol. 1219 of Lecture Notes in Computer Science, Berlin, Germany, Apr. 1997, Springer-Verlag, pp. 50–61.
[30] F. PEÑA-MORA AND S. VADHAVKAR, *Design rationale and design patterns in reusable software design*, in Artificial Intelligence in Design (AID'96), Dordrecht, 1996, Kluwer Academic Publishers, pp. 251–268.
[31] G. P. PICCO, *Mobile agents: An introduction*, Microprocessors and Microsystems, 25 (2001), pp. 65–74.
[32] S. POSLAD, P. BUCKLE, AND R. HADINGHAM, *The FIPA-OS agent platform: Open source for open standards*, in Fifth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, Manchester, UK, Apr. 2000, pp. 355–368.
[33] A. RIEL, *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996.
[34] B. SCHNEIER, *Applied Cryptography*, John Wiley, New York, NY, 2nd ed., 1996.
[35] G. SCHREIBER, H. AKKERMANS, A. ANJEWIERDEN, R. DE HOOG, N. SHADBOLT, W. V. DE VELDE, AND B. WIELINGA, *Knowledge Engineering and Management, the CommonKADS Methodology*, MIT Press, 1999.
[36] S. V. SPLUNTER, M. SABOU, F. BRAZIER, AND D. RICHARDS, *Configuring web service, using structurings and techniques from agent configuration*, in Proceedings of the 2003 IEEE/WIC International Conference on Web Intelligence (WI 2003), Halifax, Canada, Oct. 2003, pp. 153–160.
[37] N. SURI, J. BRADSHAW, M. BREEDY, P. GROTH, G. HILL, AND R. JEFFERS, *Strong mobility and fine-grained resource control in NOMADS*, in Proceedings of the Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA 2000), Zurich, Switzerland, Sept. 2000, pp. 2–15.
[38] A. TRIPATHI, N. KARNIK, M. VORA, T. AHMED, AND R. SINGH, *Mobile agent programming in Ajanta*, in Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, May 1999, pp. 190–197.
[39] D. WALLACH, D. BALFANZ, D. DEAN, AND E. FELTEN, *Extensible security architectures for Java*, in Proceedings of the 16th Symposium on Operating System Principles, St. Malo, France, Oct. 1997, ACM, pp. 116–128.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 16, 2006

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 101–117. http://www.scpe.org
ISSN 1895-1767

A DISTRIBUTED CONTENT-BASED SEARCH ENGINE BASED ON MOBILE CODE AND WEB SERVICE TECHNOLOGY

VOLKER ROTH[†] , JAN PETERS[‡] , AND ULRICH PINSDORF[§]

Abstract. Current search engines crawl the Web, download content, and digest this content locally. For multimedia content, this involves considerable volumes of data. Furthermore, this process covers only publicly available content because content providers are concerned that they otherwise loose control over the distribution of their intellectual property. We present the prototype of our secure and distributed search engine, which dynamically pushes content based feature extraction to image providers. Thereby, the volume of data that is transported over the network is significantly reduced, and the concerns mentioned above are alleviated. The distribution of feature extraction and matching algorithms is done by mobile software agents. Subsequent search requests performed upon the resulting feature indices by means of remote feature comparison can either be realized through mobile software agents, or by the use of implicitly created Web services which wrap the remote comparison functionality, and thereby improve the interoperability of the search engine. We give a description of the search engine's architecture and implementation, depict our concepts to integrate agent and Web service technology, and present quantitative evaluation results. Furthermore, we discuss related security mechanisms for content protection and server security.

Key words. distributed systems, content based retrieval, mobile agents, Web services, content security

1. Introduction. The availability of vast amounts of multimedia contents in the Internet requires sophisticated means for searching and retrieval. Current search engines are generally based on a centralized gatherer which traverses the hyperlinks of the World Wide Web starting from known entry points, and which retrieves and digests all relevant data found. This approach has two disadvantages: (a) it is data intensive, and (b) search engines cover only contents which are freely available for download. One might argue that transfer volume is not an issue because ample bandwidth is available on the Internet backbones. However, edge networks generally pay considerable penalties if they exceed their transfer volume quotas. They have an interest not to exceed their quota and to keep it as low as possible. Search engines should be designed to honor this desire. The second disadvantage results from the fact that commercial content providers loose control over the distribution of their intellectual property, once the search engine downloads it. Consequently, providers offer local searches and the number of algorithms they provide is likely limited.

In this paper we report on our distributed search engine prototype, which pushes content-based feature extraction (and optionally feature comparison) to the edge networks, and which can alleviate the aforementioned intellectual property concerns. The search engine is based on *mobile software agents* (see *e.g.* [52, 53] for an introduction to mobile agents). The benefits of mobile agent based feature extraction are:

- Multiple image sources can be processed in parallel. Each image source contributes to the processing power required to extract salient image features of its images.
- Multiple (*e.g.*, composable) feature extraction and comparison algorithms can be deployed concurrently and easily.
- Feature vectors are generally more compact than images, therefore less data must be transported from image sources to the gatherer.
- Feature extraction takes place at the image source. The images must not be exported from it, and original images generally cannot be reproduced from the feature vectors.
- The search algorithm can be provided by the searching client instead of the content providers. This leads to a decoupling of content and algorithm providers.

The achievable amount of parallelization and the reduced data transfer volumes have a significant positive impact on completion time of the feature extraction process. Disadvantages of the mobile agent based search engine are:

- Image sources have to set aside computing resources for feature extracting agents.
- Content providers have to host a middleware for mobile agents and give them access to their image content.
- The search engine's client has to provide a mobile agent middleware to distribute feature extraction resp. comparison algorithms, and to send further query agents for search requests.

[†]FX Palo Alto Laboratory, USA, volker.roth@acm.org
[‡]Fraunhofer IGD, Germany, jan.peters@igd.fraunhofer.de
[§]Fraunhofer IGD, Germany, ulrich.pinsdorf@igd.fraunhofer.de

- Running mobile code on a server poses a considerable security risk. Therefore, security of the mobile agent middleware is an essential requirement for the practicality of the approach.

In case, the feature extraction process and the distribution of corresponding feature comparison algorithms to image sources is realized by means of mobile agents in an initial index phase, following search queries need not necessarily rely on the same interaction paradigm. As described in §4, recent research results allow us to implicitly and transparently describe resp. provide access to distributed algorithms by means of the Web service protocol suite. According to the application scenario (confer §2), this fact relativizes some of the above mentioned disadvantages:

- Presuming that distribution of feature extraction and comparison algorithms is done by a small group of experts using mobile code, subsequent provision of these specialized algorithms can be done for a broad group of users using Web service compliant client applications.
- Further assuming that the level of trust between image providers and algorithm providers is higher than the level of trust between image providers and end users of the search engine, image providers can clearly distinguish the security policies and mechanisms bound to the respective interaction paradigm (mobile code vs. Web service technology).

We built our search engine prototype on the mobile agent server *SeMoA* [41]. Although SeMoA supports a rich set of security features, we do not claim that its security is perfect—the fact that it is programmed in Java alone renders it vulnerable to a variety of *Denial of Service* (DoS) attacks [7]. However, SeMoA provides a rich set of cryptographic features to protect the data of agents (*e.g.*, collected images) against disclosure on untrusted hosts, and an architecture that emphasizes separation of agents.

The basic concepts of mobile agent based image search engines have been mentioned by several authors before [6, 2, 39, 54] but have not yet been addressed in sufficient detail and in the context of a practical system. In this paper, we contribute a nuanced discussion and comparison of operation modes, a description of our implementation, and a quantitative analysis of the benefits of our search engine.

2. Concepts and Models. There does not seem to be a universal understanding when a program crosses the border to agent-hood. Software agents are often defined as being *reactive*, *autonomous*, *goal-oriented*, and *continuous* [18] though further attributes exist. *Mobile* agents have the ability to relocate—at some point of their execution they can halt and initiate a *migration* to some distant host where they resume execution. During migration, an agent's program as well as its current execution state and accompanying data is transported to its new host. When and where an agent migrates is part of the agent's program. In general, mobile agents rely on an infrastructure of mobile agent servers which handle agent transport, setup and deinstallation. What mobile agent technology brings to bear on the problem of image indexing and retrieval is easy means of *software distribution*. Briefly, mobile agents provide a flexible and easy mechanism to transport content-based feature extraction and matching algorithms to the source of the images rather than vice versa. The impact on network utilization and scalability if profound—instead of putting the burden of gathering contents completely onto the shoulders of a centralized gatherer and its connected network interface the load is shared among the gatherer and the image servers and all image servers can be indexed in parallel.

The W3C defines a Web service architecture [51] as a software system designed to support interoperable machine-to-machine interaction over a network. Thereby, a Web service is an abstract notation which is implemented as a computational resource. These Web service compliant modular software components are described, published and invoked by means of a few core specifications, namely *Web Service Description Language (WSDL)* [11], *Universal Description, Discovery and Integration (UDDI)* [32], and *Simple Object Access Protocol (SOAP)* [50]. Based on these specifications a variety of new protocols and standards have been evolved, addressing workflow definition and service federation, publish-subcribed messaging and transactions, as well as reliable messaging and security. In turn, this Web service protocol stack provides the foundation for current distributed service-oriented architectures, already impacting a broad range of commerce and industry. Integrating mobile agent and Web service technology, as described in details in §4, results in a middleware of the image provider which is able to accept client-specific feature extraction and matching algorithms as mobile code components, and in turn, provides this functionality to a broad range of Web service compliant applications.

Below, we illustrate two slightly varying models of image search engines based on mobile agent and content-based retrieval technology. The first model resembles an optimized gatherer which still has a central repository of feature vectors (though feature extraction is done remotely). We refer to this model as the *gatherer model*. The second model keeps a distributed index and no data needs to be shipped over the network during indexing. We refer to this model as the *incubator*

model[1]. Furthermore, we present an *extended incubator model* comprising Web service technology to provide the actual search functionality to a broad group of end users. The general search is always threefold. First, a number of index agents sent by the broker are responsible for feature extraction at the content servers (*index phase*). Second, a search agent is used to find images which match a user defined query based on the pre-calculated feature vectors (*search phase*). Finally, a fetch agent collects the desired images from the according image servers (*fetch phase*). Alternatively, the last two steps can be replaced by Web service interactions.

Below, we describe and discuss the three models. In §3 and §4, we explain in greater detail to what extent and how we have implemented the models in our prototype.

2.1. The Gatherer Model. The gatherer model consists of a central image broker, several image servers, *index agents*, *search agents*, and *fetch agents* (all agents are mobile and relocate during their life cycle). The image broker dispatches index agents which transport feature extraction algorithms to one or more image servers. On these servers, the index agents extract relevant feature vectors from local images and send or take image entries back to the image broker. At the image broker, all image entries are merged into the central index (see Fig. 2.1 for illustration). Each image entry consists of a feature vector, the URL pointing to the host where the image was retrieved, an image ID that uniquely identifies the image at the image server, an optional thumbnail, and optional further information on the image such as its size or licensing information. The globally unique ID of an image consists of the globally unique URL plus the locally unique image ID.

Based on the index data, image brokers can either serve requests in a client/server fashion, or they support mobile agent queries as follows. A client sends a search agent to the broker which queries for similar images by means of an example image, a sketch, a prototypical image, or a feature vector which is extracted from either of these query images. The query result consists of extended image entries which contain the normalized distance between the query and the entry's feature vector in addition to the entry itself. The search agent transports the result set back to the client who selects images for retrieval based on the included thumbnails, or refines the query (*e.g.*, by means of relevance feedback). Once an image is selected for retrieval, the client sends a fetch agent that migrates to the server on which the image is stored (directed by the URL which is stored in the image entry) and retrieves the image based on the local image ID also contained in the entry (see Fig. 2.2). This can be preceded by a negotiation phase in which agent and image provider agree *e.g.*, on licensing terms and the payment of license fees.

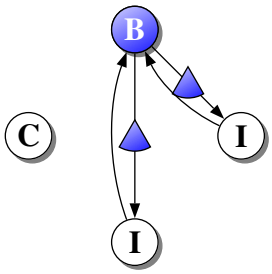


FIG. 2.1. *Gatherer model (index phase): The broker (B) dispatches feature extraction agents (denoted as triangles) to the image sources (I). Once the agents completed extracting the features of all images, they carry the feature vectors back to the broker.*

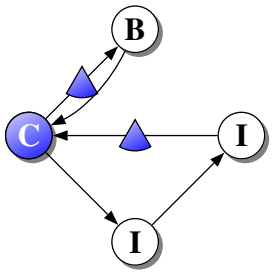


FIG. 2.2. *Gatherer model (search/fetch phase): The client (C) sends a search agent to the broker (B), which retrieves image entries matching the client's query, and transports the entries back to the client. The client selects images for retrieval, which are subsequently collected from the image sources (I) by the fetch agent.*

The transfer volume savings of this model are proportional to the compression factor of the feature extraction algorithm. On the one hand, a constant overhead incurs because the feature extraction algorithm must be transported to each image server. On the other hand, only one network connection request is required for transporting the agent—compared to one connection per image in the case of ordinary gatherers (unless the gatherer and the image server support sessions). Content providers retain control over their intellectual property because only feature vectors are exported, from which the original image generally cannot be reproduced in high quality.

2.2. The Incubator Model. The incubator model can do even better than the optimized gatherer model. In the incubator model, one index agent per image server is dispatched and takes residence at the image server. There, it extracts features as previously described but sets up an index directly at the image server (see Fig. 2.3). The index agent may

[1] Incubator: "A place or situation that permits or encourages the formation and development, as of new ideas." The American Heritage Dictionary of the English Language, Fourth Edition

also monitor the local image repository for changes, and it can update its index accordingly and incrementally. The only communication between the broker and the index agent is a short notice that the computation of the index is completed and the index agent is ready to provide service to search agents. The image broker is still a central point of access but it resembles more a yellow pages server. It refers search agents to the image servers where index agents reside (see Fig. 2.4). Once the client selects an image for retrieval, the process continues as in the gatherer model.

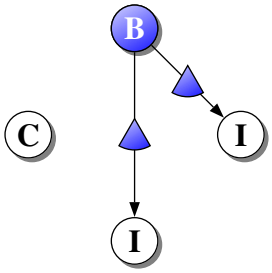


FIG. 2.3. *Incubator model (index phase): The broker (B) dispatches feature extraction agents (denoted as triangles) to the image sources (I). Once the agents completed extracting the features of all images, they register a feature comparison service at the image sources.*

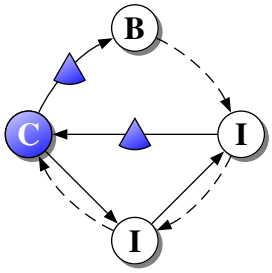


FIG. 2.4. *Incubator model (search/fetch phase): The client (C) sends a search agent to the broker (B), which retrieves a list of image sources, and searches them in turn (dashed lines). The client selects images for retrieval, which are subsequently collected from the image sources (I) by the fetch agent.*

Based on its index the index agent serves queries of search agents which visit the image server. On each image server, the search agent merges previously collected results with the results of its local search, and prunes the overall number of results *e.g.*, to a user-defined maximum number of n images with the lowest overall distance metric (unless security considerations take precedence, see also §3.2). The distance metric must be normalized so that the pruning is accurate. Multiple search agents can be dispatched in parallel to speed up the search. In such a case, the individual search results must be merged and pruned at a suitable host *e.g.* a server provided by the broker or the client itself. Moreover a broker may incubate an image server with different index agents, each one resembling a different matching algorithm.

Additionally, brokers may launch search agents on behalf of a client *e.g.*, if clients do not use mobile agents directly but rather access the broker through a regular Web interface. In this model the optimization is independent from the overall size of the content and the compression factor of the feature extraction algorithm. Only the matching images are transferred over the network.

2.3. The Extended Incubator Model. In the extended incubator model, the index phase is clearly separated from the search resp. fetch phase by means of the interaction paradigm used between the involved entities. As in the basic incubator model described above, the image broker dispatches one index agent per image server which takes residence at the image server. Again, the index agent extracts features and sets up a local index directly at the image server which is incementally updated, if the local image repository changes. In contrast to the basic incubator model, the search service provided by the index agent is automatically wrapped by a corresponding Web service stub. Furthermore, the corresponding Web service description (WSDL) is provided and the Web service is registered at a Web service compliant directory service hosted by the image broker (see Fig. 2.5).

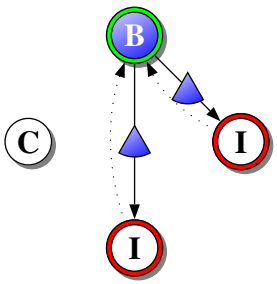


FIG. 2.5. *Extended Incubator model (index phase): The broker (B) dispatches feature extraction agents (denoted as triangles) to the image sources (I). Once the agents completed extracting the features of all images, they register a feature comparison service at the image sources which is automatically provided as Web service (visualized by red rings) and registered (dotted line) at the broker's Web service compliant directory service (visualized by green ring).*

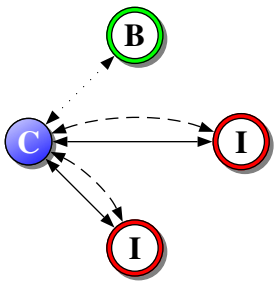


FIG. 2.6. *Extended Incubator model (search/fetch phase): First, the client (C) queries (dotted line) the Web service compliant directory service of the broker (B) which returns the locations of available comparison and fetch Web service instances. Subsequently, the client searches for images requesting (solid line) the search Web services at the image providers (I), and finally downloads the desired ones using the corresponding fetch Web services (dashed line).*

As in the basic model, the image broker is a central point of access which resembles a kind of yellow pages server. But this yellow pages server is realized as a Web service directory instead of a local service triggered by incoming search agents. Thus, the client interacts with the broker by means of the Web service discovery protocol UDDI within the search phase. Subsequently, it directly invokes the search Web service by means of SOAP requests, transmitting the image as search pattern and receiving the image IDs of comparable images—according to the selected comparison algorithm provided by the image broker—together with their thumbnails (see Fig. 2.6). Finally, the client requests the selected images from the corresponding image servers, again using the appropriate (fetch) Web services provided by these servers.

Instead of transmitting a reference image as search pattern to each image provider during the search phase, the broker could provide an appropriate feature extraction Web service which transforms received images into smaller feature vectors. Subsequently, the client would directly use this feature vector as search pattern for search Web service requests to lower the needed network bandwidth.

2.4. Comparison of Concepts and Benefits. The advantage of the gatherer and the incubator model over traditional centralized image search engines is that processor and memory consumption is shared between the image providers whose contents are processed, and the broker. Network utilization is considerably lower the mobile agent based models than in the traditional approach, and the feature extraction process completes considerably faster as a consequence of parallelization. None of the models export image contents to third parties.

The gatherer model is still somewhat centralized. Queries are answered by the broker. If the number of images is huge then the feature collection is likewise huge. Hence, while the gatherer model improves the process of index compilation it does not significantly improve the query process. Finally, if the broker fails then the entire service becomes unavailable.

Searching is less efficient in both incubator models than it is in the gatherer model because all image servers must be visited resp. queried by the search agent resp. the Web service compliant search client in turn before the query results are shown to the user. Certainly this can be alleviated by sending multiple search agents resp. triggering multiple search queries in parallel. However, the yellow pages maintained by the broker are much smaller than a full index of feature vectors, and they change less often then an index.[2] Therefore, replication (*e.g.,* by caching or by fail-over servers) can be implemented easier and more efficiently than this would be possible in the gatherer model.

The incubator models are particularly useful if image providers (who offer a broad range of images) team up with image brokers (who distribute specialized retrieval algorithms). Hence, the image broker may act as a well-known *portal site* with a focused marketing that addresses a specific target audience. The relationship between providers and brokers can be many-to-many, and their business relationships can be fluent and flexible. The advantage is that both parties can concentrate on their core competencies. Image provider specializes on content provisioning, and the image broker specializes on retrieval technology and image matching algorithms. In this regard our approach differs from *e.g.* meta search engines, which combine the results of regular search engines that in turn download contents. In our approach, however, search functionality is pushed to the content.

All the models (the gatherer model and the incubator models) can support multiple content-based retrieval mechanisms in parallel as well as retrieval mechanisms based on annotated information such as the name of photographer or painter, the year of production, or the price of licensing the image for specific uses. For instance, image servers can accept more than one index agent at the same time, and search agents resp. the Web service client can compute the intersection of multiple distinct result sets based on the global or local ID of each image entry.

In the extended incubator model, the used interaction paradigms are clearly separated according to the respective processing phases of the search engine. Thus, this model is specifically avantageous to support thin or legacy search clients which are not able to run the mobile agent middleware resp. to connect such a middleware with the actual client application: During the index phase, the image brokers still make use of the mobile agent paradigm and the resulting benefits—since image brokers and image providers are tightly bound to each other (*e.g.,* according to a business case), it can be assumed that they share the same mobile agent middleware. Providing search and fetch services as Web services by the image providers and a corresponding Web service directory by the image broker, the client application can easily be connected to the distributed search engine using one of the numerous Web service compliant frameworks available for arbitrary devices, operating systems, and corresponding programming languages.

Nevertheless, this model has some disadvantages during the search and fetch phase, which are specific to the client/server architecture as implemented by the described Web service infrastructure resp. specific to the Web service protocol suite. Depending on the underlying transport protocol (*e.g.,* HTTP), Web service requests are processed synchronously where required. Hence, the connection between client and image providers has to remain established while the search algorithm compares given images resp. feature vectors with the local index on the image server. In contrast, the mobile

[2] Image providers regularly add content but if a server is added or removed from the system then content is added or removed as well.

agent paradigm is much more flexible, *e.g.*, in scenarios with mobile devices. Furthermore, using the XML-based Web service protocols blows up the transported data and decreases the overall performance again, because of the necessary XML-based data type encoding/decoding.

As consequence, the user should carefully select or combine the appropriate search engine models according to the specific application scenario, considering the respective advantages and disadvantages.

3. Basic Architecture. The gatherer model and both incubator models are built on top of the SeMoA mobile code middleware [41]. Agents are received by network transport daemons, and they are injected into a pipeline of filters (see Fig. 3.1) which perform various security services and security checks on incoming agents (see §3.2 for more details). If an agent is admitted to the server then the agent may publish or retrieve service interface objects subject to access control restrictions. Upon termination, the agent is again processed by a filter pipeline and is subsequently migrated to its next hop.

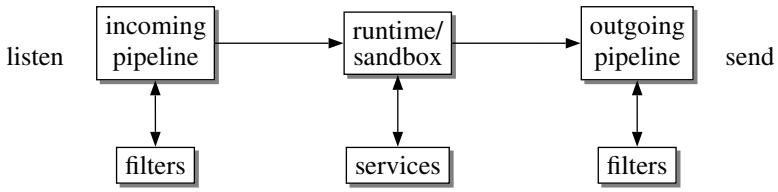


FIG. 3.1. *The middleware runs a daemon which listens for incoming agents. Each agent is piped through several filters before it is admitted to the runtime system where it can access services by name. Agents can register and retrieve services by name (subject to access control). Before migration, each agent is piped through outgoing filters again.*

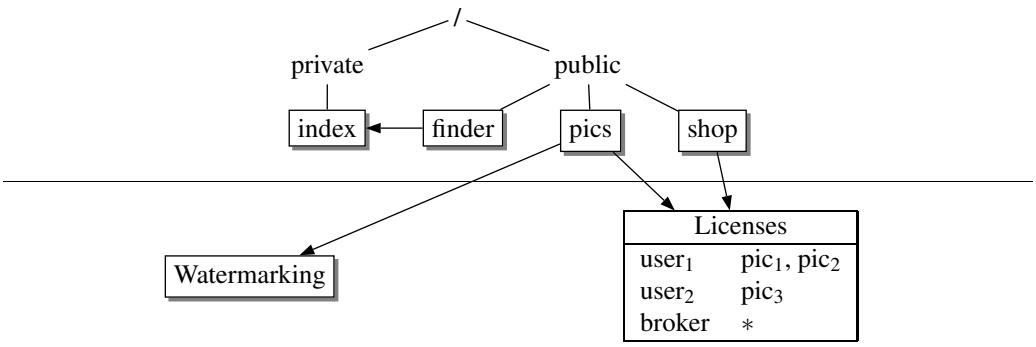


FIG. 3.2. *Agents can publish and retrieve services in a hierarchical name space. For instance, an image provider publishes the* pics *service under the path "/public/pics". The* pics *service iterates image names and thumbnails without restriction, but retrieves full quality images only if it is invoked by an agent whose owner has purchased a license. Agents can negotiate and purchase licenses on behalf of their owners by the* shop *service.*

Services are published in a hierarchical name space (similar to a hierarchical file system), which simplifies the grouping of services and the definition of access control policies. Agents may publish services dynamically at runtime in an allowed subspace of the name space, and the server may publish services or launch daemons statically at boot time. For instance, an image broker provides a static *index* service that his agents (and only his agents) can access in order to merge collected feature vectors with previously collected ones. In our implementation, the *index* service is backed by a file system and provides concurrent read/write access to the stored information. The image broker also publishes a static *finder* service which, on input of a query, returns matching image entries. This service is backed by the *index* service (as illustrated by the horizontal arrow in Fig. 3.2) but restricts access to the index to a limited set of operations and can therefore be made accessible to search agents (by placing it in the public area of the name space). In the incubator model, an index agent publishes the *finder* service dynamically at the image provider, and it keeps a private (unpublished) *index* service as the back end of its *finder* service. In that case, the image entries are stored by which ever resource backs the storage of the index agent[3]. Figure 3.3 gives a simplified view of the interface and class design in the UML [17] notation.

Image providers publish the static *pics* and *shop* services. The *pics* service iterates image IDs (*e.g.*, a locally unique image name) and thumbnails without restriction, but retrieves full quality images (based on the image ID) only if the

[3]Typically a file system or RAM of the host computer, the middleware provides abstractions for the actual type of agent storage which could also be backed by a database.

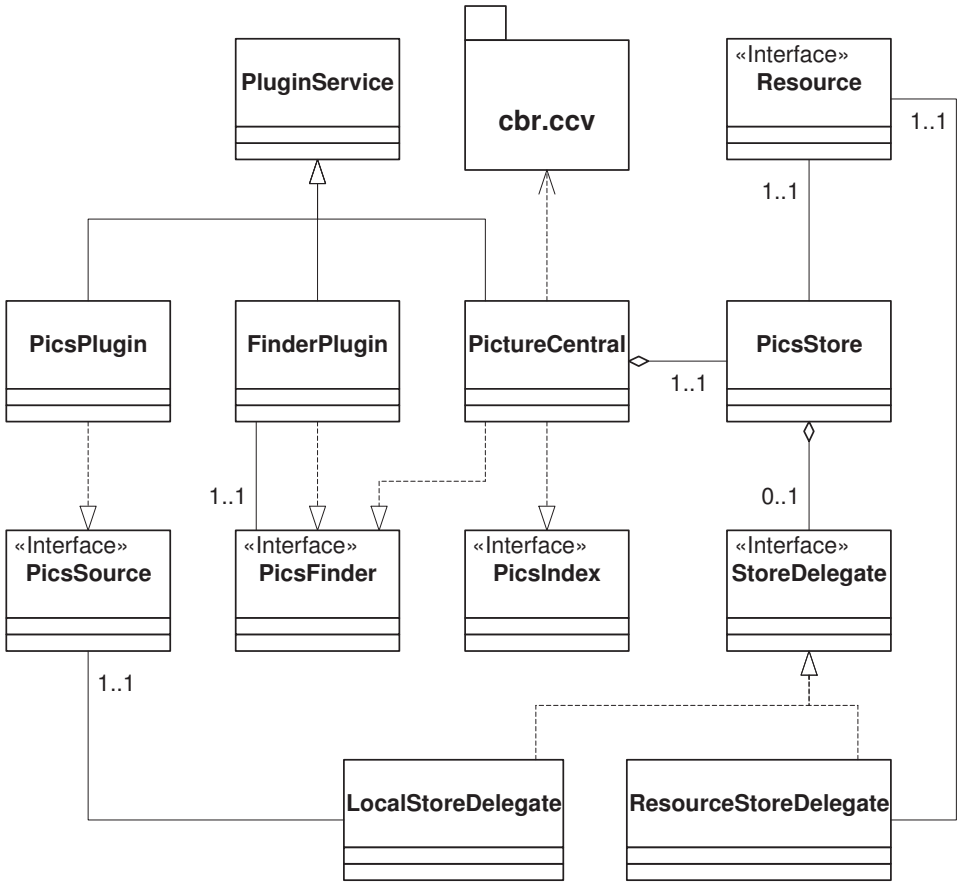


FIG. 3.3. *A simplified view of the interface and class design related to the CBR services is given above. Storage of information is handled through the "Store" abstraction for which we implementated instances that map to RAM or file systems.*

owner of the invoking agent already purchased a license for the image in question. In this case, the *pics* service may also embed the client ID as a digital watermark[4] into the retrieved image so that unauthorized usage of copyrighted material can be traced. Clients (resp. their agents) can negotiate and purchase licenses by the *shop* service[5]. Brokers can purchase a license for all images for the purpose of indexing (presumably at a low price and under the legally important condition that no full quality images are illicitly exported). Alternatively, image providers can grant brokers access to their images based on prior offline agreement.

The *pics* service provides a simple and sufficient interface so that feature extraction algorithms can iterate and extract features from existing images, irrespective of the heterogeneity of deployed image databases (*e.g.*, the schema of the database or the fact that images are simply stored in a file system).

3.1. Implementation Details. The prototype implementation uses *Color Coherence Vectors* [36] as feature extraction and comparison algorithm. Feature vectors consist of 128 float values; each vector is computed as follows: the image is blurred using a simple 3×3 convolution filter which averages the color values of all horizontal and vertical neighbors of the filtered pixel. The blurred image is then quantized to a color space of 64 colors. In the last step, the pixels of the image are classified into coherent and incoherent pixels. Coherent pixels are pixels which are part of a horizontally and vertically connected pixel area of the same color whose size exceeds a certain threshold τ (a fixed percentage of the total image area). Incoherent pixels are pixels which are not coherent pixels. For each of the 64 colors, the coherent and incoherent pixel counts are summed up separately and normalized with regard to the total image area. This results in

[4]The term "digital watermarking" refers to steganographic means of embedding copyright markers in multimedia data so that the marking is imperceptible, undetachable, as well as robust against a variety of adverse and inadvertent manipulations of the media such as lossy compression, format conversion, *et cetera.* See *e.g.*, [1] for an overview over digital watermarking.

[5]Together with Siemens Corporation we engineered and evaluated a licensing system for images retrieved by mobile agents. The system founds on Kerberos tickets and was used with both retrieving mechanisms described above [16].

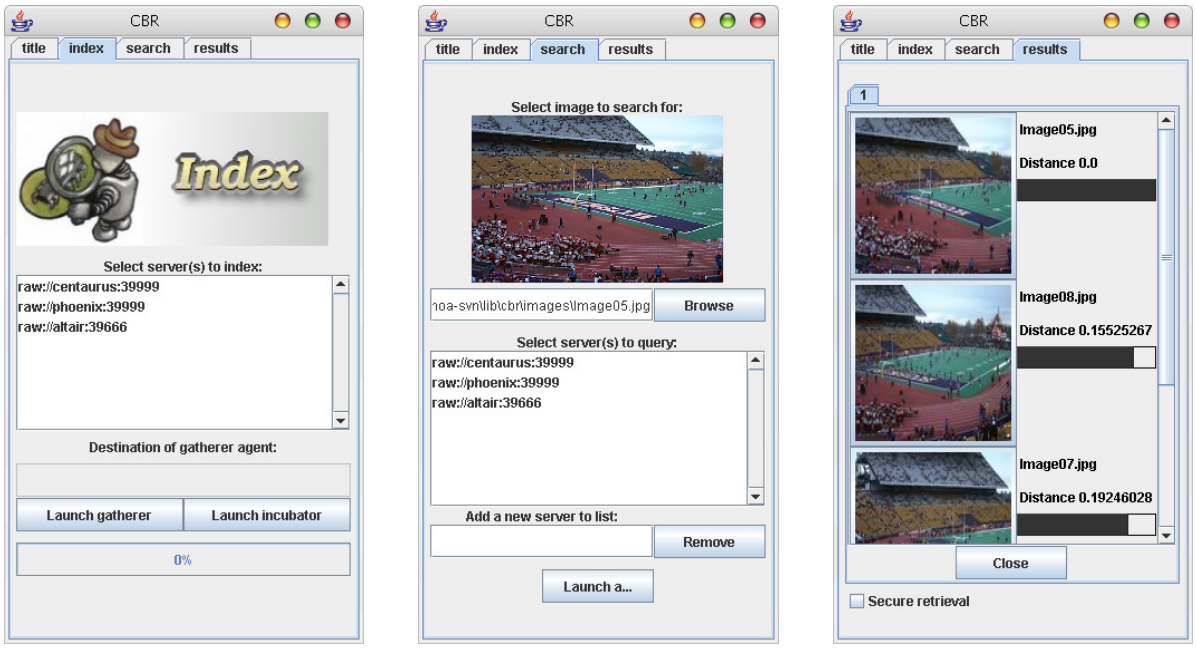


FIG. 3.4. *Three shots of the prototype GUI. The panel used to launch index agents is left, in the middle is the panel used to start search agents for a given example query image, and the panel which shows the query results after the search agent returned is right.*

a 128 dimensional vector. The L_1 distance is taken as a measure of similarity between two color coherence vectors. Let $\langle(h_1,\bar{h}_1),\ldots,(h_n,\bar{h}_n)\rangle$ be a color coherence vector where h_i is the percentage of coherent pixels of color i and $\bar{h}_i$ is the percentage of incoherent pixels of color i then the L_1 distance is defined as: $\|h-h'\| = \sum_{i=1}^{n}(\mid h_i - h'_i \mid + \mid \bar{h}_i - \bar{h}'_i \mid)$. The Color Coherence Vector algorithm has the advantage that it is easy to implement, reasonably fast, and achieves a high compression rate.

Images are reduced to a vector whose encoding is less than 600 bytes. For feature extraction algorithms whose output has a length comparable to the size of the images no volume transfer savings are achieved. Although in this case the processor utilization is still distributed among the image servers (this results in a speedup linear in the number of image servers, given a uniform distribution of images). Here, we assume that *number of search engines* $\ll$ *number of image servers.*

The graphical user interface of the demonstrator is shown in Fig. 3.4. The left view shows the panel which is used to launch index agents. From a list of known servers, a subset can be chosen. On pressing the button titled *start indexing*, index agents are created and dispatched to each selected server. On the target server, each index agent looks up the *pics* service which must be registered in the target server, and starts the feature extraction process. Upon completion, it publishes an instance of the *finder* service in the target server, which search agents can look up and query.

The middle view shows the panel which is used to dispatch search agents. Again, a number of servers can be chosen from a given list. The search agent takes a user-provided example image (which can as well be a sketch or prototypical image), hops in turn to all selected image servers, and collects image entries with a distance less than a given threshold and up to a given maximum number. The overall best matches (thumbnails but not full images) are reported back and presented in the results panel.

The results panel shows the retrieved thumbnails and each entry's distance to the query image. If the user clicks on a thumbnail then a fetch agent is created and dispatched to the host where the image was retrieved, and returns the corresponding full image. The check box in the lower left corner of the results panel enables secure retrieval. If it is checked then retrieved images are transported in encrypted form as explained in [40].

Although we implemented only one feature extracting and matching method so far, the interfaces are designed to support multiple and alternative implementations transparently. All implementations have been developed in the Java programming language (Java Version 2)[6].

[6]The middleware SeMoA and the implementation of the agent based search engine are distributed as open source software at `www.semoa.org`

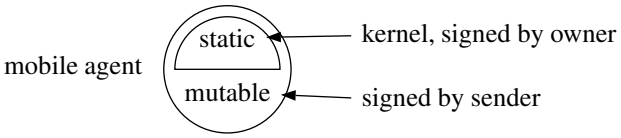


FIG. 3.5. *A mobile agent consists of static data (which does not change during the agent's lifetime e.g., code and a random agent ID) and mutable state. The owner assumes responsibility for her agent by signing its static part (the* kernel*), and the most recent host of an agent signs the entire agent to assume responsibility for the agent's most recent state.*

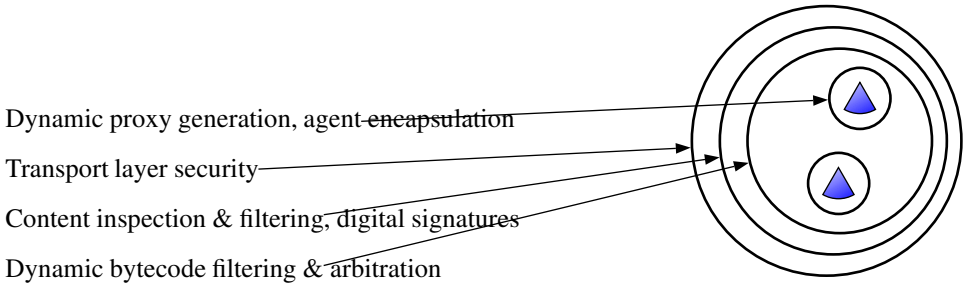


FIG. 3.6. *The security architecture resembles an onion. Agents have to pass all layers successfully to be admitted to the system. The outer layers keep threats out of the system. The innermost layer encapsulates and confines agents.*

3.2. Content Protection. It has been argued that mobile agents achieve a greater level of control over the media being searched on [6]. This is only part of the truth, though. In practice, various covert channels [26] as well as direct means of cheating can be used *e.g.*, by malicious index agents and colluding search agents to subvert image export restrictions. The billing schemes proposed by Belmon and Yee (which account for projected losses due to covert channels) punish thieves and ordinary clients likewise and will hardly be accepted. Still, using *e.g.*, the incubator model can improve confidence that image contents are not exported illicitly from image servers. Evidence of stealing images on the part of index agents can be established by reverse engineering the agents' code which is present at the image server, if it comes to the worst. SeMoA requires that each sender of an agent digitally signs the static parts of his agent (including the code), which establishes a non-repudiable proof of ownership. This signature yields a unique and unforgeable agent *kernel*. Furthermore, each server must sign the entire agent before transport. This signature binds the new state of the agent to its kernel and protects the agent against tampering during transport. Thereby each server documents its responsibility for any state changes that the agent may have undergone while being hosted by it (see Fig. 3.5 for an illustration of signatures).

Nevertheless, an agent must also have means of protecting itself against a malicious host as soon as for-profit services are involved. If search agents pass by multiple competing image providers there is a certain chance that a provider does not play by the rules. One possible attack that a dishonest provider may launch on an agent is to manipulate the accumulated search results or to replace or degrade the quality of images retrieved from a competitor. Thereby the perpetrator increases the likelihood that images are purchased from him the next time. Other attacks involve tampering with the agent's code in order to alter its negotiation strategy, decision making, or simply to cause harm at the servers of competitors.

SeMoA prevents tampering with the code by requiring that code must be digitally signed. Any such tampering invalidates the kernel of the agent and hence the agent subsequently fails to "speak on behalf of its original owner." Tampering with accumulated images is prevented by establishing encrypted subsections in the agent so that each subsection can be accessed only by the agent's owner and by hosts that he authorizes [40]. For instance, the fetch agent is programmed so that it stores each retrieved image in the section that is dedicated to its current host. This section is automatically encrypted by the *encrypt* filter when the agent departs.

This setup has a drawback, though. For security reasons agents do not carry private keys for the encrypted sections. Hence they cannot access Agents cannot access results that they collected prior to reaching the current host (these results are encrypted and by assumption the current host does not have a matching private key). This prevents an agent from pruning its accumulated results *e.g.*, to a fixed upper number of overall best results (we referred to this problem in §2.2). One workaround is to program agents so that they regularly migrate to a trusted and authorized host where the agent can access and prune all results that have been accumulated up to this point.

3.3. Server Safety and Security. For practical purposes it is essential that agent servers are protected against attacks by malicious agents. Agents must not be able to disrupt or otherwise negatively effect the operation of an image server

or other agents hosted by it. For mobile agent servers based on Java this is currently an elusive goal—the Java runtime system is vulnerable to a variety of *denial of service* attacks [7].

However, SeMoA makes a best effort to protect the runtime system against malicious code, and provides pluggable bytecode filtering and arbitration modules. Before a class is loaded into the name space of an agent each module may inspect, reject, or instrument the bytecode of that class. Currently, SeMoA includes a module that rejects classes which *e.g.*, override the `finalize` method, a well-known and simple way to attack the garbage collector thread of the virtual machine (a more subtle variant of this attack may be directed at the `close` method of some I/O classes). The same extension mechanism can be used to add resource control by bytecode arbitration [23] as well as additional security checks.

Each agent is run in a separate name space with a separate class loader and in a separate thread group. Thereby, interference of agents is prevented. A special security manager filters and sorts newly created threads so that for instance threads of the *Abstract Window Toolkit* are not accidentally placed in the thread group of an agent. Marshalling and unmarshalling is done by the initial agent thread from within the agent's sandbox so that an agent cannot exploit callbacks in the *Java Serialization Framework* to hijack server threads. The server transports an agent only after all threads of that agent have terminated, which prevents the adverse or inadvertent creation of clones or zombies (or, for that matter, widespread infection of servers by a worm).

Once running, an agent may publish and retrieve service objects (such as the *finder* service) by name in the server's object registry if the agent has appropriate permissions. Published objects are automatically wrapped in a proxy object which is created dynamically. The proxy prevents uncontrolled aliasing of the service object and automatically invalidates references to it as soon as the agent terminates. This makes the service object available for garbage collection. Agents cannot share classes (by virtue of separate name spaces the classes would not be type-compatible) but they may share interfaces for the purpose of method invocation and communication. However, two interfaces are shared across name spaces only if their implementations are mapped to the same value by a cryptographic hash function. In such a case, a super-ordinate class loader assures type-compatibility.

Before an agent is loaded and run, it must pass a configurable pipeline of pluggable filter modules (see also Fig. 3.1). Each filter may reject the agent in the case of errors. SeMoA provides a variety of security related filters some of which transparently handle digital signatures, certificate chain validation, and encryption of subsections of an agent. Agent transport is possible both in the clear and over a mutually authenticated *Secure Sockets Layer* (SSL) connection. A schematic illustration of SeMoA's security architecture is given in Fig. 3.6.

Moreover SeMoA allows the execution of agents from different agents system by means of an interoperability layer [42], such as JADE agents. The security measures are designed in a way that they apply fully to any kind of dynamically loaded code executed within the SeMoA middleware.

4. Integration of Mobile Agents and Web Services. According to the Web service architecture as defined by W3C [51], a *Web service* is an abstract notation which is implemented by a concrete *agent* as computational resource, owned by and acting on behalf of a person or organization. An agent realizes one or more services and may request other services, in turn. The Web service specification ensures the interoperability between systems through machine-to-machine interaction over a network, whereas it avoids any attempt to govern the implementation of agents.

On the one hand, this agent concept is defined on a high level of abstraction neglecting detailed characteristics of the numerous existing software agents systems. Furthermore, most current applications base either on Web service or on software agent technology, but not on both simultaneously. On the other hand, the Web service specification already indicates the integration of both technologies resp. hints at similarities and possible extensions/improvements of one technology through the other.

Similar to Web services, software agents can encapsulate business or application logic. Rather, software agents can dynamically discover, combine and execute such processes, and further offer multiple services or behaviors that can be processed concurrently. In order to move from one system to another or even to communicate with each other, mobile agents currently need a common platform on which they operate. Thus, they are useful for business partners only if these actually share a common platform. The consistent use of Web service standards for description of capabilities, communication, and agent discovery would establish interoperability not only between different agent platforms but also between agent platforms and traditional Web services, combining the advantages of two worlds.

To dynamically deploy both technologies without much overhead for the application developer mobile agents and Web services have to be integrated in a seamless manner, whereas mapping of processes have to be fully automated.

4.1. Requirements and Definitions. To integrate mobile agent and Web service technology in a seamless manner, components have to be designed, which map between the different mechanisms for *service description*, *service invocation*,

and *service discovery*, in both worlds. In other words, messages resp. representations from the according Web service protocols (WSDL, SOAP, UDDI) have to be translated into corresponding requests resp. data types of the agent system, and vice versa.

When we talk of a *Web service engine* we mean the summary of these components, which can be integrated into an existing agent system, and thereby extend this system with Web service abilities. Especially in some communication-centered agent systems, this engine further has to bridge the gap between the synchronous behavior of SOAP, and the asynchronous messaging paradigm as it is specified by FIPA with FIPA-ACL, for example (cf. [15]). Describing the level of integration, we want to define the following features for the integration of Web services and agents:

self-contained The Web service engine integrates all components for a seamless transition from one technology to the other, without the need of additional external resources.

bidirectional The Web service engine supports the invocation of Web services by agents as well as the provisioning of agent services by means of Web services. Thus, it allows cross boundary interaction in both directions.

automated After being properly configured and started, the Web service engine does not require any further manual steps executed by the user during runtime.

transparent The components of the Web service engine are transparently integrated into the agent system resp. the Web service infrastructure. Neither agents nor Web services as the acting entities recognize the existence of the Web service engine.

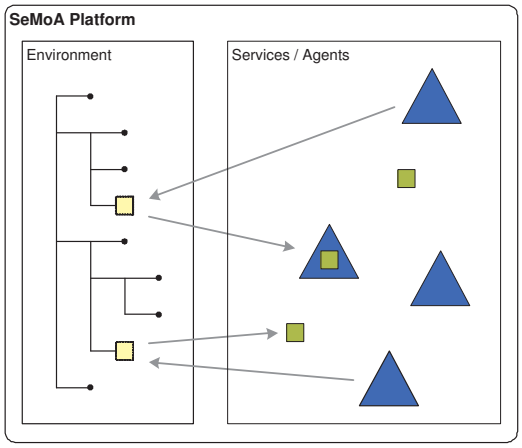


FIG. 4.1. *Hierarchical service environment.*

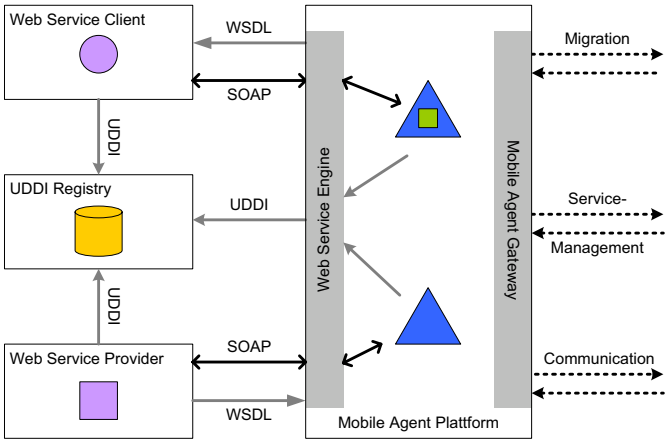


FIG. 4.2. *WSE Integration Architecture.*

4.2. System Integration. As already mentioned in §3.3, the only mean of interaction between mobile agents and/or services on the local server is based on direct method invocation through shared Java interfaces (cf. Fig. 4.1). Services are registered locally as entities within a hierarchical namespace (the service *environment*). A service requester searches for an appropriate service interface within a defined sub hierarchy of the environment. If successful, a Java object is returned which implements the given interface.

To integrate Web service technology in SeMoA, the local service management has been extended by a Web service engine (WSE) fulfilling all the requirements defined above. Presupposing that either a certain *Web service flag* has been set when publishing a service, or a service has been published in a certain *Web service enabled region* of the environment, this service is automatically wrapped by a Web service stub which is accessible by remote clients via SOAP requests. Further a corresponding Web service description is generated and registered on a remote UDDI-compliant directory server. The WSDL-conform service description allows both, a broad search for services according to a given category, and a concrete search for services implementing a given interface and/or having a specific name in a hierarchical namespace which corresponds to SeMoA's local environment. Thereby, a service interface is unequivocally identified by its name and the cryptographic hash code of its normalized method signature. When a service is requested from a Web service enabled region of the environment which does not exist locally, the WSE tries to find a corresponding Web service and returns a transparently generated Web service client stub, implementing the desired interface(s). Figures 4.3 and 4.4 show the corresponding extended processes for (Web) service deployment and undeployment resp. (Web) service discovery.

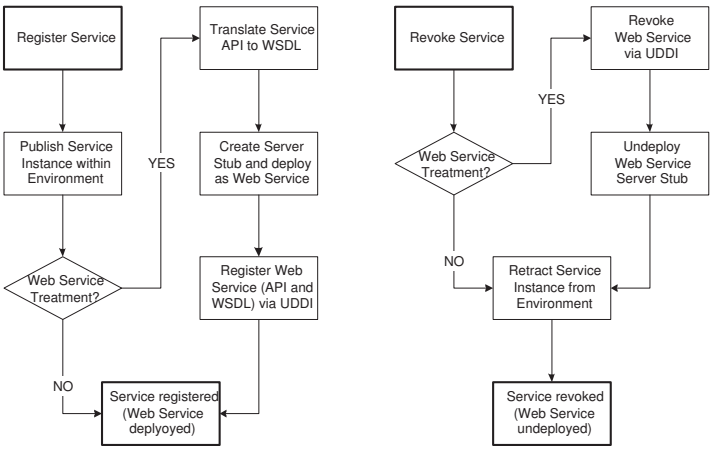


FIG. 4.3. *Deployment/Undeployment of a (Web) service.*

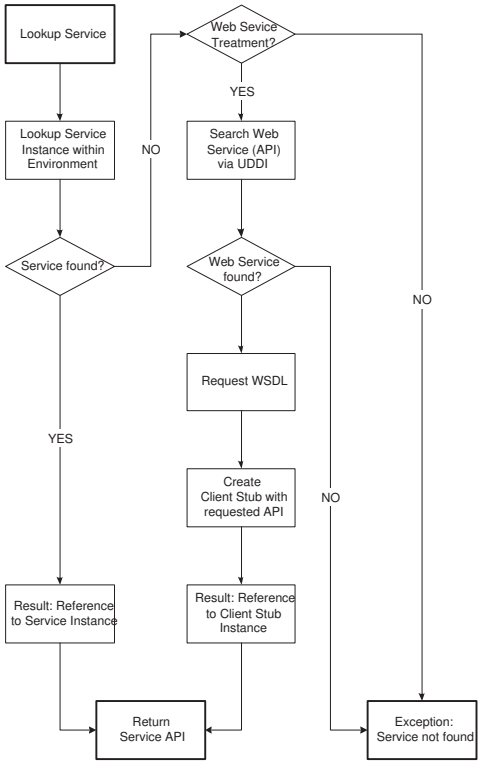


FIG. 4.4. *(Web) service discovery.*

Thus, in addition to the functionality provided by an existent mobile agent gateway, agents (△) are enabled to request and interact with external Web services (□), and external Web service clients (○) are enabled to request and invoke services encapsulated by agents (see Fig. 4.2). Further details about the internal WSE modules can be found in [37, 38].

4.3. Secutity Aspects. The current use of Web services to share information and services across organisations make necessary a new mean of access control. Besides, the use of Web service protocols imply a couple of Denial-of-Service attacks against XML parsers resp. new SOAP-specific attacks, *e.g.*, with respect to command injection or session hijacking.

Figure 4.5 gives an overview of existing Web service security mechanisms within the Web service protocol stack: *Transport Layer Security (SSL/TLS)* [19] offers a secure channel (point-to-point) between Web service client and Web service provider. *XML-Encryption* [22] and *XML-Signature* [4] provide basic security mechanisms for XML messages. These specifications are used within the *WS-Security* protocol family [3] to create appropriate SOAP security headers. Furthermore, high-level protocols as there are the *Security Assertion Meta Language (SAML)* [33] to exchange authentication and authorization information, or the XML *Access Control Markup Language (XACML)* [34] to define role-based access control policies allow specialized security functionalities. Currently, even a bunch of new high-level protocols are specified and standardized, such as *WS-Policy*, *WS-Trust*, *WS-Privacy*, *WS-Authorization*, *WS-SecureConversation*, etc. (cf. [21]).

At the latest when combining basic Web services to composite Web services with added-value, the simple interaction paradigm between an explcit client and one service provider as server becomes obsolete. Rather, a dynamic tree models temporary bidirectional requestor/provider associations as part of a dependency hierarchy, established by a client at the root of the tree (compare [35]). Since a requestor then accesses a remote resource through a provider's Web service on behalf of another entity (the client), authentication and authorization mechanisms have to be established, which support this kind of delegation principle, and concurrently minimize the overhead for the requesting client. In this context, [28] compares current authentication and authorization infrastructures, while [12] discusses and proposes a new framework for Web service access control based on the above presented standards which decouples authentication and authorization, and further takes dynamic aspects (as there is the request context by means of an access history) into account.

In our opinion, appropriate standard-based security mechanisms for Web services have to be selected and combined according to specified security requirements. Especially in the context of integrating mobile agents and Web services, the

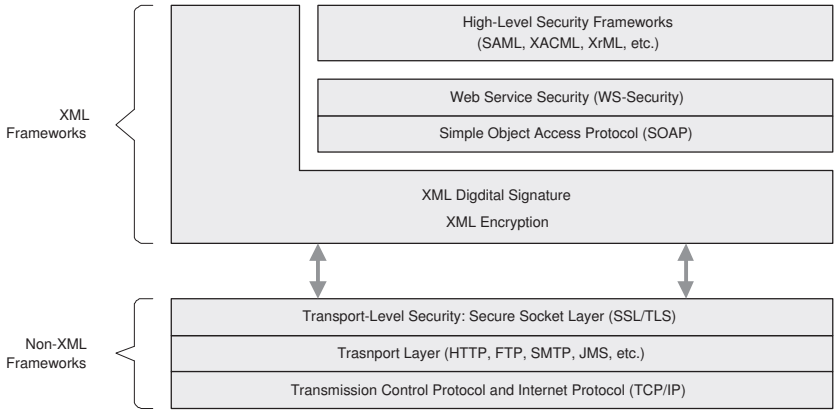


FIG. 4.5. *Basic Web service protocol stack including security layers [10].*

existing security frameworks for both technologies have to be thoroughly combined without enabling new side-channel attacks to the integrated system. Currently, we are combining transport layer security with enabled client authentication in point-to-point connections with SAML-based authentication resp. authorization tokens as part of the transmitted messages transport information about the security context of the acting entity from one host to another. Thereby, the cumulated security information is used to enforce the local security policy on the target platform.

5. Evaluation. In the incubator models, no feature vectors must ever be transported over the network. Therefore, a quantitative evaluation of this model is superfluous. We evaluated the gatherer model of our prototype for small sets of images in order to get a general idea of the potential savings that can be achieved by mobile agents compared to conventional client/server approaches. The setup of the experiments favored the conventional approach so that our results remain conservative. In practice, we expect that the relative performance of mobile agents is better. We used the hardware and software given below in our evaluation:

- 1 $\times$ Pentium III mobile, 1.2 GHz, 512 MB, FreeBSD 4.7, running Java Version 1.4.1 under the Linux emulation.
- 9 $\times$ Sun Ultra 5/10, 500 Mhz (UltraSPARC-IIe), SunOS 5.8/5.9, 256MB-512MB, running Java Version 1.4.1, *Apache* Server Version 1.3.
- Switched Fast Ethernet (100 Mbit/s)

The nine computers we used were connected by our institute's LAN, which consists of several hundred workstations and PCs, and which is accessed by more than 150 research assistants and a large number of students (although we did our tests a weekend to reduce distortion of measurements due to regular use of the network).

We first measured the performance of the conventional approach. A simple client program loaded and extracted the features of all images, with a varying number of image sources. The client was programmed in the Java programming language; it ran on the 1.2 GHz Pentium III laptop, it used three threads per image source in parallel to optimize I/O utilization (we found by experimenting that this gave the best results), and it was based on the same code that was used by the mobile agents in subsequent testing.

The image sources consisted of 8 Apache servers, each of which ran an Apache server with 48 images. All images were in JPEG format with a resolution of 756×504 pixels, and were loaded by the Apache server from the built-in hard drive. The size of images varied from approximately 280000 to 456000 bytes, depending on the JPEG compression rate. Each experiment was done three times to observe variances.

In the measurement of the mobile agent performance each Sun hosted a mobile agent server. The ninth Sun (without images) played the role of the broker. The other Suns were configured with a simple service that allows to iterate picture names and to retrieve image data for an image with a given name. Again, all images were loaded from the built-in hard drive.

In experiment one, we launched a mobile agent on the broker. This agent migrated to image server one, extracted the features of all images, transported the image entries back (excluding thumbnails), and merged the image entries with the broker's central index. In the second through eighth experiment, we launched two to eight agents in parallel which performed the same operations on the additional image servers. In each experiment, we measured the time from starting the first agent until the last agent returned and completed its task. Again, we repeated all experiments three times.

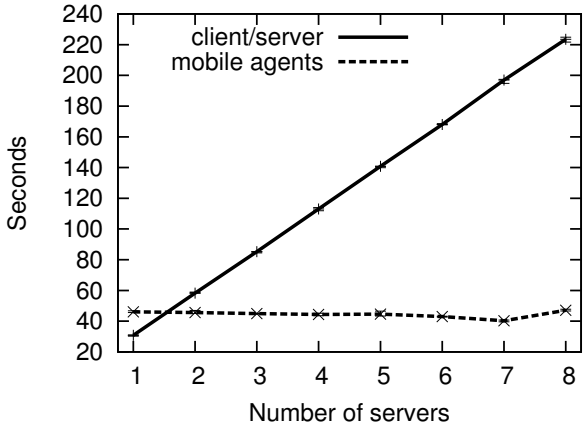


FIG. 5.1. *The time that the client/server based gather needs to complete the feature vector collection task vs. the time required by the mobile agent approach (measured from starting the first agent until completion of the last agent). The point of intersection is the break-even point at which the additional overhead of shipping and setting up mobile code is amortized by the reduced network utilization of the mobile code approach.*

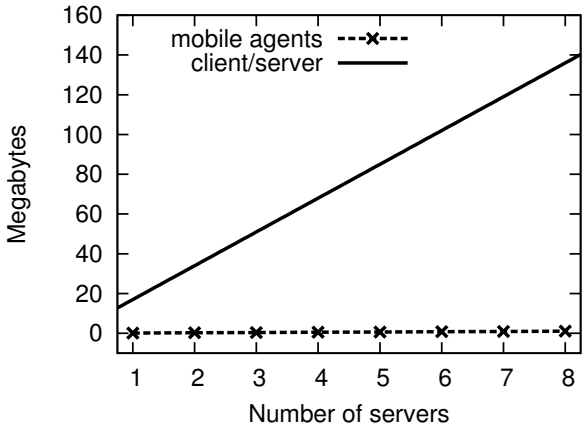


FIG. 5.2. *The amount of data transported by the network in the client/server based gatherer vs. the mobile agent approach. The client/server graph is plotted based on the size of all downloaded images times number of servers. The graph of the mobile agent approach has a slope which is too small to be noticeable compared to the upper graph.*

We also measured the sums of sizes of all image entries (including overheads for the agents) transported per experiment. Comparison of all collected results shows significant savings in favor of the mobile agent approach, as could be expected (see Fig. 5.1 and 5.2, min and max values deviate so little from the median and mean values of the client/server and mobile agent measurements that the error bars are hardly noticeable in the graphs).

Additionally, our image search engine was the subject of a field study with the objective to assess the legal aspects of electronic commerce with mobile agents. Over the period of two days, eleven lawyers used our retrieval system in the roles of the content provider, customer, and image broker, with the objective to trade images. Overall, more than a thousand agents were dispatched. Our prototype proved to be user-friendly, robust, and reliable.

6. Related Work. Considerable work is done in the general area of CBR, see *e.g.,* [27] for the proceedings of a recent conference. Well received work by several authors reports on CBR systems for the World Wide Web [45, 44, 5]. All these image search engines are based on the client/server paradigm of collecting images from the Web. Mobile agent technology is complementary to this work. It remains to be investigated how well the algorithms developed by the authors mentioned above can be adapted to be used within a mobile agent framework. The idea of using mobile agents for content based image retrieval has been mentioned before [39, 2, 54]. Mobile agents have also been applied in related applications. For instance, in [25], Johansen reports on the use of mobile agents in the context of a weather information system (mobile agents process and deduce weather information from satellite imagery). Early attempts to integrate static agents and Web services have been described in [31, 55, 8]. More sophisticated solutions, especially in the context of FIPA-compliant agent platforms, have been presented in [29, 14, 46, 20]. In contrast, the existing contributions based on mobile agents have more diverse goals, and thereby do not completely follow transparent integration of agents and Web services [9, 30, 35, 13, 24].

It is not our intention to claim originality of the idea, but to report unique aspects of our architecture (mobile agent based incubator and the gatherer model; combination with Web services), the results of our evaluation, and our experiences with respect to the usefulness of the application.

7. Summary. Digital images are a valuable commodity and we expect that more and more photo agencies make use of the Internet as the principal platform for advertising, customer relationship management, and—most importantly—content distribution. We presented two models of deploying mobile agents to gather image information from the Internet and a third extended model using Web services. All models take into account that content providers must retain control over their intellectual property. Multiple complimentary retrieval methods can operate in parallel. The models support flexible software distribution, updates, and deinstallation, and they can be extended to account for negotiation of license terms and automatic fingerprinting of retrieved images based on digital watermarks.

The models differ in the grade of decentralization. In the gatherer model, the amount of data transported over the network depends on the size of the images and the compression factor of the deployed feature extraction algorithms. In

our case, this is less than 1% of the image data transported by regular gatherers. In the incubator model, no image data is shipped over the network at all. Independently of the size of feature vectors, all models achieve a constant speedup, which is proportional to the number of image servers indexed in parallel. Image providers must set up and reserve computing resources for the mobile agent server. They can operate this server in conjunction with a Web server *e.g.*, by attaching the agent server to the Web server by Servlets. Furthermore, the extended incubator model allows to attach non mobile agent clients by means of an arbitrary Web service compliant middleware framework, which is adjusted to the special needs resp. abilities of the client device.

One avenue for improvement is the combination of the incubator and the gatherer model. The index agent that takes residence at the image provider may cluster the feature vectors *e.g.*, as described in [49]. Rather than sending only its *finished* message to the broker it may submit a number of centroids of the densest clusters. Search agents which visit the broker may thus opportunistically prune the search space by migrating only to servers with centroids most similar to the query vector.

Mobile agent infrastructures require a sound security model, which accounts for the various threats. Some progress has been achieved in the area of mobile agent security [47, 48], although a number of hard problems are still unsolved. Yet it is probably fair to say that in principle the attainable level of security is reasonable enough to justify the application of mobile agents in some real-world applications. However, before this can happen, runtime systems must become more robust *e.g.*, Java must become considerably more robust against *denial of service* attacks [7].

Using Web services in the extended incubator model opens a bunch of new Web service specific security risks and disadvantages. And although, there are already a bunch of different and specialized Web service compliant frameworks, some more work and research has to be done, before these Web service frameworks are completely interoperable with each other[7] and reach acceptable performance values comparable with other (proprietary) client/server protocols for distributed systems. Furthermore, the new Web service protocols still have to proof their applicability in the variety of Web service based application scenarios.

Acknowledgments. We thank Patric Kabus for his help in programming the basic search engine prototype. Further we thank Matthias Pressfreund and Dennis Bartussek for their help in programming the Web service extensions.

REFERENCES

[1] M. ARNOLD, M. SCHMUCKER, AND S. D. WOLTHUSEN, *Techniques and Applications of Digital Watermarking and Content Protection*, The Artech House Computer Security Series, Artech House, Norwood, MA, USA, 2003.

[2] C. ARORA, P. NIRANKARI, H. GHOSH, AND S. CHAUDHURY, *Content based image retrieval using mobile agents*, in Third International Conference on Computational Intelligence and Multimedia Applications (ICCIMA '99), 1999, pp. 248–252.

[3] B. ATKINSON, G. DELLA-LIBERA, S. HADA, M. HONDO, P. HALLAM-BAKER, J. KLEIN, B. LAMACCHIA, P. LEACH, J. MANFERDELLI, H. MARUYAMA, A. NADALIN, N. NAGARATNAM, H. PRAFULLCHANDRA, J. SHEWCHUK, AND D. SIMON, *Specification: Web Service Security (WS-Security)*, Tech. Report Versoin 1.0, IBM developerWorks, April 2002. `ftp://www6.software.ibm.com/software/developer/library/ws-secure.pdf`

[4] M. BARTEL, J. BOYER, B. FOX, B. LAMACCHIA, AND E. SIMON, *XML-Signature Syntax and Processing*, W3C recommendation, World Wide Web Consortium (W3C), February 2002. `http://www.w3.org/TR/xmldsig-core/`

[5] M. BEIGI, A. B. BENITEZ, AND S.-F. CHANG, *MetaSEEk: A content–based meta–search engine for images*, in Proc. Storage and Retrieval for Image and Video Databases VI (SPIE), San Jose, CA, USA, January 1998.

[6] S. G. BELMON AND B. S. YEE, *Mobile agents and intellectual property protection*, in Rothermel and Hohl [43], pp. 172–182.

[7] W. BINDER AND V. ROTH, *Secure mobile agent systems using Java – where are we heading?*, in Proc. 17th ACM Symposium on Applied Computing, Special Track on Agents, Interactions, Mobility, and Systems (SAC/AIMS), Madrid, Spain, March 2002, ACM.

[8] F. BÜSCHER, *How Multi-Agent Systems and Web Services can work together*, in Net.ObjectDays 2004: NODe Young Researchers Workshop '04, Erfurt, Germany, September 2004.

[9] J. CAO, Y. SUN, Y. WANG, AND S. DAS, *Scalable Load Balacning on Distributed Web Servers Using Mobile Agents*, Journal on Parallel and Distributed Computing, 63 (2003), pp. 996–1005. ISSN:0743-7315.

[10] M. CHANILIAU, *Web Services-Sicherheit und die SAML*, online article, XML and Web Services Magazin, January 2004. `http://www.entwickler.com/itr/online_artikel/psecom,id,468,nodeid,69.html`

[11] E. CHRISTENSEN, F. CURBERA, G. MEREDITH, AND S. WEERAWARANA, *Web Services Description Language (WSDL), Version 1.1*, W3C working group note, World Wide Web Consortium (W3C), March 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`

[12] M. COETZEE AND J. ELOFF, *Towards Web Service access control*, in Computers & Security, vol. 23, Elsevier, October 2004, pp. 559–570.

[13] D. COONEY AND P. ROE, *Mobile Agents Make for Flexible Web Services*, in Proceedings of The Ninth Australian World Wide Web Conference, Quensland, Australia, July 2003. `http://ausweb.scu.edu.au/aw03/papers/cooney/`

[14] J. DALE, A. HAJNAL, M. KERNLAND, AND L. Z. VARGA, *Integrating Web Services into Agentcities Recommendation*, tech. report, Agentcities Task Force, November 2003. `http://www.agentcities.org/rec/00006/actf-rec-00006a.pdf`

[15] FIPA, *ACL message structure specication*, Tech. Report FIPA document XC00061E, Foundation for Intelligent Physical Agents, August 2001. `http://www.fipa.org/specs/fipa00061/`

[7] Confer the ambitions of the Web Service Interoperability Organization (WS-I) at `www.ws-i.org`.

[16] K. FISCHER AND V. LOTZ, *Authorization and Delegation of Privileges in Mobile Agent Systems*, Mobile Agents, (2002).
[17] M. FOWLER AND K. SCOTT, *UML Distilled*, Addison-Wesley, Reading, Massachusetts, U. S. A., 1997.
[18] S. FRANKLIN AND A. GRAESSER, *Is it an agent, or just a program?*, in Intelligent Agents III, vol. 1193 of Lecture Notes in Artificial Intelligence, Berlin, 1997, Springer Verlag, pp. 21–36.
[19] A. O. FREIER, P. KARLTON, AND P. C. KOCHER, *The SSL Protocol, Version 3.0*, internet draft, Netscape, November 1996. `http://wp.netscape.com/eng/ssl3/`
[20] D. GREENWOOD AND M. CALISTI, *Engineering Web Service—Agent Integration*, in IEEE International Conference on Systems, Man and Cybernetics (SMC 2004), The Hague, The Netherlands, October 2004.
[21] IBM AND MICROSOFT, *Security in a Web Services World: A Proposed Architecture and Roadmap*, whitepaper, IBM Developer Works, April 2002. `http://www-128.ibm.com/developerworks/library/specification/ws-secmap/`
[22] T. IMAMURA, B. DILLAWAY, AND E. SIMON, *XML Encryption Syntax and Processing*, W3C recommendation, World Wide Web Consortium (W3C), Dezember 2002. `http://www.w3.org/TR/xmlenc-core/`.
[23] P. R. C. IN JAVA: APPLICATION TO MOBILE AGENT SECURITY, *Walter binder, jarle g. hulaas, and alex villanzon*, in 1st Int'l Workshop on Secure Mobile Multi-Agent Systems at the 5th Int'l Conference on Autonomous Agents, Montreal, Canada, May 2001.
[24] F. ISHIKAWA, N. YOSHIOKA, Y. TAHARA, AND S. HONIDEN, *Toward Synthesis of Web Services and Mobile Agents*, in Proceedings of the AAMAS'2004 Workshop on Web Services and Agent-based Engineering (WSABE), New York, USA, July 2004. `http://honiden-lab.ex.nii.ac.jp/~f-ishikawa/docs/wsabe2004/camera-ready.pdf`
[25] D. JOHANSEN, *Mobile agent applicability*, in Rothermel and Hohl [43], pp. 80–98.
[26] B. W. LAMPSON, *A note on the confinement problem*, Communications of the ACM, 10 (1973), pp. 613–615.
[27] M. S. LEW, N. SEBE, AND J. P. EAKINS, eds., *Proceedings of the International Conference on Image and Video Retrieval*, vol. 2383 of Lecture Notes in Computer Science, Springer Verlag, July 2002.
[28] J. LOPEZ, R. OPPLIGER, AND G. PERNUL, *Authentication and authorization infrastructures (AAIs): a comparative survey*, in Computers & Security, vol. 23, Elsevier, October 2004, pp. 578–590.
[29] M. LYELL, L. ROSEN, M. CASAGNI-SIMKINS, AND D. NORRIS, *On Software Agents and Web Services: Usage and Design Concepts and Issues*, in The 1st International Workshop on Web Services and Agent-based Engineering, Sydney, Australia, July 2003.
[30] Z. MAAMAR, Q. Z. SHENG, AND B. BENATALLAH, *Interleaving Web Services Composition and Execution Using Software Agents and Delegation*, in The 1st International Workshop on Web Services and Agent-based Engineering, Sydney, Australia, July 2003.
[31] L. MOREAU, *Agents for the Grid: A Comparison with Web Services (Part I: the transport layer)*, in Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002), Berlin, Germany, May 2002.
[32] OASIS, *Universal Description, Discovery and Integration (UDDI), Version 3*, technical committee specification, Organization for the Advancement of Structured Information Standards (OASIS), October 2003. `http://www.uddi.org/pubs/uddi_v3.htm`
[33] ———, *Security Assertions Markup Language (SAML), Version 2.0*, working draft, Organization for the Advancement of Structured Information Standards (OASIS), 2004. `http://www.oasis-open.org/committees/security`
[34] ———, *XML Access Control Markup Language (XACML), Version 2.0*, committee draft, Organization for the Advancement of Structured Information Standards (OASIS), September 2004. `http://docs.oasis-open.org/xacml/access_control-xacml-2_0-core-spec-cd-02.pdf`
[35] A. PADOVITZ, S. KRISHNASWAMY, AND S. W. LOKE, *Towards Efficient Selection of Web Services*, in The 1st International Workshop on Web Services and Agent-based Engineering, Sydney, Australia, July 2003.
[36] G. PASS, R. ZABIH, AND J. MILLER, *Comparing images using color coherence vectors*, in Proc. ACM Conference on Multimedia, Boston, Massachusetts, U. S. A., November 1996.
[37] J. PETERS, *Integration of Mobile Agents and Web Services*, in Proceedings of The First European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2005), Software Technology Research Laboratory, De Montfort University, Leicester, UK, April 2005, De Montfort University, pp. 53–58.
[38] J. PETERS AND J. OETTING, *Interoperability with Component Standards and Web Services*, tech. report, SicAri Consortium, 2004. Technical Report.
[39] V. ROTH, *Distributed image indexing and retrieval with mobile agents*, in IEE European Workshop on Distributed Imaging, no. 1999/109 in IEE Electronics & Communications, Savoy Place, London, WC2R 0BL, UK, November 1999, IEE, pp. 14/1–14/5. ISSN 0963-3308.
[40] V. ROTH AND V. CONAN, *Encrypting Java Archives and its application to mobile agent security*, in Agent Mediated Electronic Commerce: A European Perspective, F. Dignum and C. Sierra, eds., vol. 1991 of Lecture Notes in Artifical Intelligence, Springer Verlag, Berlin, 2001, pp. 232–244.
[41] V. ROTH AND M. JALALI, *Concepts and architecture of a security-centric mobile agent server*, in Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001), Dallas, Texas, U.S.A., March 2001, IEEE Computer Society, pp. 435–442. ISBN 0-7695-1065-5.
[42] V. ROTH, U. PINSDORF, AND W. BINDER, *Mobile Agent Interoperability Revisited*, in Mobile Agents 2001. Poster Session, K. Marzullo, A. L. Murphy, and G. P. Picco, eds., Atlanta, Georgia, USA, December 2001, 5th IEEE International Conference Mobile Agents (MA), IEEE Society Press, pp. 5–8.
[43] K. ROTHERMEL AND F. HOHL, eds., *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, vol. 1477 of Lecture Notes in Computer Science, Springer Verlag, Berlin Heidelberg, September 1998.
[44] S. SCLAROFF, L. TAYCHER, AND M. L. CASCIA, *ImageRover: A content-based image browser for the world wide web*, in Proc. IEEE Workshop on Content-based Access of Image and Video Libraries, San Juan, Puerto Rico, June 1997.
[45] J. R. SMITH AND S.-F. CHANG, *An image and video search engine for the world–wide web*, in Proc. Storage and Retrieval for Image and Video Databases V (SPIE), San Jose, CA, USA, February 1997.
[46] L. Z. VARGA, *WSDL2Agent: Tool to Help the Integration of Existing Web Services into Agent Systems and Semantic Web Service Environments*. `http://sas.ilab.sztaki.hu:8080/wsdl2agent/`
[47] G. VIGNA, ed., *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, Springer Verlag, Berlin Heidelberg, 1998.
[48] J. VITEK AND C. JENSEN, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of Lecture Notes in Computer Science, Springer-Verlag Inc., New York, NY, USA, 1999.

[49] S. VOLMER, *Buoy Indexing of Metric Feature Spaces for Fast Approximate Image Queries*, in Proc. Eurographics 2001 Workshop on Multimedia, Manchester, UK, September 2001, pp. 121–130.
[50] W3C, *Simple Object Access Protocol (SOAP), Version 1.2*, W3C recommendation, World Wide Web Consortium (W3C), June 2003. `http://www.w3.org/TR/soap/`
[51] ———, *Web Services Architecture*, W3C working group note, World Wide Web Consortium (W3C), February 2004. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`
[52] J. E. WHITE, *Mobile agents*, in Software Agents, J. Bradshaw, ed., AAAI/MIT Press, Menlo Park, CA, 1997, ch. 18, pp. 437–472.
[53] M. WOOLRIDGE AND N. JENNINGS, *Intelligent agents: Theory and pratice*, Knowledge Engineering Review, 10 (1995).
[54] J. YOU AND H. A. COHEN, *A new approach to image retrieval by fast indexing and searching*, in Proc. DICTA'97, Auckland, N.Z., 1997, pp. 425–430.
[55] S.-T. YUAN AND K.-J. LIN, *WISE - Building Simple Intelligence into Web Services*, in The 1st International Workshop on Web Services and Agent-based Engineering, Sydney, Australia, July 2003.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 16, 2006

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 119–119. http://www.scpe.org
ISSN 1895-1767

BOOK REWIES

EDITED BY SHAHRAM RAHIMI

Expert Systems: Principles and Programming
Joseph C. Giarratano and Gary D. Riley
Fourth Edition, Course Technology, Boston, MA, 2004
856 pages, $131.95
Language: English
ISBN: 0534384471

Expert systems is one of the most successful, practical, and recognizable subsets of classical Artificial Intelligence. The ability to supply decisions or descision-making support in a specific domain has seen a vast application of expert systems in various fields such as healthcare, military, business, accounting, production, video games, and human resources. The theoretical and practical knowledge of expert systems is indispensable for a computer science, management information science or software engineering student.

In its fourth edition, *Expert Systems: Principles and Programming,* as the title suggests, aims to be used as a complete textbook on this topic. The authors are respected authorities on expert systems, and were involved in the development of the popular CLIPS expert system tool which is dealt with thoroughly in this book. The book itself is divided into two major sections: the first six chapters deal with the theory of expert systems, the rationale behind their historical development and the current state of research; the next six sections are an introduction to CLIPS and how to use it to develop practical applications. The clear division between theory and practice serves to guide the student in choosing specific topics throughout the book. Each chapter ends with a set of problems and a useful bibliography. Appendix G provides a comprehensive list of software resources and will prove to be a very valuable asset to the student interested in exploring the practical aspects of expert systems as well as those who will be developing commercial applications incorporating expert systems.

The first six chapters provide an indepth intoduction to expert systems, and deal with the representation of knowledge and methods of inference and reasoning. Each of these topics are introduced from scratch—for example, when dealing with knowledge representation, logic is described from its very basics, starting from propositional logic. The chapters on reasoning under uncertainty and inexact reasoning are very well constructed and are the most attractive feature of the book for me. Topics such as fuzzy logic and Dempster-Shafer theory are explained in good detail along with their practical significance. There is an objective flow through the chapters which helps to tie in the concepts and give an understanding on the progress of topics. Also, the disadvantages and pitfalls behind expert systems in general, and specific topics are well documented.

The second section focusing on the CLIPS expert system tool is meant to be an aid in understanding and reinforcing the concepts of the first section, but does not require a thorough reading of the latter. CLIPS, developed in part by the authors at NASA, has become quite popular as a tool for studying expert system in many university courses as well as being used in several commercial and industrial applications. The expert system programming in CLIPS does not require much experience with programming and can be picked up rapidly thanks to its simple syntax and the helpful examples in the book. A new feature of the fourth edition is the introduction of COOL, the CLIPS Object-Oriented Language, which allows expert systems programmers to develop their systems in an object-oriented environment. The section is supplemented by a CD containing Windows and MacOS executables for CLIPS and reference guides—all of which can be downloaded from the Internet as well.

Although the CLIPS examples deal with some problems of uncertainty in reasoning, there is no mention of using fuzzy logic or Dempster-Shafer theory in a practical setting—a serious disadvantage to the effectiveness of the book, considering the availability of tools such as FuzzyClips. The book itself is expensive—on the wrong side of a hundred dollars, but that seems to be the trend for academic books these days and shouldn't be a deterrent to anyone interested in the subject considering it has become one of the standard textbooks on expert systems.

Raheel Ahmad,
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2_ε using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the PDCP WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.