Security Issues in an Active Documents Application based on Java and Jini

Matei Ciobanu Morogan (matei@dsv.su.se)
Fredrik Kilander (fk@dsv.su.se)
K2LAB
Dpt. of Computing and Systems Sciences
Stockholm University

November 3, 1999

Abstract

The Jima infrastructure for *active documents* is built as a system of mobile agents based on Java and Jini. Using a meeting scenario application as catalyst, a set of selected security issues are identified. The Extended Java Security Platform is briefly displayed and design solutions for three security issues (mobility servers, Jini service objects and enforcing distributed routing decisions) are discussed in detail.

1 Introduction

Instant access to resources and generous connectivity across any form of natural barrier appears to be crucial properties in the concept of ubiquitous computing. However, such unlimited freedom was never practical even in the days when there were only human users in the computer networks.

Today, access and connectivity is still needed by users, but the humans are now in the company of electronic counterparts. Popularly known as agents, these software robots perform serious tasks and make sensitive decisions. No more than we want to see the mechanized Golems of science fiction turned into instruments of abuse do we wish to jeopardize the integrity of their soft siblings. Security is an issue more than ever; a careful balancing act between need and deed.

In this paper we review security considerations in a system infrastructure for mobile software agents operating in a local area network. In general terms the infrastructure supports a system of *active documents*. Active documents are aware, proactive and social participants in communication rather than inanimate frames of stored data. To implement this, documents consists of mobile agents with plans, purpose and knowledge of the payload and the organization in which it plays a role. In the presence of an agent, dynamic *presentation* is easily generalized to *interaction*. At that point the display of a document's content becomes an instance in a general framework of distributed applications.

Software agents depend on an infrastructure in which to act and we here identify these primary components:

- Human users
- Agents
- System resources

Human users are those who use the system for communication. Agents in the system carry the information to be communicated. To facilitate timely and accurate delivery agents and users must employ various system resources (which may be other agents).

There are few novel security issues at stake here since most of them can be formulated in terms of access rights to resources. However, the dynamic nature of the system emphasizes the need to be attentive to the current security context. This context emerges from the system's ambition to reach the user across a number of terminal services like mobile telephone, pop-up dialogue etc, and the mobility of the user. For example, although a message is cleared for delivery to user J's personal computer both J and the computer may at that time be in an unsuitable location.

Other important issues involves protection of the payload and the payload's integrity, authorization of users, agents and devices, and non-repudiation of human interaction with an active document, to name a few.

2 Overview of the Jima infrastructure

The following discussion assumes that the reader is familiar with general Java programming, Jini basics like clients, servers and lookup, and mobile agent technology.

2.1 Components

The Jima infrastructure consists of a number of Jini services. Several of these services are also Jini clients. A rough categorization of the services and clients results in the following:

- Jini lookup servers
- Mobility servers
- Personal proxies (aka personal routers)
- Active documents
- Localizer services
- Terminal services
- Subsystem services

2.1.1 Jini Lookup Servers

Jini lookup servers act as a object brokers between clients and services (Java programs). A client requesting a service queries the lookup servers and recieves one or more service objects in return. The service object is then used to communicate with the service. The advantage of lookup servers and service objects in a dynamic environment is that the client does not need to know anything beyond the Jini protocol and the service's Java interface.

A collection of Jini lookup servers and services is usually restricted to a local area network and this is also our present scope.

2.1.2 Mobility Servers

A mobility server, in our framework, is an execution environment (a Java Virtual Machine) in which the code of a mobile agent may inject itself for execution. Mobility servers appear to agents as Jini services.

2.1.3 Personal Proxies

A personal proxy is a service which fulfills two important tasks. The first is to persistenly represent the user in the network and give the user an address and a net presence. The second task is that of routing, i.e. the ability to find and establish a communication with the user.

Active documents approach the proxy to reach the user. The document must negotiate with the proxy for a level of access to the user. The proxy and the document together form a decision by weighting together user preferences, document importance, the time of day and the current or expected reachability of the user. In some cases the document is only allowed to hand over a short text abstract, in other cases it may talk directly to the user across a sophisticated graphical user interface of its own design.

2.1.4 Active Documents

Active documents consists of agents and payload. The agent part of the document is there to ensure the proper management of the document. For example, that the correct recipients are notified and encouraged to respond in a timely fashion; that different renditions of the content are available and that workflow protocol and business rules are observed.

Mobility for an active document can be exploited in several ways. One possibility is to allow the document to physically migrate onto a mobile platform (like a laptop computer) and continue to execute there even though the computer has been disconnected from the local network.

The payload in an active document may be many things. An interesting challenge is to define the payload as one or several other agents, each specialized to perform a certain duty once the outer envelope has fulfilled a necessary objective, for example transportation of the document to its destination.

Active documents act as Jini clients of proxy servers, mobility servers and terminal services (see below).

2.1.5 Localizer Services

Localizer services are sensors, interpreters and databases which together forms a system for tracking the physical position of people and devices. There are many debatable issues with this, so we will immediately recognize that localization information is restricted. It is important to include devices as they must be localized too. We do not specify the existence of a particular localization service, nor do we require one, but it is easier if there is one.

2.1.6 Terminal Services

Terminal artefacts are physical user devices, such as video displays, mice, keyboards, live-boards, printers etc. These devices provide their usual service, i.e. input from and output to users.

A *terminal service* is a Jini service which acts as a front-end for a terminal artefact. In most cases, the service object exports the generic capacity of the device. It may also extend the services of one or more *virtual devices*. These may be provided by additional capacity implemented in software. A special class of virtual devices are *remote devices* which cannot be immediately manipulated. For example, a terminal service for a pager may only be able to accept a short text message.

It should be clear that the Jini terminal services form a layer of abstraction over the total collection of hardware in the network. The purpose of this layer is to enable the agents (primarily documents and proxies) to interact with the hardware and the human users through a defined set of operations.

2.1.7 Subsystem Services

This is a set of Jini services which act as a general library to the clients.

2.2 Interactions

An active document makes contact with a proxy by obtaining the proxy's service object from a Jini lookup server. Through the service object, the document negotiates with the proxy for a way to interact with the user.

The proxy, in turn, must then find a way to communicate with the user. To do this it contacts one or more terminal services which represent devices close to the user. The terminal services of choice (and the devices they represent) fall into two rough categories:

- devices with an expected proximity to the user, and
- devices with random proximity to the user.

Devices with an expected proximity are those which the user is known to patronize, i.e. a certain PDA or laptop computer, a specific office computer, a mobile or stationary telephone etc.

Devices with random proximity are those which by chance happen to be close to the user at some time. For example, when the user walks into a conference room the equipment there represents potential contact points. To exploit this random proximity it must be possible to ascertain the position of users and devices.

Since devices are increasingly mobile their identity and location is just as important as that of their users. Coupled to this is the possibility of public terminal services. For example, a user of a laptop computer may be willing to export a public terminal service for a virtual device which implementes a small text window while reserving full GUI interaction for the user's personal proxy. To extend the conference room example above, the availability of possible means for contacting the user is extended to include not only the stationary equipment in the room, but also the public terminal services in mobile devices brought by other users.

3 A Meeting Scenario

To give a more concrete view of an active document system we use a meeting scenario.

The model shown in figure 1 contains the basic elements which relate to the concept of a meeting between people (counterclockwise from the top): people *(group members, called, present)*, coordinates *(place, time, archive)*, resources *(agenda, material, minutes)* and messages *(announcement, reminder, rescheduling, cancellation)*. At the center, an active document represents a particular meeting by relating instances of the other elements. We assume for the moment that we can create a suitable active document for each element.

Some documents, like the agenda, can only be updated by a restricted number of people, but may be readable by all group members. Material may be submitted by several people, or perhaps passed before a moderator. The meeting document agent should enforce these restrictions.

The meeting should also ensure that messages are sent to the participants. Announcements, reminders, cancellations and reschedulings are all important, especially when the time remaining to the meeting diminishes.

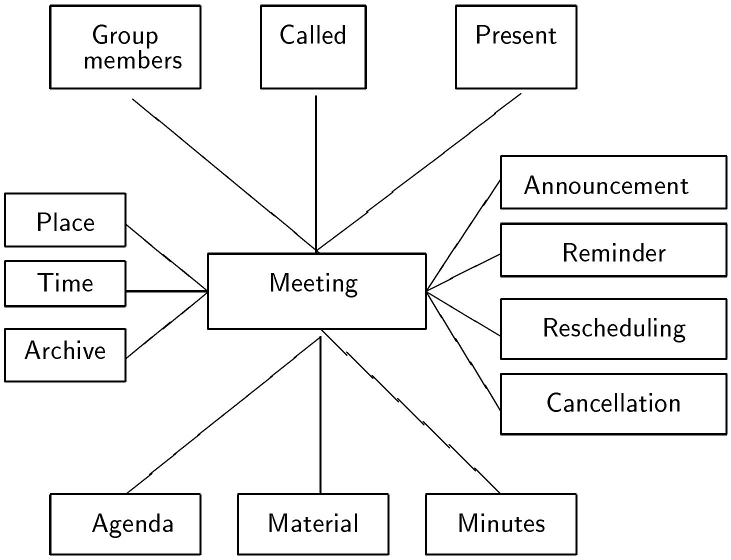

Figure 1: A simple meeting model.

The materials sent to the meeting may be required reading for the participants. In this case, a copy should be sent to those called and each copy falls under the same delivery restrictions as the messages. Furthermore, the copy may require the user to interact, e.g. actively confirm that a certain text has been read.

During the meeting, certain material may be required to be physically available to the participants. In this case, resources such as printers, faxes, projectors and software associated with the location must be located and utilised by the documents. For example, the final agenda is printed, slide presentations are downloaded into or accessable from a multimedia system or the participant's mobile computers are given the necessary access rights.

Finally, before the meeting can archive itself, the minutes should be approved and signed by the appointed members. A copy of the minutes should also be sent to those who were present (or called).

3.1 Security Issues

The Jima system needs a security structure to be able to function properly. In the above scenario, for example, all the active documents need some form of protection to prevent rogue entities from interfering with the meeting process.

The documents should be able to protect the data they contain during transit. For example, an active document going from the meeting document to a participant could be either an announcement, a reminder, a cancellation or a rescheduling message. These messages should only be readable by their recipient, since the details of the meeting may be confidential. To achieve this, the active document could use an encryption function before leaving the originating host and encrypt its content, so that it is only readable by the intended recipient.

Once a participant (or a proxy of a participant) receives a message from the meeting document it should be able to verify its validity, namely that it really does originate from the meeting document and not from somebody claiming to be the meeting document. In the same way, the meeting document should be able to verify the validity of the messages it gets from the participants. An active document should therefore be able to request and receive a validity check for the content from the originator. This could be in the form of a digital signature on the content or

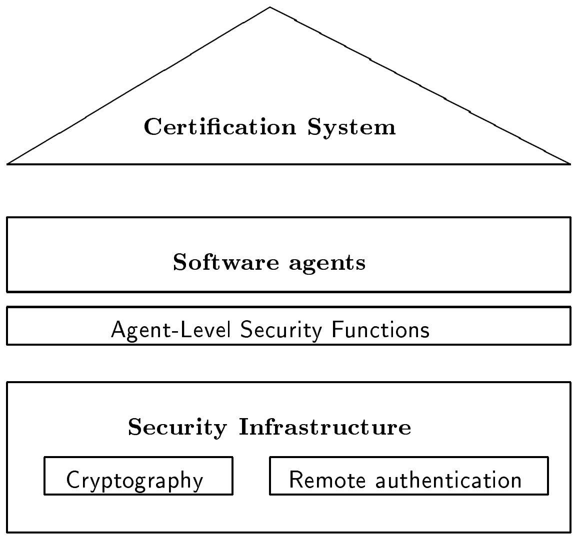

Figure 2: Java-based security platform.

it could be some form of capability that the originator gives the active document - the capability to cancel a meeting, for example.

Therefore, the documents should have access to security mechanisms that enforce strong authentication of peers and enveloping of data. A system to grant and enforce capabilities is also useful. To provide these services the Extended Java Security Platform is used.

4 The Extended Java Security Platform

Java has APIs for most basic security functions. The core implementation contains several packages that deal with security:

- java.security – here are located classes and interfaces for:
  - key generation, handling and storing
  - generation of message digests
  - permissions
  - signatures
- java.security.acl – interfaces for access control lists
- java.security.cert – certificate handling
- java.security.interfaces – interfaces for generating RSA and DSA keys
- java.security.spec – classes and interfaces for key specifications and algorithm parameter specifications

We use these packages in a security platform that is based on Java. The platform is designed to provide the different security services needed by mobile agents in general and active documents in particular. This platform is structured in three layers (figure 2).

The lower layer consists of the basic security mechanisms, like encryption, signature and key generation. This layer implements the interfaces specified by Java and also adds some other classes that are needed in the higher-level protocols.

In order to use asymmetric cryptography for authentication and enveloping of data, a certification system is needed. The certification system is a tree-like structure of certification servers with a common root. The root, which is the highest certification authority, signs the certificates of all underlying certification servers. The clients can access the certification servers in order to get their certificates signed and also to verify other certificates. The platform contains a certification system that uses certificates that comply with the X509 standard. The certification servers may be set up as Jini services.

The middle layer in our platform consists of the services used by the clients directly. These services include strong authentication of remote entities, encapsulation of messages, a system for authorisation and delegation of authority among entities. The services rely on the underlying security mechanisms and also use the certification system for authentication.

5 Specific Security Issues in the Jima Architecture

In this section we review some issues which we view as central. Much of the basic security technology is available in standard Java, but we do not evaluate any particular implementation here. We use the Jima architecture to reduce the complexity of the general case and begin by identifying three basic questions:

1. When can a mobility server confidently execute the code of a mobile agent?
2. When can a Jini client confidently call methods in a Jini service object?
3. How can a personal proxy confidently mediate the interaction between an active document and the user?

5.1 Mobility Servers

The first question depends on how mobility servers and agents are implemented. To answer it, one must envision a solution for mobile code. Since we are using Java, we can propose the sandboxed model used for applets, where the code is restricted by code inspection and the environment to a defined set of actions. This restriction is not convenient for a system which is basically a distributed application, and does not exclude the possibility of a rouge object performing a denial-of-service attack against CPU and memory from within the sandbox. We could design a configurable sandbox where the agent must negotiate for resources and can be preempted by the mobility server, but this adds another layer of complexity to an already sophisticated system.

For Jima we initially employ a mobility server technology which is as simple as possible. A mobility server is created as an RMI service. A Java object moves to the server's JVM by calling a remote server method with itself as one of the parameters. The mobility server allocates a new thread of execution and assigns it to the specified callback in the object.

We want the mobility server to examine the foreign object and to reject or approve it before allowing it to execute. This implies that such an inspection must be possible without calling methods in the object. In the simplest case, the mobility server calls a function $f(O)$ with the object as the argument and from the result determines a go or no-go for execution.

We propose that the function $f()$ extracts a certificate from the object. If the certificate is accepted by the mobility server's configuration, the public key it contains is used to check the authenticity of the object's signature. If the signature verifies, the object is allowed to execute.

The function $f()$ could accept a reference to a standard Java object, in which case the signature must be retrievable as a public member field. An interesting issue here is how to access the object's certificate and signature without allowing the object, whose origin is yet unknown, to run. The solution is to use a container object that holds the object to be transmitted, its signature and its certificate:

$$h(O) \Rightarrow g(O, (s(O) \to S), C) \to O_w$$

Instead of passing itself directly, the agent's application-level code applies function $h()$ on object O (itself). Function $h()$ immediately translates to $g(O, s(O), C)$ which adds a signature S and a certificate C to produce the wire object O_w.

The mobility server application-level code applies $f(O_w)$ and internally retrieves the original object O, the signature S and the certificate C. If the signature is accepted $f()$ returns O, otherwise the empty set (null).

$$f(O_w) \to \{O \mid \emptyset\}$$

At this point, the mobility server can store the signature of the requesting object in a safe place. From the certificate of the object it is possible to extract the identity of the issuer. If the object misbehaves there will be an audit record for post-mortem analysis.

The agent then, should not indiscriminantly trust service claiming to be mobility server. It could be someone posing as a mobility service in order to gain access to the data that the agent carries. As a consequence, the mobility server must be prepared to yield its own certificate to the agent for inspection. The agent may then encrypt the object component of the wire object O_w with the mobility server's public key (found in the certificate).

5.2 Jini Service Objects

A Jini client accesses a service through a service object. However, before calling methods in the service object, the client may want to verify that the service is authorised. For example, in a local area network with several printer services, it would not be complicated to add a printer service which both prints and sends copies of everything to a spy. A certified service object does not prevent bad behaviour, it just verifies that distributed software components are approved by the organization. It is at the client's latitude to choose either to trust or not to trust the issuer of an authorisation. In the printer example above, user A may trust a printer that has the authorisation of the system manager, but may choose not to trust a printer authorised by user B. User B, on the other hand, will trust both printers.

In the previous section we argued for a technique with signed objects. This is not generally convenient in Jini where service objects are registered and found through registrars (lookup servers). Storing a composite object is possible, but if we want to extend the same functionality to non-discriminating clients each service must register and maintain leases for two different objects. A better solution would be if the native object could be used and inspected directly.

In the Jini registrar, each service stores a reference to its service object under a set of attributes. These attributes could be used to store a certificate and the service object's signature. A security-aware client chooses a service and retrieves the service object along with the certificate, the signature and the other attributes.

Before calling methods in the service object, the client can check the validity of the certificate and use its public key to verify the service object's signature.

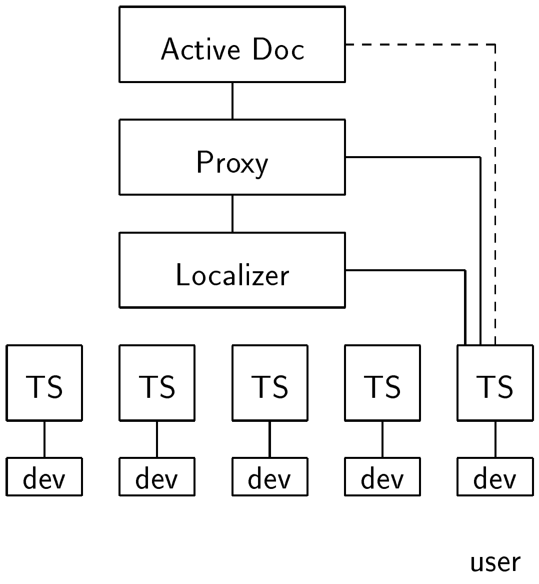

Figure 3: How an active document reaches the user.

5.3 Enforcing Proxy Routing

In the Jima architecture, an active document attempts to communicate with a user. It must, however, approach the user's proxy to do so. The proxy and the document negotiate for the best way to reach the user, but the proxy must have the final word on how, where and when. If it does not, its usefulness as a mediatior will very likely soon be overrun by intelligent and aggressive spam (horrible thought).

In figure 3 we show a conceptual sketch of the primary components in the interaction between an active document and a user. First, the document contacts the proxy and presents a request. The proxy queries the localizer service (or employs internal knowledge) to find a terminal service (TS) close to the user. If the proxy allows the document to approach the user it reveals the identity of the terminal service to the active document which then contacts it directly.

In order to enforce the proxy's decision, we must prevent the document from going directly to the terminal service. We could prevent the document from querying the localizer (and this may be a good restriction for other reasons), but this only holds back knowledge, it does not prevent the document from trying *every* terminal service. Thus, the default policy must be that active documents can not be clients of terminal services, unless they are approved by a proxy. Furthermore, the terminal service may have an even more restricted policy.

To implement this strategy, the proxy makes a best guess at a candidate terminal service and makes a probing query if it is available under the current configuration. If it is, the proxy generates a ticket and gives it to the document.

The ticket is an object that contains the rights granted by the proxy to a given active document. In the most simple case, this could be just the permission to show content of some type to the user. However, there could be situations that require a more specific definition of the permissions granted to the active document. Documents that are considered by the proxy to be very urgent, for example, could be allowed by the terminal service to show up at a time when messages normally would not be let through (i. e. when the user is busy).

The ticket must also contain the identification of the active document it is issued for, so that even if other entities manage to copy it from the network, only its

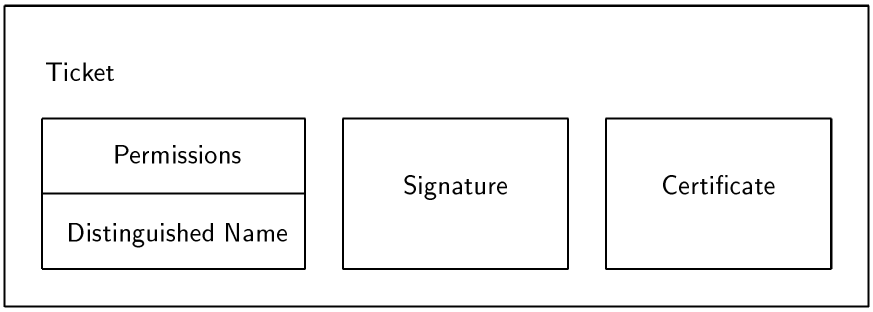

Figure 4: The structure of a ticket.

intended receiver can use it. For this, the *distinguished name* of the active document can be used. The distinguished name can be extracted from the certificate and it has the characteristic of being unique.

The permissions and the distinguished name must be signed by the proxy, and this signature, together with the certificate of proxy, are added to the ticket (figure 4). The ticket is then given to the document together with the terminal service's address. The active document may now approach the terminal service and present the ticket. The ticket expires after a certain time and can only be used once.

When the terminal service receives an active document it takes the ticket and checks the certificate of the issuing proxy service. If the certificate is valid and the terminal service accepts documents from that particular proxy, the signature is verified and the distinguished name of the recipient from the ticket compared with the distinguished name of the document. If everything is correct, the terminal service can read the permissions granted by the proxy to the document and use that information, together with the identity of the issuer and its own policy, to decide whether to allow or deny access for the document.

Thus, tickets allow terminal services to protect themselves from unwanted clients and only allow documents that are approved by proxy servers to go through.

6 Related Work

A nice treatise on mobile agent programming systems by Karnik and Tripathi [11] illuminates many important issues which this paper omits or glances by. For example, the desired properties of mobility servers (execution environments) and the mobile code itself, the namespaces to be used and how names are resolved (into Jini service templates, for example), security issues, agent monitoring and control and fault tolerance.

Yi Cheng [3] has written a comprehensive thesis on security for mobile agents which in describes many of the features we rely on here.

Digital certificates and their usage [7, 4] together with Java security functions [9, 12] are important foundations for our security platform.

The Mobile People Architecture (MPA) [1, 14] embodies many ideas similar to our own, in particular the personal proxy concept. Our approaches differ on the way terminal devices are accessed. The MPA places *application drivers* in the proxy while our infrastructure assume that terminal access is distributed. Another difference is that MPA emphasizes communication between persons, while we assume communication mediated by active documents.

The dynamic availability of terminal services in our infrastructure may be a useful element in the development of collaborative environments like *Information*

Workspaces [8] at Stanford University, or virtual workspaces [13, 15, 5]. In particular, we expect that transparent blending of mobile and stationary equipment will become increasingly important.

We choose to indicate only a few sources for information about Jini [10, 2, 6] as this is a developing area.

7 Summary

We have described an infrastructure based on Java and Jini for active documents in the form of mobile agents and identified a number of security issues in this environment. Using a meeting application we have identified several places where security is important. Three central topics (mobility servers, Jini service objects and enforcing distributed routing decisions) were investigated and design solutions found using digital certificates, signatures and tickets.

References

[1] Guido Appenzeller, Kevin Lai, Petros Maniatis, Mema Roussopoulos, Edward Swierk, Xinhua Zhao, and Mary Baker. The mobile people architecture. Technical Report CSL-TR-99-777, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, California 94305-9040, January 1999.

[2] Ken Arnold, Bryan O´sullivan, W. Scheifler, Robert, Jim Waldo, and Ann Wollrath. *The Jini (TM) Specification.* Addison-Wesley Publishing Company, 1999. ISBN: 0201616343.

[3] Yi Cheng. A comprehensive security infrastructure for mobile agents. Licentiate thesis 97-013, Department of Computer and Systems Sciences, Stockholm University, 1997.

[4] Alan Davidson. Availability, integrity and consistency in certificate management systems. Licentiate thesis 99-012, Department of Computer and Systems Sciences, Stockholm University, 1999.

[5] C. Davis, Richard, A. Landay, James, Victor Chen, Jonathan Huang, B. Lee, Rebecca, C. Li, Francis, James Lin, B. Morrey III, Charles, Ben Schleimer, N. Price, Morgan, and N. Schilit, Bill. Notepals: Lightweight note sharing by the group, for the group. In *CHI 99 Conference Proceedings, Human Factors in Computing Systems*, pages 338–345, 1999.

[6] W. Edwards, Keith. *Core Jini.* Prentice Hall, 1999. ISBN: 013014469X.

[7] Jalal Feghhi, Jalil Feghhi, and Peter Williams. *Digital Certificates.* Addison-Wesley, 1999. ISBN: 0-201-30980-7.

[8] Interactive workspaces. http://graphics.stanford.edu/projects/iwork, 1999.

[9] Java 2 platform documentation. http://java.sun.com/products/jdk/1.2/docs, 1999.

[10] Jini (tm) connection technology. http://www.sun.com/jini, 1999.

[11] M. Karnik, Neeran and R. Tripathi, Anand. Design issues in mobile agent programming systems. Technical report, Departmen of Computer Science, University of Minnesota, Minneapolis, MN 55455, USA, June 1998.

[12] Scott Oaks. *Java Security.* O'Reilly, 1998. ISBN: 1-56592-403-7.

[13] Jun Rekimoto and Masanori Saitoh. Augmented surfaces: A spatially continuous work space for hybrid computing environments. In *CHI 99 Conference Proceedings, Human Factors in Computing Systems*, pages 378–385, 1999.

[14] Mema Roussopoulos, Petros Maniatis, Edward Swierk, Kevin Lai, Guido Appenzeller, and Mary Baker. Person-level routing in the mobile people architecture. Technical report, Department of Computer Science, Stanford University, Stanford, California 94305, 1999.

[15] Jason Stewart, B. Bederson, Benjamin, and Allison Druin. Single display groupware: A model for co-present collaboration. In *CHI 99 Conference Proceedings, Human Factors in Computing Systems*, pages 286–293, 1999.