Aglets: Programming Mobile Agents in Java

Danny B. Lange[1], Mitsuru Oshima[1], Günter Karjoth[2], and Kazuya Kosaka[1]

[1] Tokyo Research Laboratory, IBM Japan Ltd.
[2] Zurich Research Laboratory, IBM Swiss Ltd.

Abstract. Mobile agents are programs that can be dispatched from one computer and delivered to a remote computer for execution. Arriving at the remote computer, they present their credentials and obtain access to local services and data. They also provide a single uniform paradigm for distributed object computing, encompassing synchrony and asynchrony, message-passing and object-passing, and stationary objects and mobile objects. In this paper, we describe our Java-based mobile agents called *Aglets* and present its programming interface, called *Java Aglet API.*

1 Introduction

Java[3] and applets [1] have revolutionized the Web, and executable content has become a common term in the Web glossary. Applets are essentially sets of program code that can be downloaded, instantiated, and executed in Web browsers. Recently, this concept has been matched by the introduction of the servlet [2]. A servlet moves program code in the opposite direction to an applet; that is, it allows the client program to upload additional program code to a server. The servlet's code is then instantiated and executed in the server.

The aglet represents the next leap forward in the evolution of executable content on the Internet, allowing program code to be transported along with state information. Aglets [4] are Java objects that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, be dispatched to a remote host, and resume execution there. When the aglet moves, it takes along its program code as well as its data. Conceptually, the aglet is a mobile agent because it supports the ideas of autonomous execution and dynamic routing on its itinerary.

The Java Aglet API (J-AAPI) [3] is a proposed industry standard for Java-based mobile agents. J-AAPI was developed by a research team at the IBM Tokyo Research Laboratory in Japan in response to a call for a uniform platform for mobile agents in heterogeneous environments such as the Internet.

The notable thing about J-AAPI is that it extends the renowned write once, run everywhere capability of Java to agents. That is, once you have written an aglet, it will run on every machine that supports J-AAPI. You do not have to concern yourself with the underlying hardware or operating system, or with the nature of the particular implementation of J-AAPI on the host where your aglet is running.

[3] Java is a trademark of Sun Microsystems, Inc.

In this paper, we describe the technical advantages of mobile agents, give an overview of Aglets, and explain how to program mobile Internet agents by using J-AAPI. Next, we briefly describe the Aglets Security Model, sketch some usage scenarios, and relate Aglets to other work. Finally, we draw some conclusions.

2 So Why Mobile Agents?

Today, the central paradigm linking all distributed object technologies is a synchronous message-passing paradigm whereby all objects are distributed, but stationary, and interact with each other through message-passing. This paradigm is incomplete, and needs to be enhanced in some fashion with additional paradigms such as asynchronous message-passing, object mobility, and active objects.

Mobile agents can provide a single uniform paradigm for distributed object computing, encompassing synchrony and asynchrony, message-passing and object-passing, and stationary objects and mobile objects. Mobile agents, like elementary particles in physics, are fundamental to distributed object computing. Along with mobility, agents have the following unique and important computational characteristics:

Object-passing: When a mobile agent moves, the whole object is passed; that is, its code, data, execution state, and travel itinerary are all passed together.

Autonomous: The mobile agent contains sufficient information for it to decide what to do, where to go, and when to go.

Asynchronous: The mobile agent has its own thread of execution and can execute asynchronously.

Local interaction: The mobile agent interacts with other mobile agents or stationary objects locally. If needed, it can dispatch messenger agents or surrogate agents, which are all mobile agents, to facilitate remote interaction.

Disconnected operation: The mobile agent can perform its tasks whether the network connection is open or closed. If the network connection is closed and it needs to move, it can wait until the connection is reopened.

Parallel execution: Multiple mobile agents can be dispatched to different sites to perform tasks in parallel.

The technical advantages of mobile agents are many, and there is no single alternative to all of the functionality they provide [11]. In addition to supporting existing network services, mobile agents are also likely to make possible new services and thus new businesses.

3 J-AAPI and the Aglet Object Model

This section provides a quick review of elements of the Java Aglet Application Programming Interface (J-AAPI).

J-AAPI is a proposed standard for interfacing aglets and their environment. This is the API you will be using to create aglets. It contains methods for

initializing an aglet, message-handling, and dispatching, retracting, deactivating/activating, cloning, and disposing of the aglet. J-AAPI is simple, flexible, and stable. You can write platform- independent aglets and expect them to run on any host that supports this interface.

The interfaces and classes of J-AAPI are as follows:

- J-AAPI Interfaces
  - AgletContext
  - FutureReply
  - MessageManager
- J-AAPI Classes
  - Aglet
  - AgletIdentifier
  - AgletProxy
  - Itinerary
  - Message

First, we will describe the object model underlying J-AAPI. The aglet object model defines a set of abstractions and the behavior needed to leverage mobile agent technology in Internet-like open wide-area networks. The key abstractions found in this model are aglet, context, proxy, message, itinerary, and identifier:

- An **aglet** is a mobile Java object that visits aglet-enabled hosts in a computer network. It is *autonomous*, since it runs in its own thread of execution after arriving at a host, and *reactive*, because of its ability to respond to incoming messages.
- A **context** is an aglet's workplace. It is a stationary object that provides a means for maintaining and managing running aglets in a uniform execution environment where the host system is secured against malicious aglets. One node in a computer network may host multiple contexts.
- A **proxy** is a representative of an aglet. It serves as a shield for the aglet, protecting it from direct access to its public methods. The proxy also provides location transparency for the aglet. That is, it can hide the real location of the aglet.
- A **message** is an object exchanged between aglets. It allows for synchronous as well as asynchronous message-passing between aglets. Message-passing can be used by aglets to collaborate and exchange information in a loosely coupled fashion. A message manager allows concurrency control of incoming messages.
- An **itinerary** is an aglet's travel plan. It provides a convenient abstraction for non-trivial travel patterns and routing.
- An **identifier** is bound to each aglet. This identifier is globally unique and immutable throughout the lifetime of the aglet.

Behavior supported in the aglet object model includes creation, cloning, dispatching, retraction, deactivation, activation, disposing, and messaging:

- The **creation** of an aglet takes place in a context. The new aglet is assigned an identifier, inserted in the context, and initialized. The aglet starts executing as soon as it has been successfully initialized.
- The **cloning** of an aglet produces an almost identical copy of the original aglet in the same context. The only differences are the assigned identifier and the fact that execution restarts in the new aglet. Notice that execution threads are not cloned.
- **Dispatching** an aglet from one context to another removes it from its current context and inserts it into the destination context, where it restarts execution (execution threads will not migrate). We say that the aglet has been pushed to its new context.
- The **retraction** of an aglet pulls (removes) it from its current context and inserts it into the context from which the retraction was requested.
- The **deactivation** of an aglet temporarily removes it from its current context and places it in secondary storage. **Activation** of an aglet restores it in a context.
- The **disposal** of an aglet halts its current execution and removes it from its current context.
- **Messaging** between aglets involves sending, receiving, and handling messages synchronously as well as asynchronously.
- A **naming** mechanism automatically assigns immutable identities to new aglets. These identities should be guaranteed to be globally unique.

4 Aglet Programming

The Aglet class is the key class in J-AAPI. This is the abstract class that you, as an aglet developer, will use as base class when you create a customized aglet. The Aglet class defines methods that allows the aglet to control its own life cycle, namely, methods for cloning, dispatching, deactivating, and disposing of itself. It also defines methods that are intended to be overridden in its subclasses by the aglet programmer, and provide you with the necessary "hooks" to customize the behavior of the aglet. These methods are systematically invoked by the system when certain events take place in the life cycle of an aglet. Table 1 shows the relationship between these life cycle events and methods in the aglet.

It is easy for you to create a customized aglet. Start by importing the aglet package, which contains all the definitions of the J-AAPI. Next, define the `MyFirstAglet` class, which is inherited from the `Aglet` class:

```
import aglet.*;
public class MyFirstAglet extends Aglet { ... }
```

For instance, if you want your aglet to perform some specific initialization when it is created you just need to override its `onCreation` method:

```
public void onCreation(Object init) {
   // Do some initialization here...
}
```

The event	As the event takes place	After the event has taken place
Creation		`onCreation()`
Cloning	`onCloning()`	`onClone()`
Dispatching	`onDispatching()`	`onArrival()`
Retracting	`onReverting()`	`onArrival()`
Disposal	`onDisposing()`	
Deactivation	`onDeactivating()`	
Activation		`onActivation()`

Table 1. Events and methods in the aglet.

When an aglet has been created or arrives in a new context, it is given its own thread of execution through a system invocation of its run method. You can view this invocation as a means of giving the aglet a degree of autonomy. The `run` method is called every time the aglet arrives or is activated in a new context. We can say that the `run` method becomes the main entry point for the aglet's thread of execution. By overriding this method, you can customize your aglet's autonomous behavior.

```
public void run() {
   // Do something here...
}
```

For example, you can use `run()` to let the aglet dispatch itself to some remote context. You can do so by letting the aglet call its dispatch method with the Uniform Resource Locator (URL) of the remote host as the argument. This URL should specify the host and domain names of the destination context, and the protocol (`atp`) to be used for transferring the aglet over the network:

```
dispatch(new URL("atp://some.host.com"));
```

So what exactly happens to your aglet when `dispatch()` is called? Well, basically the aglet will disappear from your machine and reappear in the same state at the specified destination. First, a transfer protocol is used to bring the aglet (byte code and state information) safely over the network. Next, a special technique called object serialization [5] is used to preserve the state information of the aglet. What it does is to make a sequential byte representation of the aglet that later can be de-serialized to recreate the aglet's state.

Aglets can also receive messages. Message-handling in the aglet follows a call-back scheme. When a message is sent to the aglet, its handleMessage method is invoked with the actual message passed along as argument. It is now up to this method to handle the incoming messages. It should return true if a given message is handled; otherwise, it should return "false". The sender will then know if the aglet actually handled the message. In this example, the aglet will recognize and respond accordingly to "hello"messages:

```
public boolean handleMessage(Message msg) {
   if ("hello".equals(msg.kind)) {
      doHello();    // Respond to the 'hello' message...
      return true;  // Yes, I handled this message.
   } else
      return false; // No, I did not handle this message.
}
```

5 The Agent Transfer Protocol (ATP)

The *Agent Transfer Protocol* (ATP) is an application-level protocol for distributed agent-based systems. Aimed at the Internet, ATP offers a simple and platform-independent protocol for transferring agents between networked computers. While mobile agents may be programmed in many different languages and for a variety of vendor-specific agent platforms (consisting of virtual machines and libraries), ATP offers the opportunity to handle agent mobility in a general and uniform way:

- A machine hosting agents has an ATP-based agent service, which is a component capable of receiving and sending agents from remote hosts via the ATP protocol. The agent service is identified by a unique address, independent of the specific agent platforms supported by the machine. A machine can run multiple agent services.
- A machine can host different types of agents, provided it supports the corresponding agent platforms.
- Any agent platform should include a handler of ATP messages.
- An ATP message carries sufficient information to identify the specific agent platform (at the receiver host) then its ATP handler is called to handle the message.

The ATP protocol is based on a request/response paradigm between ATP services. ATP service A establishes a connection with ATP service B, then sends a request to B and waits for the response. Thus, A acts as a sender (of the request) and B acts as a recipient. An agent host machine can support more than two agent systems, which may be provided by different vendors (see figure 1).

A request includes a request line, specifying the request method, the protocol version, and the required resource, followed by a MIME-like message containing request modifiers, sender information, and possible content in its body. A response include a status line, specifying a success or error code and the protocol version, followed by a MIME-like message containing response modifiers, sender information, and possible content in its body. ATP defines four standard request methods for agent services: dispatch, retract, fetch, and message.

Dispatch an Agent: To dispatch an agent from an agent service A to an agent service B, A sends a dispatch request to B. The agent is contained in the

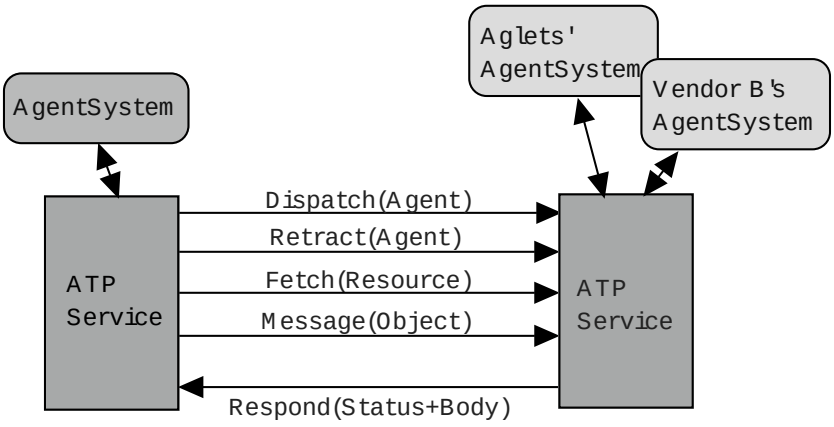

Fig. 1. Agent Transfer Protocol

body of the request. When B receives the agent, it will send A a response containing a status code.

Retract an Agent: To retract an agent from an agent service B to agent service A, the latter sends a retract request to B. The reply from B is a response with a status code and the specified agent in its body.

Fetch a Class: To execute the agent, an agent service B may need to retrieve executable code from the agent's origin; let us say this origin is an agent service A. To do so, B sends a fetch request to A. The reply from A is a response with a status code and the required executable code in its body.

Send a Message: To send a message from an agent in agent service A to an agent in agent service B, A sends B a message request, containing that message. The reply from B is a response with a status code.

ATP/0.1 is implemented in the atp-package as a part of the IBM Aglets Workbench, which can be downloaded from: `http://www.trl.ibm.co.jp/aglets/`.

6 The Aglet Security Model

The *Aglet Security Model* provides the overall framework for aglet security. The purpose of the model is to show the flexibility with which various security policies can be defined. It describes how and where a secure system enforces security policies.

Security policies, defined by one administrative authority and a set of rules, specify:

- under what conditions aglets may access objects;

- what authentication is required to prove the identity of users and other principals, what they can do, and whether they can delegate their rights;
- the security of communications between aglets and between contexts, including the trust required between them;
- what degree of accountability is necessary for which security-relevant activities.

Since aglets are Java objects that can migrate from one host to another, they have potential access to all resources on the host. They also rely on the security of the Java interpreter for their proper execution. Thus, aglet security and Java security go hand in hand. All the security concerns raised about Java also affect the safe execution of aglets. A small, local bug in the implementation of the hosting Java interpreter will affect the security of the aglet workbench.

Principals and Identities

Authentication gives an assurance of identity. The legitimate owner of an identity is known as a principal. A principal can be an individual, a corporation, a daemon thread, or a smart card; anything that can have an identity. An identity consists of a name and possibly other attributes. Identities are registered in and authenticable by the system.

For the individual aglets we identify the following three principals:

- Every aglet has an identifier that is unique over the aglet's life time, that is, independent of the context in which the aglet is currently executing. The aglet identity includes the aglet's class name, a kind of "product" name, for authentication. The basic identifier might be used in policies that refer to specific instances of aglets, such as that an aglet may dispose of any of its sons. An aglet's product name might be used when only a certain type of aglet is meant, such as aglets of the class `ibm.aglets.samples.Watcher` might access specific HTML files.
- The aglet manufacturer (a human or a company) produces a well-defined and reliable aglet. It is in the manufacturer's interest that no claims be issued for damage caused by a malfunctioning aglet.
- The aglet owner (who may also be a human or a company) is concerned mainly about the safety of the dispatched aglet. Can the returned results be trusted? Did every context execute the aglet properly? To ensure its safety, the owner may define security preferences, a set of rules that specify who may access/interact with the aglet on its itinerary. However, as the aglet has to rely on the context to comply with these rules, they are no more than a statement of intention. Security preferences also allow the owner to limit the aglet's capabilities, for example, by specifying some global allowance on the maximum number of hops, cpu-time consumption, an so on.

For aglet hosting we identify the following three principals:

- The context manifests the execution platform of the aglet. Its identity is the URL of the host together with a qualifier if there is more than one context. Unlike aglets, contexts are long-lived objects and thus may keep their identity, - that is, their address, - even after updates or complete replacements of the software/hardware realizing the context. For very security-critical applications in which trust is associated with a specific version of the context, the identity of a context must have an attribute like the serial number of a CPU. Since a software license might be granted only to a specific computer, identified by the serial number of its CPU, a context's serial number refers to a specific release of its software/hardware.
- The context manufacturer (company) produces a reliable context according to the J-AAPI specification. It is in the manufacturer's interest that no damage can be claimed to have been caused by a malfunctioning context.
- The context master (human or company) is responsible for the safety and security of the context and its underlying machine. The context master defines the security policy for the context under his or her control, namely, the protection of the local resources against aglets. It is also the context master's responsibility to guarantee that no aglet can interfere with any other aglet active in the same context if it is not explicitly permitted to do so by the aglet owner.

Security Policies

We envision that some of the principals introduced here particularly the aglet owner and the context master, may define security policies. A secure aglet system realizes the composition of all security policies involved. For example, although the owner of the aglet might have specified that it can consume up to 10 seconds within each visited context, the master of the current context may set a limit of 5 seconds.

The above example shows that there is an order among security policies defined by different principals, such as:

$$\text{AgletManufacturer} < \text{AgletOwner} < \text{ContextMaster}$$

indicating that the context master authority has the strongest impact on the execution of aglets within a given context.

Contexts have to protect themselves against aglets, and aglets must be prevented from interfering with each other. The aglet context establishes a reference monitor, which gives an aglet access to a resource only if it is in accordance with the access control policy established by the context master. Thus a context establishes a domain of logically related services under a common security policy governing all aglets in that context.

The master of the context configures authorization policies for incoming aglets. In general, there is a hierarchy of authorization policies:

- general level for an unauthenticated manufacturer
- organization level for an unauthenticated owner

– per-aglet level otherwise.

Additionally, authorization may be given with respect to computing power, occupancy level, organizational affiliation, pricing, code certification, or by type of aglet.

According to a security policy defined by using above terms, the reference monitor of an aglet context might give permission to:

– obtain file information,
– read local files,
– write local files,
– delete files,
– connect to network ports in the originating context,
– connect to network ports in other contexts,
– load libraries, and
– create pop-up windows.

The above resources are taken from the Java model. In an aglet system, there are additional resources for such things as:

– creating new aglets,
– cloning a specified aglet,
– dispatching/disposing of an aglet,
– retracting remotely located aglets into the current context,
– retrieving a list of fellow aglets present in the same context,
– obtaining the context property indicated by the key.

The context interface is used by an aglet to gain information about its environment, including other aglets currently active in that environment, and eventually to interact with them.

An aglet also carries the security preferences of its owner, which include rules that govern the aglet's consent for cooperation. However, the aglet has to rely on the context for compliance. The aglet's security preferences describe who and under which circumstances somebody may

– dispose of,
– deactivate,
– activate,
– clone,
– dispatch, or
– retract the aglet.

7 Usage Scenarios

A few usage scenarios are sketched below.

Accessing Databases

A client creates an aglet to gather information from several databases. The client provides the aglet with an itinerary and sends it off to the first database server. The itinerary controls the routing of the aglet and is able to handle exceptional conditions. For example, if a server is not responding, the aglet will go on to other servers first and then return to that server later.

- On each database server, the aglet can be engaged in heavy data retrieval without suffering from network latency, since the database is accessed locally at a very high bandwidth.
- The possibly mobile client can be disconnected from the network while the information gathering takes place. The aglet will return when the client is eventually reconnected.
- The database structure and server topology are not revealed to the network.

Web Site Monitor

A Web browser can dispatch an aglet to a Web site. The aglet can perform a local high-bandwidth search among the documents at that site or it can monitor specific documents for updates. When a document is updated the aglet can either send an email to its owner or return to its origin.

- The client can search and monitor the Web even while it is disconnected, and monitoring of Web sites does not require a continuous network connection.
- For Web sites, it means that interested readers are automatically and instantaneously notified of important updates.
- A Web-crawling aglet can move between Web sites to perform high-bandwidth local searching or index building.

Distributed Meeting Scheduling

An aglet can be dispatched by a client to arrange a meeting. The agent will use an itinerary to visit other hosts (meeting participants) in order to negotiate an acceptable meeting schedule. At each host it will accommodate the host's calendar system. When negotiation has finished, the aglet will notify all participants of the meeting time and place.

- The meeting scheduling takes place in a truly heterogeneous and distributed fashion, with no central calendar system.
- Multiple mobile clients can have meetings arranged this way, since not all participants need to be connected to the network at the same time.
- The meeting-scheduling aglet acts like an electronic secretary.

Auction

A client can dispatch an aglet that offers an item for sale. Other clients can dispatch aglets in search of specific items. When such aglets meet, they may bargain. "Smart" aglets may make a good bargain but less clever aglets may make less good bargains.

- There is no central auction or marketplace server. Commerce aglets may meet on any host.
- There is no fixed algorithm for accepting or rejecting a bid. It is all up to the individually programmed aglets.

8 Related Work

The object model used in Aglets is derived from concurrent object-oriented systems, such as ABCL/1 [13] and ConcurrentSmalltalk [14]. Because Aglets have both autonomy and message handing, this object model allows concurrency inside an aglet.

Basic discussions of code mobility and related issues can be found in Distributed Smalltalk [15]. The key difference here is that Java [1] can have multiple versions of classes with the same name using the ClassLoader architecture, whereas Smalltalk assumes a single name space for classes.

TeleScript [12] is the first production-level agent system, and provides an authentication and authorization mechanism as well as a reference monitor for host protection. Agent TCL [8] is a transportable agent system based on TCL [9]. These agent systems support a process migration of an agent, and thus require a special engine or interpreter to run the system.

In contrast to these systems, Aglets never migrate a process or threads. Instead, we adopted the method callback model, which does not require any modification of the Java virtual machine. Therefore, Aglets can run on any platform that supports Java.

Java-To-Go [cite] and MOLE [10], like Aglets, are mobile agent systems based on Java. MALS [19] is also based on Java but, unlike Aglets, it has a preprocessor to simulate a process migration. Because this preprocessor generates Java source code, an agent program can run under any standard Java virtual machines.

9 Conclusion

To summarize, aglets provide in a unique and powerful way a new and viable paradigm that unifies:

- mobile and stationary objects,
- object-passing, message-passing, and data-passing,
- autonomous and passive objects,
- asynchronous and synchronous processing,

– local and remote objects,
– disconnected and connected operations, and
– parallel and sequential execution.

Our top priority is to make aglets ubiquitous. For that purpose, we have made Aglets freely downloadable at sites in the U.S. and Japan so that we can receive technical feedback and application ideas from public users. Since we placed it on our public WWW homepage [4] in July 1996, more than 10,000 people downloaded the Aglets Workbench.

We carefully designed Aglets APIs to be independent of their implementation so that other parties can build their own implementations. Another important design point is the Aglets migration model. We decided to adopt a method callback model, because it allows us to design and implement aglets without any modifications to its underlying language system, while a process migration model might require less programming effort on dispatching. The ATP/0.1 specification is a public document with unrestricted distribution. The protocol has been modeled over HTTP, and proposes a standard way of transporting agents independently of any particular implementation.

Security is of paramount importance to users of mobile agents. Without any consideration of security, a mobile agent system could become a breeding ground for software viruses. To prevent this and other security threats, we developed the Aglets Security Model as the first step towards the provision of a complete security framework. Next, we will design and implement the Aglets Security API, based on the model developed, to provide the required mechanisms to protect against the pertinent vulnerabilities of mobile agents. Thus we want to achieve our goal to establish aglets in a commercial world.

Although the idea of mobile agents is not totally new, practical systems have been emerging only very recently, because of the influence of Java. It would seem possible to use such systems for building new network-oriented applications, but this has yet to be proved. Therefore, our next step will be to use Aglets in real applications.

Acknowledgments

First, our thanks goes to the Aglets Team, without whom aglets would not have made it. They are (in alphabetical order) Dr. Daniel Chang, Teajae Lee, Yoshiaki Mima, Dr. Yuichi Nakamura, Kazuyuki Tsuda, Gaku Yamamoto, and Dr. Aridor Yariv.

References

1. Sun Microsystems Inc.: Java. http://java.sun.com
2. Sun Microsystems Inc.: Jeeves - a Java-Powered Internet Server and Framework. http://www.javasoft.com/products/jeeves/

3. IBM Corp.: Java Aglet Application Programming Interface (J-AAPI) White Paper – Draft. http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html
4. IBM Corp.: The Aglets homepage. http://www.trl.ibm.co.jp/aglets/
5. Sun Microsystems Inc.: Object Serialization. http://www.javasoft.com/products/JDK/1.1/docs/guide/serialization/
6. IBM Corp.: Agent Transfer Protocol ATP/0.1 Draft. http://www.trl.ibm.co.jp/atp/atp.htm
7. Sun Microsystems Inc.: The JavaSecurity API and Digital Signatures. http://www.javasoft.com/products/JDK/1.1/docs/guide/security/
8. Gray, R.: Agent Tcl: A flexible and secure mobile-agent system. Fourth Annual Tcl/Tk/Workshop (TCL 96), 1996
9. Ousterhout, J.: Tcl and the Tk Toolkit. Addison-Wesley, 1994
10. Strasser, M., Baumann, J., Radouniklis, N., and Hohl, F.: MOLE - A Java Based Mobile Agent System. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html
11. Harrison, C., Chess, D., and Kershenbaum, A.: Mobile Agent: Are they a good idea? Research Report RC19887, IBM Research Division, 1995
12. Tardo, J. and Valente, L.: Mobile Agent Security and TeleScript. IEEE CompCon, 1996
13. Yonezawa, A. and Tokoro, M.: Object-Oriented Concurrent Programming. MIT Press
14. Yokote, Y. and Tokoro, M.: Experience and Evolution of ConcurrentSmalltalk. OOPSLA '87 Proceedings
15. Bennet, J.: The Design and Implementation of Distributed Smalltalk. OOPSLA'87 Proceedings
16. Dean, D., Felten, D., and Wallach, D.: Java Security: From HotJava to Netscape and Beyond. IEEE Symposium on Security and Privacy, May 1996
17. Genesereth, M. and Ketchpel, S.: Software Agents. Communications of the ACM, Volume 37, Number 7, July 1994
18. Li, W. and Messerschmitt, D.: Java-To-Go. http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go/
19. Katsuno, Y., Nakahara, M., and Tokoro, M.: Agent-Oriented Language, MALS. Computer Software, Vol.14, No. 2, 1997, JSSST

This article was processed using the LaTeX macro package with LLNCS style