Enabling Open Window for Use with Enterprise Firewalls

ANDERS HEINTZ

MASTER OF SCIENCE PROGRAMME
Department of Computer Science and Electrical Engineering
Division of Media Technology

2004:089 CIV • ISSN: 1402 – 1617 • ISRN: LTU – EX – 04/89 – SE
Enabling OpenWindow for use with enterprise firewalls

Anders Heintz

February 22, 2004
Abstract

This paper describes a system for enabling Remote Proximity, Inc.'s collaboration product OpenWindow for enterprise firewalls. Enabling for enterprise firewalls means that the application should use standard communication protocols which firewalls recognize and allow. To resolve this, SOAP (Simple Object Access Protocol) and CORBA (Common Object Request Broker) were introduced into the system.

Introducing SOAP and CORBA into the OpenWindow application did not only require an extension of the current design, it implied a total remake of the existing inter-object communication model (consisting of RMI and Voyager). This new model allowed all four protocols to coexist as well as enabled the product for more protocols. This model and the resulting prototype shows that there exists no ideal model where all these inter-object communication protocols coexist and the therefore is the resulting prototype a compromise between theory and pragmatism.
Preface

The work on this paper started in late February 2002 and ended in July the same year as a thesis project for the author. Parts of the prototype developed are now used in Remote Proximity, Inc.'s product OpenWindow, which is currently being marketed as a competitor to products like WebEx.

The work has been performed in the Remote Proximity, Inc. Stockholm office with daily contacts with the tutor at Remote Proximity, but also at Remote Proximity, Inc. headquarters in Boston, USA.

First, I thank my tutor at Remote Proximity, Inc.; David Bergman for all his patience with stupid questions, tuition and good friendship. I also thank my girlfriend, Sofia Nilsson, for all her patience with my long nights in front of my computer. Last, but not least, I would like to thank my employers during the time of this work, Include Datorkonsulter AB and Kentor IT AB, for letting me build the prototype and write this paper on work hours.

Thanks!
### Contents

1 Introduction .................................................. 1
   1.1 Task ................................................................ 2
   1.2 Purpose ...................................................... 2
   1.3 Scope .......................................................... 3

2 Methods ............................................................... 4
   2.1 SOAP .......................................................... 4
      2.1.1 Client .................................................... 4
      2.1.2 Server .................................................... 6
      2.1.3 Implementation selection .............................. 6
   2.2 CORBA ......................................................... 7
   2.3 Use of multiple inter-object communication protocols .......... 9
      2.3.1 Message flow ............................................ 10

3 Implementation .................................................... 11
   3.1 General ......................................................... 11
   3.2 Incorporation of SOAP and CORBA into OpenWindow .......... 11
      3.2.1 Initial design ............................................ 11
      3.2.2 Conflict with existing design ......................... 12
      3.2.3 OpenWindow Communicator ............................ 13
      3.2.4 Implementation of SOAP interfaces .................. 14
      3.2.5 Implementation of CORBA interfaces ................ 15

4 Discussion ......................................................... 18
   4.1 Standards — pros and cons .................................. 18

5 Further Work ...................................................... 20

6 References ........................................................ 21
Chapter 1

Introduction

The OpenWindow product is a shared desktop, “enabling collaboration anywhere, anytime”\(^1\) where people can collaborate asynchronously or in real-time in a shared desktop environment. OpenWindow features functions like viewing and working on peer-created documents, co-surfing, viewing slide shows, etc. Almost any document format can be viewed and shared.

The ambition with OpenWindow is not only to provide a shared document viewing tool for normal PC environments (e.g. Microsoft Windows), but to provide a platform-independent solution for virtually any user type, OS and device. The range of client devices stretches from normal PC’s running Microsoft OS’ to Palm devices and WAP telephones.

The OpenWindow application is written in pure java, the client is Java 1.1 compliant and the server Java 2. By keeping the client Java 1.1 compliant, there is no need for additional JRE (Java Runtime Environment) installation when running the client as an applet in a web browser on normal PC systems. “Normal PC systems” means Microsoft OS’s with Microsoft Internet Explorer 5.x or higher. The Microsoft Internet Explorer ships with the Microsoft Java virtual machine, which is Java 1.1 compliant and to avoid the additional JRE downloads (which can be complicated for the normal user) the client software has to be Java 1.1 compliant.

The system uses a traditional data polling technique; information is retrieved from the server by each client periodically or when the information is likely to have changed on the server. Whenever changes are made on the client, the client “pushes” the information change (the information delta) to the server.

A general solution for distributed systems, like the Java 2 class Proxy\(^2\), is unfortunately not possible to use in the OpenWindow product since the client has to be (as described above) compatible with JDK 1.1 and PersonalJava.

---

\(^1\)OpenWindow web site: http://www.remoteproximity.com/website/index.html

\(^2\)Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the super class of all dynamic proxy classes created by those methods.” ... “A dynamic proxy class is a class that implements a list of interfaces specified at runtime when the class is created...”
1.1 Task

The goal was to create an extended prototype version of the OpenWindow application with a set of new inter-object communication protocols to make the product usable behind strict firewalls.

A formal task list:

1. Implement a HTTP interface between the client and the server.
2. Implement a CORBA\textsuperscript{3} interface between the client and the server.

As Figure 1.1 shows, the OpenWindow system will rely on four different inter-object communication protocols, Voyager, RMI, HTTP (SOAP) and CORBA.

1.2 Purpose

Although the polling nature of the OpenWindow architecture bypasses most firewalls blocking only incoming TCP/IP connections, some enterprise firewalls prohibit outgoing connections that use ad-hoc port numbers and/or ad-hoc communication protocols. Since the application targets major enterprise environments, it is imperative to use standard TCP/IP ports and communication protocols, such as HTTP and CORBA.

Initially, OpenWindow supported communication through the native Java inter-object communication protocol RMI (Remote Method Invocation)\textsuperscript{4} and the communication protocol provided by the Voyager\textsuperscript{5} product.

Thus, the ultimate goal was to extend the current system to cover a wider market segment and enable this solution for enterprises with high security demands.

\textsuperscript{3}CORBA: Common Object Request Broker Architecture

\textsuperscript{4}http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html [RMI]

\textsuperscript{5}Created by ObjectSpace, Inc., recently sold to Recursion Software, Inc.
1.3 Scope

In order to fulfill the goal, a prototype was created with the features described in the task section. Limitations for the development of the prototype were decided on in cooperation with RP:

1. Only implementation and documentation, neither optimization nor implementation of more feature than current implementations

2. No analysis to determine the best SOAP and CORBA implementations.

3. Java versions available at clients are assumed to be Java 1.1.4 or higher.

The next section describes the solution to the problems described above. Chapter 3 describes implementation the results of the solution, Chapter 4 discusses the solution and results and Chapter 5 describes future work.
Chapter 2

Methods

This section describes the method and the design decisions made. It also describes standards used for implementing the prototypes.

Remote Proximity, Inc. is very prone to use open standards, even if it would have been feasible to create an own inter-object communication protocol using standard Java serialization and socket programming or XML de-/serialization, but then OpenWindow would have been tied to the Java world and that would not only limit the future of the product, it also contradicts Remote Proximity, Inc. guidelines and spirit. Therefore open standards were used. At time of writing, there exist formal plans for implementations of Flash\(^1\) and Brew\(^2\) clients in a not very distant future.

2.1 SOAP

SOAP, Simple Object Access Protocol is a W3C (WWW Consortium)\(^3\) standard. The SOAP specification states:

“SOAP is a lightweight protocol for exchange of information in decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.”

In rough words, SOAP can be described as RPC (Remote Procedure Call) over HTTP using XML.

2.1.1 Client

The client process, depicted in 2.1, is divided into three phases: i) client application method calling, ii) serialization/deserialization of arguments/return result and, iii) en-/decoding phase, SOAP-XML request is wrapped in a HTTP request.
CHAPTER 2. METHODS

Figure 2.1: SOAP client process overview

Figure 2.2: SOAP server process overview
2.1.2 Server

Figure 2.2 shows the SOAP server process. It is not completely different from the client process. As depicted, the SOAP server contains an additional servlet which processes the request non-existent in the client process.

1. Request is received from client
2. A new servlet thread is spawned and the request is forwarded to the new thread.
3. The request is processed by the servlet engine.
4. The HTTP request is decoded and the SOAP-XML request is deserialized.
5. The SOAP servlet invokes the service method.
6. The result is returned.
7. Response (result) is sent to the encoder which encodes the result as SOAP-XML and creates a HTTP response.
8. The HTTP response is sent to the SOAP servlet.
9. Response is sent to the HTTP server.
10. Response is sent to the calling SOAP client.

2.1.3 Implementation selection

There are numerous SOAP-implementations available, both commercial and open source. In our case, the decision came upon Apache SOAP. Apache SOAP is well documented, open source implementation of the SOAP submission to W3C and based on the IBM SOAP4J, which it also supersedes. Thus, the decision to use Apache SOAP was based on:

1. Apache's reputation for providing good software
2. Provides server-side infrastructure for deploying, managing and running SOAP services.
3. Research indicated that it is one of the most widely used SOAP implementation.
4. Can be used with Java 1.1.
5. Has built-in support for basic Java datatypes (apart from primitives) like Vector, Hash Table, etc.

See Section 5 for more information.

To learn more about the Apache SOAP, please visit http://xml.apache.org/soap/

¹http://www.macromedia.com/software/flash/ [Flash]
²http://www.qualcomm.com/brew/ [Brew]
³http://www.w3.org/TR/SOAP/ [SOAP]
2.2 CORBA

According to OMG (Object Management Group)\(^4\),

"CORBA is the acronym for Common Object Request Broker Architecture, OMG’s open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network."\(^4\)

CORBA uses stubs and skeletons in a similar way to RMI (to put it correctly, RMI uses stubs and skeletons similarly to CORBA). The client (the caller) uses a local representation of a remote object, the stub, which implements the same interface as the remote object. The stub is responsible for, via the ORB (Object Request Broker), calling remote methods on the server side (the receiving end). The receiver skeleton assembles (unmarshals) the message (the parameters) and calls the actual implementing object. Then the result (return value or exception) is disassembled (marshalled) and sent back to the caller.

\(^4\)http://www.omg.org [CORBA]
CHAPTER 2. METHODS

For each object type that should be used remotely, passed as an argument or a return result remotely, an interface is defined in the OMG IDL (Interface Definition Language). Any client that wants to invoke an operation on the object must use the IDL interface to specify the operation it wants to perform, and to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them. The interface definition is then used to marshal the results for their trip back, and to unmarshal them when they reach their destination.

The IDL interface definition is independent of programming languages, and there exists mappings for all popular programming languages: C, C++, Java, Smalltalk, Lisp, etc.
### 2.3 Use of multiple inter-object communication protocols

Using multiple inter-object communication protocols side by side, requires a clearly defined and unambiguous communication interface which makes underlying, protocol specific, layers transparent and therefore enable application development without concern of the inter-object communication protocol.

Figure 2.4 shows the selected structure for enabling multiple protocols in the OpenWindow application. It clearly depicts which “layers” are common for SOAP and CORBA and which are protocol dependent. The SOAPRoomCommunicator and the CORBARoomCommunicator implement the same interface, RoomCommunicator. The RoomCommunicator interface contains the necessary methods for the OpenWindow client/server communication. More information regarding this interface can be found in section 3.2.3.

The client application is initialized with an instance of a RoomCommunicator implementation. As depicted in figure 2.4, the implementation can be either one of SOAPRoomCommunicator or CORBARoomCommunicator (or one of the existing protocol communicators, RMIRoomCommunicator and VoyagerRoomCommunicator) depending on the appearance of the server URL the client is provided with. A reference to this object is kept in the client throughout its entire lifetime unless a communication failure occurs. Then, the RoomCommunicator is reinitialized using, primarily, the initial server URL. However, the client may have been provided with more than one URL and therefore another inter-object communication protocol may be selected if an error for a certain protocol persists.

<table>
<thead>
<tr>
<th>OpenWindow Client</th>
<th>SOAPRoomCommunicator</th>
<th>CORBARoomCommunicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Transport</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network (e.g. TCP/IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP Server/Servlet Engine</td>
<td></td>
<td>CORBA &quot;server&quot;</td>
</tr>
<tr>
<td>SOAP Server Servlet</td>
<td>CORBARoomCommunicator</td>
<td>CORBARoomCommunicator</td>
</tr>
<tr>
<td>SOAPRoomCommunicatorImpl</td>
<td></td>
<td>CORBARoomCommunicatorImpl</td>
</tr>
<tr>
<td>Autonomous Communicator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMI/Voyager</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenWindow Server</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4: The OpenWindow inter-object communication using SOAP and CORBA


2.3.1 Message flow

Following the reasoning in the previous section, the “message flow”, i.e. the method call flow from the OpenWindow client to the server, is shown in figure 2.5.

1. A RoomCommunicator interface method call is issued from the OpenWindow client application.

2. Depending on the initialization URI, the RoomCommunicator implementation instance makes a remote call to the corresponding inter-object communication protocol server over a network, i.e. Internet.

3. There exists a “listener” for each protocol enabled in the OpenWindow server installation which receives the remote call over the network.

4. All “listeners” has a reference to a Autonomous communicator which internally in the server environment (the server environment may be distributed as well) uses RMI for communication with the server representation of the desktop room.
Chapter 3

Implementation

This section describes how the prototype version of OpenWindow was implemented and which problems occurred during the implementation phase.

3.1 General

The final result is a prototype version of the OpenWindow application where:

i) multiple inter-object communication protocols are allowed to coexist,

ii) SOAP and,

iii) CORBA communication is enabled.

OpenWindow was initially equipped only with the RMI and the Voyager inter-object communication protocols. Extending the protocol set increased the complexity significantly and even though CORBA and RMI are comparable, there exist a number of differences that requires a well defined communication structure.

Originally, the inter-object communication was distributed in the sense that all individual object types handled their own communication and were implemented using the Java-specific binary protocols mentioned earlier.

Voyager, RMI and CORBA, being binary protocols, have somewhat similar API's, SOAP however, has not. The SOAP core specification lacks a number of features from traditional messaging systems and distributed object systems. Features missing are among other; distributed garbage collection, objects-by-reference and activation. SOAP, Simple Object Access Protocol is, as the name indicates, only a simple way of retrieving remote objects.

3.2 Incorporation of SOAP and CORBA into OpenWindow

3.2.1 Initial design

Originally, OpenWindow used only the Voyager product for inter-object communication since it was available for handheld devices and J2ME as well as normal desktop environments. However, in certain aspects it was desirable with RMI as an addition to the Voyager product, and due to great similarities, RMI was easily introduced. Figure 3.1 shows the original design
(with both Voyager and RMI). Even though this design is the most common (and perhaps “correct”) way of making a system distributed, even though it is perhaps not the best solution when a system must be equipped with multiple distribution contingencies.

### 3.2.2 Conflict with existing design

During the design and implementation of the prototype, it became obvious that the original design, where each object type was aware and responsible for its own communication, was not suitable with the incorporation of SOAP and CORBA. There are three principal reasons to this, first, SOAP largely differs from the binary RMI/Voyager/CORBA protocols, second, it is likely the inter-object communication protocol set will be further extended and make the code very “cluttered” therefore hard to maintain. Finally, since all object types are responsible for its own communication, each object type had to be aware of how the object was serialized/deserialized, thus making the OpenWindow system dependent on third party implementations.

Since OpenWindow is a very hierarchical system, the solution was to gather all object communication in the root object, the Desktop object, in a “communicator”. A communi-
Figure 3.2: Object communication flow after gathering communication in one “single location”

The base communicator of the OpenWindow system is called the RoomCommunicator. The communicator is an interface consisting of four different methods for fetching and pushing event information:

- Fetching updates.
- Sending a room event.
- Get a snapshot of the desktop plane
CHAPTER 3. IMPLEMENTATION

- Get a snapshot of the room

**Communication methods**

**Fetching updates:** Updates are fetched from the openWindow server periodically or whenever it is likely an event has occurred (probability calculated using an internal algorithm) to reduce network traffic.

**Sending room events:** Whenever a client performs a task (opens a window, closes a document, etc) an event notification is sent to the openWindow server.

**Get a snapshot of the desktop plane:** Used for obtaining a snapshot of the desktop plane. A desktop plane snapshot is a snapshot of the desktop objects in a desktop plane.

**Get a snapshot of the room:** Used for obtaining a snapshot of the room. A room snapshot is a snapshot state of a complete room, including a list of users, forum and the desktop.

**Communication events**

There are a number of various events that are triggered in the system:

- **Avatar event:** changes in the state of a user (avatar = “an embodiment”, in this context a desktop user). Examples are when users logs in and out, changes online state, etc.

- **Forum event** changes in the forum associated with the desktop forum object.

- **Desktop event:** components are created, resized and destroyed.

- **Document event:** documents stored in the desktop changes.

- **Drawing event:** associated with drawing layers, either emitted as a bi-effect of certain actions or embodiments of such actions.

**3.2.4 Implementation of SOAP interfaces**

As described earlier, the major reason for SOAP inter-object client/server interface was to enable clients hidden behind “paranoid” firewalls. By using SOAP and a well configurable HTTP server, all inter-object traffic can take place using the HTTP protocol on the HTTP port (80) using the object to XML mapping of SOAP.

**Implementing SOAP communication**

In SOAP, all datatypes that are meant to be transferred as a method parameter or as return result have to be defined by a deployment descriptor. Basically the deployment descriptor defines, for client and server, remotely accessible methods, the arguments and result of these methods. The definition of communicable datatypes includes the name of a (de)serializer for the datatype involved. One common serializer/deserializer included in the Apache SOAP API is the BeanSerializer. The BeanSerializer is capable of serializing and deserializing a JavaBean. In this case, most of the datatypes that has to be transferred are very similar to a JavaBean and were therefore rewritten to “qualify” as a JavaBean. However, this left us with a small number of datatypes that were not serializable by the BeanSerializer. Consequently, serializers had to be specifically written for them. Eventually, this was limited to only one extra de-/serializer.
3.2.5 Implementation of CORBA interfaces

Initially the idea was to use CORBA as a data carrier, carrying serialized Java objects. The sole reason of using CORBA was to enable firewall bypassing. However, mentioning CORBA only in the sense of bypassing firewalls, the essence of the CORBA idea is neglected and of course the idea rises:

"Why is not CORBA used how it was intended to be used?"

If the goal was only to use CORBA to bypass firewalls, java object could be streamed over the CORBA "channel" by using the standard Java object serialization, but using the standard as intended, an even better advantage is created: enable the OpenWindow system for object communication with non-java clients.

By using CORBA "the way it is intended", not only may the client be implemented in another programming language like ActiveX or Brew, server objects may also be implemented as native CORBA objects (for computing intense operations like server rendering for example).

"Java for OS independent software, CORBA for programming language independence."

Implementing CORBA communication

Using CORBA the "standard" way requires that all classes used in inter-object communication implements a StreamableValue interface. This was unfortunately not desirable in the OpenWindow environment. Working like an applet, the OpenWindow client only downloads necessary classes, if something extra is required; it is downloaded on a per need basis.
By making all communicable objects implement the `StreamableValue` interface, the application is tied to a CORBA implementation even though it in some cases does not use CORBA! One of the goals with `OpenWindow` is to keep the client as thin as possible, and if all communicable objects implement the CORBA `StreamableValue` interface, a large number of CORBA-specific classes have to be downloaded. That would “thicken” the client significantly.

The solution to this problem was to build a CORBA wrapping function for each of the communicable objects. Each communicable object type has its respective wrapping class which implements the `StreamableValue` interface. These wrapping objects are merely data containers for the data in the original object.

Java objects that need to be communicable can be “decompiled” by a tool named `java2idl`\(^1\). This tool creates an IDL (Interface Definition Language) file. The IDL file is the core of every CORBA-enabled system. The `Interface Definition Language` is a programming language independent language for describing communicable objects and methods.

The IDL file can then be compiled to a “real” programming language, in this context, Java. The `idl2java` compiler creates a set of abstract classes for which implement classes are created to image a “real” object. These are the wrapper classes. The wrapper objects were then constructed with the “real” object and the wrapper object is populated with the data of the “real” object. On the receiving end, a copy of the real object can be obtained. By using these wrapper classes in the `CorbaRoomCommunicator` we can achieve “real” CORBA communication without clogging the `OpenWindow` object structure with CORBA dependencies.

The communicable `OpenWindow` object has no knowledge how to be assembled and disassembled “CORBA-style”, however, the wrapper objects do and they are populated with the data that needs to be transferred.

\(^1\)http://www.borland.com/besvisibroker/previous/index.html [Borland Visibroker 4.5 for Java]
Albeit this strategy is fully functional, it is not the perfect, desirable implementation of CORBA in any system. It is imperative the client is kept as thin as possible, and additionally it is highly desirable that the client is JDK 1.1 compliant. The strategy for inserting the CORBA implementation in the prototype is a compromise to make ends meet where CORBA is used to connect systems implemented in different programming languages yet client thinness is preserved.

However, the CORBA implementation used in the development of the prototype is not compatible with Java 1.1.4; therefore Java 2 is required on those clients CORBA needs to run on. Making the CORBA functionality in OpenWindow compatible with Java 1.1.4 is left for the future.
Chapter 4

Discussion

A major part of building the prototype was to adapt the current OpenWindow system to enable use of additional inter-object communication. The result was successful by the means that CORBA and SOAP were successfully implemented. However, the CORBA implementation is not the perfect CORBA solution due to the fact the incompatibility between the CORBA architecture and the OpenWindow architecture. But in the sense of usability, the implemented prototype is considered a success though.

The implementation of these new inter-object communication protocols has prepared the OpenWindow system for following:

- Bypassing “paranoid” firewalls
- Non-java clients
- Non-java server objects for computational intense operations
- A framework for further inter-object communication protocols.

One conclusion that can be drawn is that the java implementations of the SOAP and CORBA inter-object communication protocols do not cooperate well. This conclusion is based on the fact that when multiple protocols coexist, it is not only desirable that the business object and business logic is independent of the communication type, it is a requirement. In OpenWindow, client thinness is utterly important and class dependencies are downloaded on a need-only basis. Making the application itself dependent on protocol implementation classes not needed should be considered strictly forbidden.

The only requirement for making a class distributable using the binary Voyager and RMI, is that the class implements the Serializable interface. The Serializable interface is a part of the standard Java SDK and it therefore already exists on the client.

4.1 Standards — pros and cons

Even though Remote Proximity, Inc. advocates the use of existing standards to ensure its products is easy to maintain and may be used in a multi-OS, multi-platform environment, following a standard may not always be ideal. The SOAP case is an example of this. SOAP was complemented with an internally Remote Proximity, Inc. defined framework for XML-based “inter-object communication”, a truly simpler (and more OpenWindow specialized) object access protocol which does not suffer from the overhead of SOAP. The internal framework is
however, usable only when the client and server are provided by Remote Proximity, Inc., when clients and client applications are provided by other software vendors, they should, to ensure compatibility, use one of the standardized inter-object communication protocols.
Chapter 5

Further Work

Even if the prototype is considered a success, the work should naturally not stop, today’s IT products need constant improvement to withstand competition and the OpenWindow product is no exception from the rule.

The SOAP prototype depends on the Apache SOAP implementation. However, it is not likely that Apache’s implementation is the best choice. Even though it is robust and complies well with the SOAP 1.1 specification, it is a quite large package which does not coincide with thin clients and low bandwidths. There are lighter packages more suitable for low bandwidth/low memory devices, for example the SOAP implementation by Enhydra.org, kSOAP. kSOAP is a SOAP API suitable for the Java 2 Micro Edition (J2ME).

The same reasoning applies to the CORBA implementation as for the SOAP implementation. The Inprise (Borland) Visibroker used is probably not the perfect solution for OpenWindow. Only a very small subset of the CORBA standard is used by the prototype and there are numerous CORBA vendors with different target audiences where a more suitable implementation might be found. Additionally, the OpenWindow CORBA functionality has to be adapted to Java 1.1.4 for compatibility with older browsers and browsers with low Java functionality like Microsoft Internet Explorer.
Chapter 6

References

OpenWindow http://www remoteproximity com/website/index html
RMI http://java sun com/j2se/1.3/docs/guide/rmi-iiop/index html
Flash http://www macromedia com/software/flash
Brew http://www qualcomm com/brew
SOAP http://www w3 org/TR/SOAP
Apache SOAP http://xml apache org/soap
CORBA http://www omg org
Borland VisiBroker http://www borland com/besvisibroker/previous/index html