Abstract

RoamX provides support for the user-controlled management of a distributed X desktop, potentially spanning multiple administrative domains. Instead of terminating X-based applications and restarting them elsewhere, applications can be suspended and restored selectively on different machines while preserving their context and state. The migration of the GUI is accomplished by an extended version of xmove, a pseudoserver for X window movement, developed originally by Ethan Solomita, James Kempf and Dan Duchamp at the University of Columbia [1]. RoamX applies CORBA, Jini and JavaCards together in its infrastructure, three sophisticated, supplementary, but rather distinct technologies. This paper describes the implementation, deployment, and experiences gained throughout the development of RoamX.

1. Introduction

In the age of the Internet and Ubiquitous Computing [2], users claim to gain access to their applications and data from different localities. People don’t longer want to be bound to single machines. The Sun Ray 1 product line [3] is a prominent example emphasizing this fact. The Hot Desk technology [4] allows users to connect to their private computing session from any terminal using smartcards [5]. Users may disconnect from their current session and reconnect later while preserving their applications’ context. However, this is an all-or-nothing approach. Only whole sessions may be transferred from one display to another. Switching applications between two machines on a selective basis is not possible. Additionally, this approach is limited to a single administrative domain and works in a strictly homogeneous environment only. Sessions cannot be migrated from a Sun Ray workstation to a PC, for example.

RoamX is our approach at supporting desktop mobility between heterogeneous machines, allowing for fine-grained, user-controlled, selective migration of X11-based applications’ GUIs. RoamX is one project in the enclosing field of Internet Roaming which, in turn, is targeted at enhanced support for user mobility in the Internet [6].

The workflow when using RoamX is shown in figure 1, drawing a comparison to a traditional setting without RoamX support. Applications no longer need to be terminated and restarted when moving to a different site. Instead, applications are only suspended, thereby retaining their current state. The information describing the users’ application windows and their current configuration is stored on a JavaCard, a smartcard with an integrated Java virtual machine [7]. When logging on to a remote machine, the application’s GUI is migrated and restored in the same state.
as left at the originating site. It should be noted that the application itself continues to run on the same machine where it has been started initially.

The RoamX infrastructure, as depicted in figure 2, is comprised of three components. The xmove proxy server intercepts X11 protocol connections and routes them to a real X server, enabling the migration of an application’s GUI. The xmove browser is the user interface component. The browser displays the user’s distributed X desktop and allows the migration of X applications via drag-and-drop operations. The xmove broker acts as a mediator between the other two parties. The responsibilities of the components and their interaction will be described in detail in the following section.

As part of a technology study, we decided to apply the most appropriate technologies for each purpose in the realization of RoamX. As implementation language for the browser and the broker, we chose Java because of its platform-independence. Since the xmove proxy server we used as a basis for our implementation is a legacy C application, we had to select an appropriate protocol to talk to the broker. We selected CORBA-IIOP for this purpose [8]. RoamX is an instantiation of a general framework for roaming-aware applications, which is discussed in more detail in a subsequent section. Because of the dynamic nature of such applications, where users are assumed to roam between different administrative domains, the framework employs the Jini technology [9] with its abilities related to impromptu networking.

In the following section, we will describe the RoamX infrastructure in detail. Section 3 outlines implementation details, followed by a discussion of deployment issues in the subsequent section. Finally, our experiences and conclusions will be presented.

2. RoamX Infrastructure

The components of the RoamX infrastructure, as depicted in figure 2, are the xmove proxy server, the xmove broker and the xmove browser. Their responsibilities and interaction is described in detail throughout this section.

2.1. Xmove Proxy Server

The migration of X11 windows is facilitated through the use of an X proxy server. Our proxy server implementation is based on xmove, a pseudoserver for X window movement, developed at the University of Columbia [1]. Applications are started against such an xmove proxy server, intercepting X11 protocol requests and forwarding the communication to a real X11 server. Migrating an application is facilitated by setting up a new X11 connection with the target server, transferring the session context and terminating the original connection. Subsequent X11 requests will be dispatched transparently to the new X11 server.

2.2. Xmove Broker

Newly started applications are registered with an xmove broker. Brokers act as mediator between xmove proxy servers and the users. One vital responsibility of the broker is to control the mapping between the two different administra-
tive domains participating in our application scenario (cf. figure 2). While X applications are part of the UNIX/X domain, roaming users live in a virtual roaming domain. The mapping of a roaming user onto a concrete administrative domain is done via a locally administered domain mapping service. In our case, roaming users are identified by X.509 certificates [10], which have to be mapped onto a UNIX account and a MIT-MAGIC-COOKIE-1 [11] needed to access the X11 server, respectively.

The brokers are the primary contact point for users. As shown in figure 3, multiple brokers may coexist, with each broker serving multiple xmove proxies and multiple users. The xmove brokers are grouped into sets of collaborating instances, forming so-called communities. A community is made available to the users through Jini connection technology, as will be described in a later section. A concrete RoamX installation may be composed of multiple communities.

Each xmove broker announces itself to xmove proxy servers through a standard CORBA Trading Service [12]. Newly started xmove proxy servers can query the Trading Service and register themselves with an xmove broker. A dynamic property is used to implement a simple load balancing scheme. When querying the trader, the registered brokers are called back, returning their current load as a function of the number of connected proxy servers, connected clients, and the load of the broker’s machine.

2.3. Xmove Browser

The user’s interface to RoamX is the xmove browser, a Java Swing [13] application backed by a JavaCard. Users arriving at a remote site insert their JavaCard into a smart-card reader connected to the terminal. This triggers a discovery process, resulting in a set of xmove brokers to be returned, each managing a set of X11 displays. The subset of X11 displays the user currently is allowed access to is displayed in the xmove browser’s GUI. The user may now selectively drag and drop application GUIs inside the browser window from one machine to another, triggering the actual migration of the windows.

3. RoamX Implementation

Having illustrated the RoamX infrastructure and its components, the following subsections will describe their implementation, together with a discussion of experiences gained.

3.1. Integrating CORBA with the Legacy X11 Proxy Server

The xmove proxy server used as basis for our implementation is a legacy C application. It has to be started as a traditional, stand-alone user process after logging on to an X display. Xmove then opens a socket listening for connections by newly started X applications. This socket accepts X protocol requests like a real X server. Applications can be started against the xmove proxy server through a standard X command line option, or by setting the $DISPLAY environment variable appropriately.

The original xmove design has some drawbacks for a setting like RoamX, where users are supposed to change machines dynamically. Firstly, the xmove proxy server is owned by, and therefore under full control of, an ordinary user. Consequently, each user would have to start its own proxy server instance, each consuming its own server number.

Moreover, not only managing server numbers is awkward, access control places another burden on the original design. Access to the xmove proxy server is controlled through the X11 standard MIT-MAGIC-COOKIE-1 authorization scheme. A MIT-MAGIC-COOKIE-1 is a shared secret between the X server and any entity having granted access to the server. The original xmove proxy server simply uses the same cookie as the X server it forwards.

![Figure 3. A set of collaborating xmove brokers, forming a Jini community](image)
requests to. However, logging off from the X server terminates the validity of the cookie. Logging on again, either by the same or a different user, generates a new cookie. Applications suspended previously have to present the current cookie when they are restored at a later time. Proper management of the cookies, allowing applications to outlive single X server instances, was another issue in xmove’s redesign.

Lastly, window management in the original xmove proxy server was intermixed with X protocol requests. Commands like suspending an application window were encapsulated in a bogus X protocol message. Because the owner of the xmove proxy server instance and the user were one and the same person, this approach was acceptable. However, in R*anX, different users may access the proxy server concurrently. For example, one user currently connected to the managed X display may be granted the right to start a new application against xmove. Simultaneously, a different user may send a command to the proxy server, e.g. initiating the migration of one of his windows.

Accordingly, we redesigned the original xmove proxy server as a daemon-style process, incorporating user and cookie management. Furthermore, xmove has been augmented with a CORBA wrapper, separating functionality for window management from X protocol requests.

Writing a separate CORBA wrapper application would be the preferable way in most cases, as this approach leaves the original code untouched. However, the simple design of the original xmove proxy server allowed us to integrate CORBA with the legacy application directly.

Xmove’s original implementation centers around a UNIX select() statement, managing multiple connections with X application clients. On the backend, xmove utilizes a single connection with the X11 server it manages. Incoming requests are dispatched sequentially to the outgoing server connection. Xmove is designed as a single-threaded application and makes heavy use of global data.

The CORBA ORB itself, too, relies on the socket interface in order to accept client connections and to handle incoming requests. Therefore, we had to find a workable solution to integrate CORBA functionality with the existing xmove application design. Two different choices were at hand: non-blocking event handling and introducing multi-threading.

With non-blocking event-handling, two standardized ORB interface functions would have to be used. The first one, work_pending(), checks for outstanding requests while the second, perform_work(), instructs the ORB to handle those requests. While this approach retains the single-threaded design and needs only minor changes in the legacy code, this solution has significant drawbacks. A time-out would have to be added to the original call to select(). The selection of the time-out value is a critical design factor. When the time-out is set too long, poor response times for the CORBA communication would result. Setting the time-out too short otherwise would quickly increase the load.

The second alternative lets the ORB dispatch requests in its own threads. While a multithreaded solution doesn’t suffer from the time-out problem mentioned beforehand, it introduces the need for proper synchronization. X11 protocol requests and CORBA-based method invocations, both accessing xmove’s (global) data structures, may arrive concurrently. Fortunately, in our application scenario, the X11 protocol requests handled by the legacy code can be assumed to be handled rapidly. This allows for an easy synchronization scheme to be implemented. While the new CORBA functionality does fine-grained locking, allowing for a high degree of concurrency between multiple CORBA method invocations, the xmove code called after the select() statement relies on a global lock. In practice, the user rarely perceives delays caused through the newly introduced stringent synchronization of X11 protocol requests.

Conclusion: Naturally, integrating CORBA with legacy code is a difficult task. Introducing multi-threading and proper synchronization are only two aspects. Controlling side-effects of accessing the legacy application’s data is another critical area. These issues are highly application specific and therefore aggravate general support. Writing a separate CORBA wrapper application is the preferable solution in most common cases, not requiring any modification to the legacy code.

3.2. Service Discovery using a Jini-based Broker

The xmove proxy server provides functionality for routing X11 protocol requests to a real X server as well as for window management accessible via CORBA interfaces. While the X11 proxy server port is made available to the user through the $DISPLAY environment variable implicitly, the CORBA servants must be exported by other means. Jini is applied for this purpose, being a technology designed explicitly for impromptu networking, our envisioned usage scenario.

3.2.1. Introduction to Jini. Jini is a technology for spontaneous networking [14]. It tries to combine some vital properties of distributed systems under a set of Java-based application programming interfaces. Jini allows networked devices to setup an impromptu community without the need for device specific drivers to be installed in advance while being resilient to failures. Jini services are offered to clients in a manner that allows clients to download the code needed
to talk to the service dynamically. A set of services available locally forms a Jini community. Among the basic abstractions of Jini are Discovery, Lookup, and Leasing.

Discovery is the process of finding a Jini community. Jini communities are identified by a symbolic name, allowing to establish groups of similar services. The result of the discovery process is a set of so-called lookup services. A lookup service essentially is a directory service, holding representatives of Jini services in the form of proxies. These proxies are downloaded into the client dynamically. This way, the driver for the service doesn’t need to be known to the client in advance. Querying the directory service is termed Lookup in Jini speech.

Leases are the Jini concept to handle failures of any kind. Consumers of a service are forced to first apply for a lease. Leases are granted for a limited time only. If a consumer wants to stick to the service, he has to reapply the lease periodically. Therefore, due to network failures or ill-behaved clients forgetting to release their link to a service, resources are blocked at most for the duration of one lease.

3.2.2. Implementation of the Xmove Broker. Jini technology currently is available directly through Java libraries only. However, Sun emphasizes that Jini is applicable to arbitrary smart devices, even if they do not include their own Java virtual machine. In our case, the X11 display, represented by its managing xmove proxy server, takes on the role of the smart device. In order to make the xmove proxy server Jini-aware, we augmented the xmove broker component to act as a Jini service. The functionality the xmove broker Jini service provides mirrors the one of the CORBA interfaces of the xmove proxy server. As has been described beforehand, an additional responsibility of the broker is the mapping of the identity of roaming users accessing the service onto local accounts.

Being a well-behaved Jini service places some responsibilities on the service implementation [15]. Among them are guidelines on how to do multicast discovery of Lookup Services and rules for lease management. Much of this functionality is encapsulated in helper classes provided by the Jini libraries. Writing a Jini service is greatly simplified through the use of those utility classes. However, the use of these libraries is limited to the Java programming language.

In our application scenario, as is the case in any setting where the concrete Jini service provider cannot itself be implemented in Java, third party delegation has to be applied. The xmove proxy server, being the Jini service provider, delegates all the tasks making up a well-behaved Jini service to the xmove broker. The only responsibility left for the wrapped C-based xmove proxy server is to implement a periodical heartbeat indicating its availability to the broker. The flexibility of Jini allows the exported interfaces to be implemented either by the delegate or directly by the wrapped service. The communication protocol used in the service implementation can be chosen freely. In our case, the exported interfaces are CORBA IIOP based.

Conclusion: Jini is a technology for impromptu networking. In application scenarios where clients want to explore their environment for available services dynamically, Jini provides a pragmatic approach. Although the Jini libraries are Java-based only, implementing a Jini service using a different programming language can be achieved easily, especially when relying on third party delegation.

Figure 4. Overview of the JavaCard Roaming Framework
3.3. JavaCards as Multi-Application Access Controller

X11 displays are managed by xmove proxy servers allowing for dynamic window migration. The proxy servers are exported as Jini services by an xmove broker and made accessible from the outside via CORBA interfaces. Those interfaces are utilized by the user interface component of the $R_{Roam}X$ mobile X desktop. JavaCards are employed to realize the hardware part of our user interface component. Those Java-enhanced smartcards allow multiple card applets, so-called cardlets, to coexist on a single card. One of the cardlets forms the software part of the $R_{Roam}X$ user interface component. Both parts together are one concrete instantiation of a general framework we are developing in the context of our Internet Roaming research project [6].

An overview of this JavaCard roaming framework is shown in figure 4. The JavaCard acts as a multi-application access controller. Each roaming-aware application uses a special kind of applet-duo to implement its functionality.

This applet-duo consists of two components: an on-card applet called a cardlet, and an off-card proxy. The on-card part is installed on the JavaCard. This cardlet stores relevant application specific state when the user roams to a different site. Furthermore, the cardlet provides the URL of the corresponding off-card part of the applet-duo.

The off-card part, which may be downloaded dynamically via HTTP when arriving at a remote site, serves as a kind of proxy for the cardlet. Only the off-card proxy, hosted by a Java virtual machine on the remote machine, can communicate directly with its surrounding environment. The cardlet itself can only contact its off-card counterpart via a serial cable connecting the card reader with the host environment.

When booting such roaming-aware applications, the off-card proxy discovers its environment through Jini lookup. This way, potentially different service implementations are made available to the roaming user in a dynamical manner, and without the need for manual customization.

Conclusion: JavaCards offer a convenient medium for supporting roaming-aware applications. As a special kind of smartcard, they are perfectly suited for identification purposes. Furthermore, because several applets can coexist, a single card can be applied for multiple roaming-aware applications concurrently. Being programmable in Java is another advantage, despite severe restrictions in terms of language and library features available. However, their small memory and process power clearly limits their applicability.

4. Deployment

The $R_{Roam}X$ implementation employs a wealth of different technologies. Each technology adds its own overhead in terms of installation, administration, and resource usage. In this section, we summarize the requirements a $R_{Roam}X$ installation places on the underlying hard- and software infrastructure. Statistical values and measurements are shown in table 1 and table 2. The columns headed memory consumption in table 2 are derived from the top utility available in Linux and the Process Viewer pview under Windows NT, respectively. It should be noted that the given values give a rough approximation of the real resource consumption only.

The xmove proxy server uses a C++ ORB. This requires some shared libraries to be installed on every machine hosting an xmove proxy server instance. We employed MICO [16] as well as ORBacus [17] in our implementation, enabling us to compare two different CORBA implementations. The static resource usage of the ORB alone is approximately 4.9 MB (MICO), respectively 5.6 MB (ORBacus).

<table>
<thead>
<tr>
<th>Table 1. Installation requirements of the $R_{Roam}X$ infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product / File</strong></td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>MICO C++ 2.3.1 ORB</td>
</tr>
<tr>
<td>Services</td>
</tr>
<tr>
<td>ORBacus C++ 4.0 ORB</td>
</tr>
<tr>
<td>Name Service</td>
</tr>
<tr>
<td>ORBacus Java 4.0 ORB</td>
</tr>
<tr>
<td>Name Service</td>
</tr>
<tr>
<td>Trading Service</td>
</tr>
<tr>
<td>Jini 1.1 Alpha core</td>
</tr>
<tr>
<td>extensions</td>
</tr>
<tr>
<td>tools (HTTP server)</td>
</tr>
<tr>
<td>Lookup Service (LS)</td>
</tr>
<tr>
<td>LS client-side</td>
</tr>
<tr>
<td>xmove proxy original version</td>
</tr>
<tr>
<td>$R_{Roam}X$ version</td>
</tr>
<tr>
<td>xmove broker</td>
</tr>
<tr>
<td>xmove browser off-card proxy</td>
</tr>
<tr>
<td>on-card applet</td>
</tr>
<tr>
<td>JavaCard Roaming Framework</td>
</tr>
</tbody>
</table>

The difference between the resource consumption of the original xmove proxy server compared to the CORBA-enhanced $R_{Roam}X$ variant is severe. While the process image of the original xmove proxy is reported to use 940 KB on a Linux box, the $R_{Roam}X$ versions need 7756 KB using MICO and 6708 KB using ORBacus. The
big difference predominantly results from the shared CORBA libraries being mapped into the address space. However, the private areas of both $RoamX$ versions consume approximately 1.8 MB more memory than the original version. Again, this is mostly because of CORBA and only in parts a result of the added functionality.

The xmove broker is implemented in Java and makes use of the Jini libraries and CORBA. The classes making up the broker are packaged into a (compressed) jar-file of 125 KB. They are dependent on the Jini core (26 KB) and extensions (117 KB) as well as on the ORBacus Java ORB (2.7 MB). A running broker’s virtual address space consumes about 11.5 MB. It should be noted that at the time of measurement, the broker consisted of 21 threads, mostly managed under the hood by the Jini and CORBA runtime libraries. Each thread adds a significant amount of stack space to the image size.

The off-card part of the user interface component, the xmove browser, is composed of the JavaCard Roaming Environment (78 KB) and the off-card proxy (120 KB). Additionally, it makes use of the Jini libraries and the ORBacus Java ORB. The process image, including the Swing-based GUI, is reported to use approximately 21 MB on a Windows NT machine. Apparently, Swing adds significantly to the image’s size.

A complete $RoamX$ installation furthermore requires some infrastructure to be up and running. For example, Jini relies on at least one running instance of the Lookup Service. On Windows NT, the Lookup Service’s process consumes about 8.5 MB of memory. Additionally, the Lookup Service as well as any other Jini service, including the xmove broker, needs a HTTP server to be setup for clients to be able to download the service’s proxies dynamically. The Jini distribution comes with a small, Java-based HTTP server. However, if applied out-of-the-box, each instance runs in its own virtual machine, consuming about 7 MB for the first and another 5.8 MB for each additional instance. Still another requirement for Jini to work properly is the Java RMI Activation System Daemon to run, using further 14 MB of memory. Lastly, we need a CORBA Trading Service. We employed the Java version coming with ORBacus, consuming additional 10 MB of main memory.

5. Experiences and Conclusions

We described $RoamX$, an infrastructure supporting mobile users to migrate the GUI of X11-based applications between displays of heterogeneous workstations. Applications are no longer terminated and restarted, but instead suspended in their current context when closing an X-session. After logging on to a different X display, the user can restore his distributed desktop selectively. Beyond pure window migration, the $RoamX$ infrastructure relieves users from the need for administrative knowledge. The appliance of the Jini Connection Technology enables the automatic examination of the current environment. Equipping users with JavaCards not only makes information about their distributed desktop available locally. Particularly, it allows users living in a virtual roaming administrative domain to be identified in a secure and safe manner. Accounts in the roaming domain are mapped onto concrete accounts in the UNIX/X domain by components of the $RoamX$ infrastructure.

The implementation of $RoamX$ makes use of xmove, an X11 proxy server developed at the University of Columbia [1]. Xmove is a legacy C application we adapted to act as a daemon process, managing X applications of multiple users. Furthermore, we introduced CORBA interfaces for window management functionality.

The raw xmove proxy server has been wrapped by a Jini service, called the xmove broker, adding functionality for mapping accounts in the virtual administrative roaming

<table>
<thead>
<tr>
<th>Process</th>
<th>Type</th>
<th>Total [KB]</th>
<th>Share [KB]</th>
<th>Private [KB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmove proxy original version</td>
<td>Linux native executable</td>
<td>940</td>
<td>700</td>
<td>240</td>
</tr>
<tr>
<td>xmove proxy $RoamX$ version (MICO)</td>
<td>Linux native executable</td>
<td>7756</td>
<td>5728</td>
<td>2028</td>
</tr>
<tr>
<td>xmove proxy $RoamX$ version (ORBacus)</td>
<td>Linux native executable</td>
<td>6708</td>
<td>4612</td>
<td>2096</td>
</tr>
<tr>
<td>HTTP server</td>
<td>Java VM (Windows NT)</td>
<td>7064</td>
<td>1180</td>
<td>5884</td>
</tr>
<tr>
<td>Activation Daemon (rmid)</td>
<td>Windows NT native executable</td>
<td>14096</td>
<td>1248</td>
<td>12848</td>
</tr>
<tr>
<td>Lookup Service</td>
<td>Java VM (Windows NT)</td>
<td>8560</td>
<td>1248</td>
<td>7312</td>
</tr>
<tr>
<td>ORBacus Trading Service</td>
<td>Java VM (Windows NT)</td>
<td>10036</td>
<td>2256</td>
<td>7780</td>
</tr>
<tr>
<td>xmove broker</td>
<td>Java VM (Windows NT)</td>
<td>11540</td>
<td>1356</td>
<td>10184</td>
</tr>
<tr>
<td>JavaCard Roaming Framework &amp; xmove browser</td>
<td>Java VM (Windows NT)</td>
<td>20740</td>
<td>2452</td>
<td>18288</td>
</tr>
</tbody>
</table>
domain onto the concrete UNIX/X11 domain. The xmove broker makes itself available to the xmmove proxy servers using a CORBA Trading Service.

Roaming users utilize the RoamX infrastructure by means of a JavaCard, which acts as multi-application access controller. The underlying JavaCard Roaming Framework allows multiple roaming-aware applications to be employed in parallel, using for example JavaCards and/or Palm Pilots as access medium. In the case of RoamX, the user interacts with the infrastructure through a Java-based GUI, called the xmmove browser. Using drag-and-drop operations, application windows can be migrated between different desktops on a selective basis.

The development of the RoamX infrastructure revealed strengths and weaknesses of different distributed object technologies, in particular when applied jointly. One task was to enhance the legacy xmmove proxy server, written in C, with CORBA interfaces. Two diverse strategies were possible: writing a separate wrapper application and integrating CORBA with the legacy code. While in the first case, the original, single-threaded design is retained, the second one introduces multi-threading into a single, CORBA-enhanced server application. We chose the second alternative, introducing the problem of proper synchronization, which is highly application specific. However, due to the simple internal structure of the original xmmove proxy server, only minor modifications of the legacy code were necessary.

The integration of Jini into a C/C++ based application is another aspect investigated in the development of the RoamX infrastructure. We showed that the Jini framework is highly flexible in its design, allowing the delegation of Jini-specific functionality into the Java-based xmmove broker, while the service’s interfaces itself may be exported directly by the delegating non-Java application.

However, development as well as deployment of distributed object applications still lacks considerably. Simple tasks like the integration of the CORBA IDL-compiler into the make-support is only one example. We used the SNiFF+ IDE [18] for the development of the C/C++ based xmmove proxy server. The provided makefile templates support the Orbix IDL-compiler and its file naming conventions only. Adapting the development environment and its makefile templates to employ MICO and ORBacus was a significant effort.

On the Java side, we used JBuilder Enterprise, which includes support for the VisiBroker CORBA implementation while offering Orbix as another alternative. However, different IDL compilers cannot be integrated into the development environment as smoothly as VisiBroker and Orbix, resulting in the loss of automatic dependency checks.

The increasing acceptance of CORBA as a middleware for distributed object applications is documented for instance by its inclusion into the Java 2 platform. However, the capabilities of this implementation are still severely limited. For example, there’s no full support for CORBA 2.3, e.g. no Portable Object Adapter is available. Those and other restrictions quickly lead to the decision to use a different, CORBA 2.3 compliant ORB. One result of this decision is the need for the ORB libraries to be made available to the application. Although automatically downloading code is an integral part of the Java runtime environment, besides security aspects, the size of the ORB’s jar file (approximately 2.7 MB in the case of ORBacus) nearly prohibits this approach.

The joint appliance of CORBA, Jini and JavaCards in the realization of RoamX has shown to be beneficial. Each technology provides suitable abstractions, streamlining the overall implementation, and their combination proved to be supplementary. However, the administrative effort to setup a proper infrastructure for a complete RoamX installation is, to say the least, complicated. Different services and tools have to be installed, setup properly and started in the right order. Policy files and security settings have to be established and administered. Furthermore, the resource consumption of the whole infrastructure has been shown to be huge.

The obstacles described so far makes the joint appliance of different technologies as has been done in RoamX acceptable in the context of research projects and for the implementation of prototypes only. Increasing pervasiveness probably will contribute in simplifying things considerably, fostering the applicability in industrial-strength applications.

6. References


