Bridging Bonobo Components

Abdelmalek Squalli Houssaini

A dissertation submitted to the University of Dublin in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

September 16, 2001

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Date: September 16, 2001

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Date: September 16, 2001

Abstract

To manage the increasing complexity of software, many modern and large applications make use of components. GNOME, the free desktop environment for UNIX like systems, has created its own component model, namely Bonobo.

The Bonobo component model is based on CORBA technology. Relying on CORBA enables developers to access Bonobo components from different programming languages and platforms. It also means that IIOP, the protocol that guarantees interoperability between CORBA products, is used to communicate with Bonobo components. However, most firewalls prevent IIOP communication. This dissertation deals with bridging Bonobo components. In other words, it is about accessing Bonobo components that are behind a firewall.

SOAP is an XML based protocol that uses HTTP for transporting its messages. In order to bridge Bonobo components, we will build an IIOP to SOAP bridge and vice versa. The system will redirect all the messages destined to the Bonobo components into the bridge. As the bridge uses the HTTP protocol, all the messages will get through the firewall.

Acknowledgements

I would like to thank my supervisor Dr. Simon Dobson for his help and advice throughout the year. I would also like to thank my family for their support and encouragement. Finally, many thanks to all the classmates for their friendliness and sense of humour.

Contents

List of Figures

Chapter 1

Introduction

Over the last decades, computer hardware capacity has grown rapidly. As a result, computer software followed the move and has become more complex. It was necessary to manage this complexity. At the same time, the trend was to move from large mainframes to distributed systems. Remote Procedure Call (RPC) came then in order to make remote function calls transparent to the programmer. During the last decade, object oriented and client-server paradigms have become widespread, allowing programmers to reduce software complexity. The next step forward was towards distributed objects and software components.

Distributed object technologies came as a result of the object oriented and client server paradigms. As customers no more accepted proprietary systems because of their limitations, a consortium was set up in order to provide a common architectural framework to vendors and to promote object oriented software development. Consequently, the Object Management Group (OMG) consortium has provided the most important distributed object technology, namely the Common Object Request Broker Architecture (CORBA). OMG makes the specifications and vendors implement them. In order to provide interoperability between CORBA products, OMG has also defined a protocol called Internet Inter-

ORB Protocol (IIOP) and a neutral language called Interface Definition Language (IDL).

The adoption of component software technologies was motivated by reducing the complexity of applications. This is achieved by reducing the amount of information a programmer needs to know about the system. Component software also enables programmers to build larger, more complex applications by glueing different components together into bigger applications [dI99]. In brief, components enable to reuse code and to reduce software complexity.

The GNU Object Model Environment[1] (GNOME) is a free desktop environment for UNIX like systems. GNOME developers needed a component architecture for building large and powerful applications. This need resulted in the foundation of Bonobo[2].

Web services are a new distributed computing model. They are modular applications that can be located and invoked across the Web. Web services are based on the Simple Object Access Protocol (SOAP). SOAP is a protocol that allows invoking remote methods over the HyperText Transfer Protocol (HTTP) using messages encoded in the Extensible Markup Language (XML). Talking about SOAP, the CEO of Digital Creations said:

> At long last, the Web has a complete platform strategy. Operating systems, programming languages, database technologies, and even component models all disappear behind a straightforward Web API.

Furthermore, SOAP Security Extensions like Digital Signature are being developed so that they can be integrated with the most popular web security technology, namely Secure Sockets Layer or SSL. By doing so, important security features such as non-repudiation can be achieved.

[1] GNOME can be considered as a competitor of the K Desktop Environment (KDE).

[2] Bonobo does not stand for anything. It is an endangered ape species.

1.1 Project motivations

Since components are intended to cooperate with each other, they usually rely on a communication layer. GNOME developers decided that Bonobo should rely on CORBA[3]. As a result, Bonobo components will communicate using IIOP and will define their interfaces using IDL.

Most companies filter their network traffic by setting a firewall. Generally, such a system prevents any IIOP communication. Therefore, it is impossible to communicate with a Bonobo component that is behind a firewall.

Even though we assume that the firewall does not prevent IIOP communication, security issues would be of great concern. It would mean that further development involving the use of the CORBA security service is necessary. Obviously, this is only possible when the CORBA implementation supports the security module.

Solving the firewall and security issues mentioned in the previous two paragraphs will enable us to:

- Communicate with Bonobo components even if they are behind a firewall.
- Ensure that the communication is secure without much effort.

1.2 Project goal

Using SOAP to communicate with Bonobo components will enable us to solve both the firewall and security issues. Indeed, it solves the first problem since SOAP is based on HTTP which is a firewall friendly protocol. It also solves the second problem since a security layer can be easily built on top of HTTP. Another reason for using SOAP is that:

[3]For comparison, Microsoft's OLE relies on COM/DCOM.

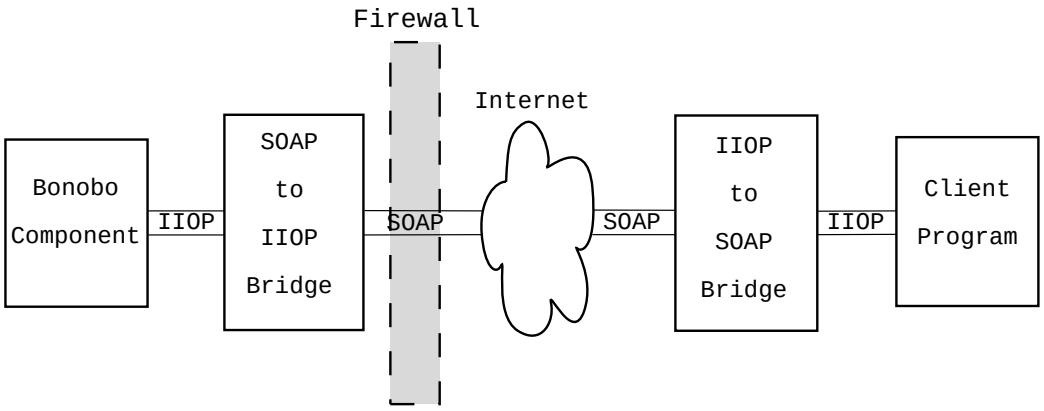

Figure 1.1: Basic principle of the bridge

> The existing Internet security infrastructure has embraced HTTP to the point that trying to communicate across organizations using *anything other than HTTP* requires an excessive amount of organizational and engineering resources. [3]

This thesis is about addressing the issues mentioned above by building an IIOP to SOAP bridge. Since it is possible to communicate with Bonobo components using IIOP, building an IIOP to SOAP bridge will make it possible to communicate with them using SOAP. Figure 1.1 shows how to bridge Bonobo components.

1.3 Roadmap

Here is the outline of the thesis:

- The **state of the art** chapter starts with a general introduction to CORBA, GNOME, Bonobo, SOAP and ORBit. Then it covers some CORBA details related to the project, before giving an idea about the bridge.
- The **Bonobo** chapter presents some details about the Bonobo technology. Due to the complete lack of Bonobo documentation, some sections in this

chapter aim to provide a good basis for understanding Bonobo. The Bonobo chapter is used in the implementation chapter but not in the design one.

- In the **design** chapter, we examine the different issues relating to building an IIOP to SOAP bridge and justify the decisions that have been taken.
- The **implementation** chapter is about implementing and testing the bridge. A Bonobo application and three CORBA clients will be used for the test exactly as shown in figure 1.1.
- In the **conclusion** chapter, we will discuss the work achieved, the difficulties encountered as well as the future work that may be done.

Chapter 2

State of the Art

2.1 CORBA

2.1.1 Distributed objects

Distributed object technologies allow objects to interact on different platforms across the network. The OMG has played a major role into building the most important distributed objects infrastructure namely CORBA. Other important infrastructures for developing distributed objects include:

- Distributed Component Object Model (DCOM).
- Enterprise Java Beans (EJBs).
- Remote Method Invocation (RMI).

The OMG states that it is:

> An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications. [Obj]

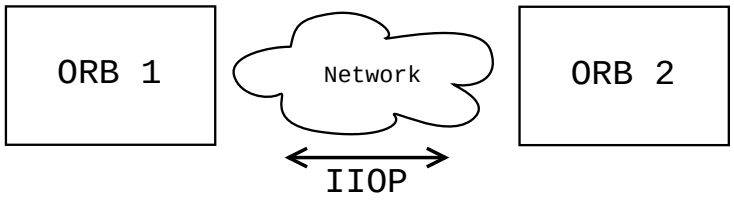

Figure 2.1: ORB interoperability

The OMG has about 800 memberships. They include virtually every large company in the computer industry, and hundreds of smaller ones [Obj]. The OMG debates and publishes CORBA specifications, it does not implement them.

CORBA allows programs to communicate regardless of their programming language, the platforms and machines they are running on. The communication between heterogeneous programs is guaranteed due to the IIOP protocol. IIOP is a TCP/IP[1] based protocol specified by the OMG consortium.

In CORBA, an object is an instance of a class, which is defined by a set of operations, attributes and exceptions. Every object provides well defined services through its interface, and every object has a reference by which it is identified in a distributed environment.

2.1.2 Object Request Brokers

An Object Request Broker (ORB) is an implementation of the CORBA specifications. Many ORBs are available. They differ in many ways. Some of them are free, others are not. Figure 2.1 shows that different ORBs spread over the network are able to interoperate via IIOP.

An ORB encapsulates the inherently heterogeneous aspects of the networks and operating systems. Thus, an ORB is a middleware because:

[1]TCP/IP stands for Transport Control Protocol/Internet Protocol

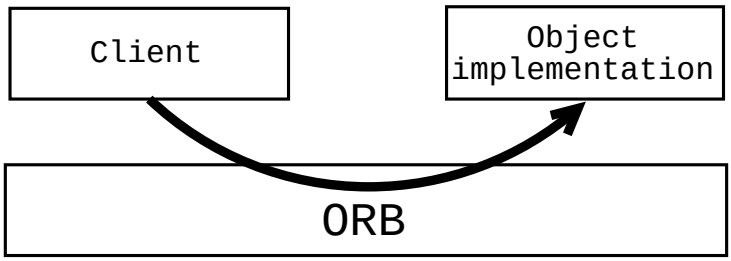

Figure 2.2: An ORB communicating a request to a target object implementation

> Middleware hides the underlying details from the application. It does this by providing a common set of services across all the platforms on which it lives. [Pop98]

An ORB can also be considered as a layer above the network and the operating system.

Technically speaking, an ORB allows a client to transparently invoke an operation on a server object (Figure 2.2). The object can be either on the same machine or across the network. Furthermore, the client need not to know the details of the object, namely its programming language, its operating system, or any other aspect that is not reflected in its interface. Briefly, an ORB makes the communication between a client and a server easier.

An ORB should accept requests, transport them to the appropriate object and possibly return results. In others words, it is responsible for the transport, marshaling and unmarshaling of messages. Marshaling consists of encoding messages and putting them on the wire, while unmarshaling consists of decoding them from the wire and placing them in an address space.

As was mentioned earlier, there are many differences between ORBs. Among these differences are the performance in terms of speed and memory usage, the language bindings provided and the CORBA services available.

Here is a list of some known ORBs:

- Free ORBs: Red Hat's ORBit, Sun's Java IDL, Xerox's ILU and MICO.
- Vendor ORBs: IONA's Orbix, IONA's OrbixWeb[2], OOC's ORBacus[3], and Inprise's VisiBroker[4].

2.2 GNOME

Bonobo is the component model that derived from the GNOME project. GNOME is a free desktop environment as well as a suite of powerful applications like the spreadsheet Gnumeric, the word processor AbiWord or the file manager Nautilus.

GNOME is part of the GNU project which aims to provide free software for Unix-like operating systems. As mentioned in the GNU website, free software means that the user is free to run, copy, distribute, study, change and improve the software[5]. The most important motivation for the GNOME project is freedom of software:

> GNOME arose from the need for a 100 percent free, open-source desktop and application programming environment. [She00]

Not surprisingly, GNOME licensing terms are of utmost importance:

> The keystone in GNOME's declaration of freedom is the GNU Public License, a clever software license from the Free Software Foundation that mandates that the source code for all GPL[6] software be freely available. [She00]

[2]Free for Universities.

[3]Free for non-commercial use.

[4]Inprise was formerly known as Borland.

[5]See `http://www.gnu.org/philosophy/free-sw.html` for more information on the meaning of free software.

[6]GPL stands for General Public License

KDE is also a desktop environment for Unix workstations. However, KDE is based on a set of libraries called Qt which had a non-free license. Now, Qt's license is also GPL for the Linux and Unix platforms [Gwy00]. KDE can be considered as GNOME's competitor. Nevertheless, it is possible to run some KDE programs from within GNOME and vice versa.

GNOME is distributed by several companies like Ximian, Red Hat, SuSe, MandrakeSoft and Sun Microsystems. Theoretically, GNOME can run on any UNIX-like platform supporting the X Windows System. Here are some of the platforms that were successfully tested: Linux (any recent distribution), Solaris, FreeBSD and NetBSD. Even though GNOME is based on the X Windows System, it is not tied to a specific window manager.

From the developer's point of view, GNOME presents a powerful framework for software development. Indeed, GNOME makes use of cutting edge technologies. Among the modules that make up the GNOME architecture are:

- GTK+[7]: The toolkit for creating user interfaces.
- ORBit: A thin and fast implementation of CORBA.
- Bonobo: The architecture for creating reusable software components and compound documents.
- OAF[8]: A framework for activating CORBA servers.

[7]GTK+ stands for the GIMP toolkit, and GIMP stands for the GNU Image Manipulation Program

[8]OAF stands for the Object Activation Framework

2.3 Bonobo

Bonobo is the component model used in the GNOME project. It is largely inspired from Microsoft's Object Linking and Embedding 2 (OLE2). However, Bonobo relies on CORBA for its communication layer whereas OLE2 relies on COM/DCOM.

Large GNOME applications like the spreadsheet Gnumeric, the file manager Nautilus or the mail client and calendar Evolution make heavy use of Bonobo. Other GNOME applications like the postscript viewer, the PDF viewer or the SVG viewer also use Bonobo but in a smaller degree. Obviously, components built for these applications may well be used in future applications.

Let's suppose that a developer wants to write a mail client program that is capable of dealing with attached files. The program should then be able to view images, run videos, render HTML documents, etc. In this case, the use of components should be considered. For example, there is no need for the program to know how to plot images. This functionality should be provided by a component like the image viewer Eye Of GNOME (EOG).

Bonobo is the framework that allows GNOME developers to create:

- **Components** which are reusable pieces of software. Components help reducing the complexity of applications by reducing the amount of information a programmer needs to know about the system [dI99]. They also allow the programmer to tackle a single problem in order to provide a specific functionality. More importantly, they are easy to reuse and debug.

  Furthermore, components present well defined interfaces and interactions:

  > The magic behind component programming is in the ”interfaces” that a component export to the world. Each one of these interfaces is a ”socket” into which other components and applications

can plug into. [dI99]

In Bonobo, interfaces are described in IDL and interactions rely on CORBA. CORBA can be considered as the glue between the cooperating components.

- **Embeddable documents** which are documents that can be inserted or embedded into other documents.
- **Controls** which are graphical components. They are embedded in applications called containers. Controls are studied in section 3.3.1.
- **Scripts** which allow to manipulate an application through its Bonobo components. This is possible since components export their CORBA interfaces. An example of scripting would be to manipulate a Gnumeric spreadsheet.

Technically speaking, Bonobo is a set of CORBA interfaces and their default implementation in C. Bonobo uses the ORBit implementation of CORBA discussed in section 2.5. It also uses the GTK+ object and type systems examined in section 3.1.

2.4 SOAP

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics. [BEK+00]

Even though the SOAP specification is not tied to a specific transport protocol, most implementations are based on HTTP. SOAP provides a mechanism for

making an RPC over HTTP. The method name and the types and values of the parameters are described in XML, and the message is transported by HTTP. The use of XML encoding allow SOAP messages to be decoded easily on any platform.

Some experts believe that SOAP has a promising future because it allows to build distributed applications based on open Internet standards. They also believe that SOAP will succeed where other distributed technologies like CORBA and DCOM have failed. The problem with CORBA is that even though ORBs from different vendors do interoperate due to the IIOP protocol, this interoperability does not apply for higher level services such as security and transaction management. Moreover, CORBA uses the Common Data Representation (CDR) for encoding its messages which is not as widely used as XML. Finally, major computer companies are backing SOAP and are trying to ensure the interoperability between their implementations.[9]

SOAP is intended to solve the following problem:

> Any time you read about a company porting an application to a Web-based interface, you can conclude that they will be throwing millions of dollars out the window in lost employee productivity.
> *Jakob Nielsen, Principal, Nielsen Norman Group*

Actually, the major feature of SOAP is that:

> It's simpler and easier to implement than any existing alternative, and makes better use of the pervasive Web infrastructure. The effect is that you can pull a system together using SOAP in weeks not quarters.
> *Tim Bray, CEO, Antarcti.Ca*

[9]For more information, see the article "Web services architect, Part 3: Is Web services the reincarnation of CORBA?" on `http://www-106.ibm.com/developerworks/webservices/library/ws-arc3/`

2.5 ORBit

ORBit is the CORBA implementation that was written for the GNOME project. All the Bonobo IDL interfaces are implemented using the ORBit ORB. GNOME developers had considered the use of two ORBs before Elliot Lee and Dick Porter decided to write a new one. They considered Xerox's ILU but they couldn't get Xerox to modify its licensing terms. Then, they adopted MICO for a while but its performance was quite poor. [LZM]

ORBit is a CORBA 2.2 compliant ORB. It was designed to have a high performance, which means low memory usage and high speed. ORBit is written in C and runs over multiple platforms. It has many language bindings including C, Perl and C++. [1]

Most of the time Bonobo components will be running on the same process. However they will be using ORBit for communicating. This seems a little bit awkward since CORBA is primarily used for distributed objects in a networked environment. Sending IIOP messages from the stub to the skeleton in order to access a local object does not help performance. Actually, ORBit treats the case of a local object separately. The shortcut is indeed included in the stub of every object. Here is the shortcut used when calling the *just_hello* method on the *HelloWorld* object.

```
if (_obj->servant && _obj->vepv && HelloWorld__classid) {
   ((POA_HelloWorld__epv *) _obj->
        vepv[HelloWorld__classid])->
             just_hello(_obj->servant, ev);
   return;
}
```

This will test if the object is in the same memory space as the client. If it is the

case, the client calls directly the method on the object without any marshaling or unmarshaling.

For security reasons, the communication over the network is disabled by default. It is possible to enable this option by creating a *.orbitrc* file in the user's home directory and having "ORBIIOPIPv4=1" in it. When this option is disabled, ORBit uses UNIX domain sockets or shared memory for interprocess communication instead of TCP ports. In this case, the object references used are not interoperable since TCP ports are not used. Therefore, these object references can not be used from another machine to access the corresponding CORBA objects. Such object references can not be interpreted by IOR parsers[10].

2.6 Invoking an operation on a CORBA object

A CORBA client is intended to access a CORBA object by invoking operations on it. The client needs to know exactly two elements:

- The object reference[11] that contains information of how to connect to the object.
- The object's interface, which allows the client to know about the operations it can invoke on the object.

2.6.1 Interface Definition Language

The Interface Definition Language (IDL) is also a specification of the OMG consortium. It is a neutral language that allows to define interfaces for CORBA objects. An interface describes the operations that a CORBA client may invoke.

[10] IOR parsers will be discussed in section 2.6.2.

[11] Here, we mean the language-dependent object reference which is ORB dependent. The ORB object reference is opaque to the client and the server, its representation is ORB specific.

Every operation should have a name and a set of parameters. IDL also allows type definition, which means that it is possible to define new types based on known ones.

An IDL interface represents the contract between a client and a server. The server should implement this interface[12], and the client may invoke any operation defined in it. The client need not to know any further information about the object server, apart from its reference. The behaviour of the operations depends on the object's implementation.

An interface example

Here is a simple example of an IDL file:

```
interface BankAccount {
   attribute long balance;
   void deposit (in unsigned long amount);
   void withdraw (in unsigned long amount);
};
```

The keyword *attribute* is just a short way for declaring get and set operations. The *deposit* and *withdraw* operations look a lot like Java methods. The keyword *in* is a parameter passing mode. It shows the direction of information. There are two other parameter passing modes: *out* and *inout*. If we are interested in knowing the balance at withdraw or deposit time, we should have this interface instead:

```
interface BankAccount {
   attribute long balance;
   void deposit (in unsigned long amount, out long balance);
   void withdraw (in unsigned long amount,
```

[12]This means that the server should implement all the operations defined in that interface.

```
                    out long balance);
};
```

IDL mappings

Even though IDL is language neutral, it maps to many programming languages such as Java, C, C++, Python and Smalltalk. This mapping defines a set of rules for translating IDL elements to language specific data types. Here is a table that shows the C mapping of some IDL types:

IDL Type	C mapping
char	signed char
octet	unsigned char
boolean	unsigned char
enum	enum
any	typedef struct any { TypeCode _type; void *_value; } any;

An IDL *any* is a data type that can represent any IDL type. Its C mapping is the *any* structure which contains two elements: a data value, and a type code describing the type of the data value.

IDL compilers

An IDL compiler translates or maps from IDL to a particular language. For example, the IDL compiler can take an IDL file as an input and produce some source code in a specific language like Java, C or C++.

An IDL compiler generates many useful source code files. Two of which are important: the *stub* and the *skeleton*. The stub is used on the client side and the

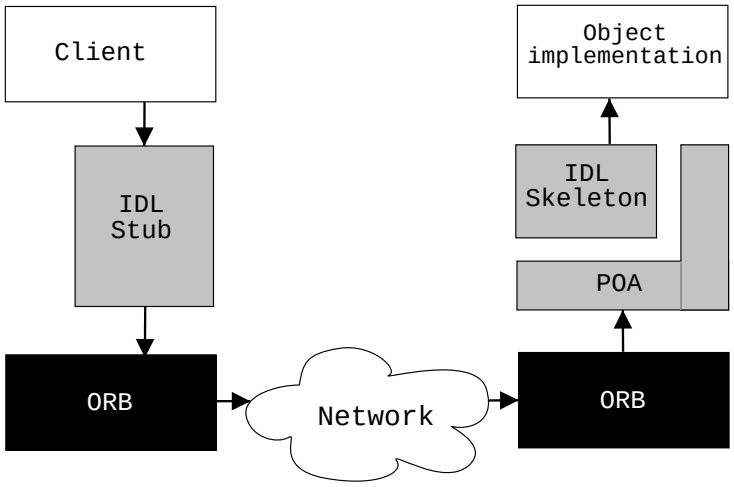

Figure 2.3: The stub and skeleton interactions

skeleton on the server side. The stub and the skeleton insulate the client and the server from networking details. The stub is a local object through which all requests to a remote object are made. It knows how to locate the skeleton and marshal messages. The skeleton takes care of all networking details and knows how to forward requests to the real object (Figure 2.3).

Different ORBs come with different IDL compilers. As an example, ORBit comes with *orbit-idl* which maps into C and Java IDL comes with *idlj* which maps into Java.

2.6.2 Interoperable Object References

An object reference allows an ORB to identify a particular object. According to the OMG specifications [Obj], it should identify the same object each time the reference is used in a request. However, there may be many distinct object references for the same object.

An object reference can be turned into a string by invoking the object_to_string

operation on the ORB object. The format of this string is predefined by IIOP. Therefore, when an ORB generates such a string, another ORB can convert it back into an object reference using the string_to_object operation[13].

Here is the code used in Java IDL:

```
String sor = orb.object_to_string(objectReference1);
objectReference2 = orb.string_to_object(sor);
```

The Interoperable Object Reference (IOR) is an object reference that is transparent to different ORBs. Clients using IORs do not need to distinguish between object references produced by a local ORB and those produced by a remote one.

Here is what a string representation of an IOR looks like:

```
IOR:010000001700000049444c3a426f6e6f626f2f436f6e74726f6c3a312e3
0000002000000caaedfba54000000010100002c0000002f746d702f6f726269
742d737175616c6c69612f6f72622d323134313832323438313535383733383
639390000000000180000000000000001240786d1df885b805000000aebff122
7e7b6388000000003c0000000101000160000006e6f777368616b2e6473672
e63732e7463642e6965001682180000000000000001240786d1df885b8050000
00aebff1227e7b6388
```

An IOR contains at least three elements:

- The version number of the transport protocol (IIOP) the server supports.
- The endpoint address, namely the IP address and the TCP port number of the server.
- An opaque object called an *object key* used by the server to identify a particular object.

[13]This works only if the string representation of the object reference has been produced by the same ORB using the object_to_string operation, or if the two ORBs used support IIOP.

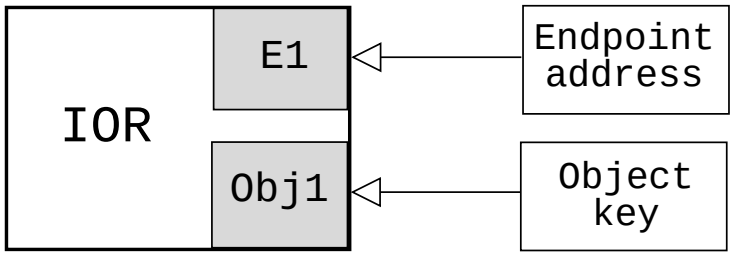

Figure 2.4: Essential elements of an IOR

The IOR in figure 2.4 points to object *Obj1* in a server whose endpoint address is *E1.*

An IOR parser allows us to extract useful information from the string representation of an IOR, including the three elements mentioned above. Many ORB vendors provide an IOR parser. For example, Orbacus ORB comes with the *iordump.exe* parser. There are also web based parsers like the ILU parser [14].

Using the Orbacus parser, here is the information extracted from the previous string representation of the IOR:

```
IOR #1:
byteorder: little endian
type_id: IDL:Bonobo/Control:1.0
Unknown profile tag: 3135221450
IIOP profile #1:
iiop_version: 1.0
host: nowshak.dsg.cs.tcd.ie
port: 33302
object_key: (24)
  0   0   0   0  18  64 120 109 ".....@xm"
 29 248 133 184   5   0   0   0 "......."
```

[14] http://www.parc.xerox.com/istl/projects/ILU/parseIOR/.

```
174 191 241  34 126 123  99 136 "+."~{c"
```

The three IOR elements mentioned above are shown in the output results. The *type_id* and the *byte order* are also included in the string representation of an IOR. The type_id represents the interface type or the repository ID, which is a unique identifier for an interface.

The byte order is very important as it will allow parsers to know exactly what the bytes represent. For example, four bytes representing an integer will be read differently depending on the byte order. The byte order information is placed in the beginning of the string representation of an IOR. In “IOR: 01000000 17000000”, the first eight digits show that the byte order is little endian. It is then possible to understand the meaning of the second eight digits; they represent an integer whose value is 23[15]. When using a big endian byte order, the same information would be represented as follows: “IOR: 00000000 00000017”.

2.7 IIOP to SOAP Bridge

OMG defines the IIOP protocol in order to guarantee the interoperability between different ORBs. Therefore, a CORBA client and a CORBA server use IIOP for communicating. More precisely, IIOP is used between the stub and the skeleton. The stub and the skeleton may be generated by different ORBs, and they will still be able to communicate (see figure 2.3).

Generally, corporate firewalls do not allow IIOP messages in and out of their network. Thus, the communication between a CORBA server and a CORBA client is impossible. However, all the corporate firewalls allow HTTP communication for Internet access.

SOAP is a protocol that provides a mechanism for exchanging structured and

[15] 17 is the hexadecimal representation of 23.

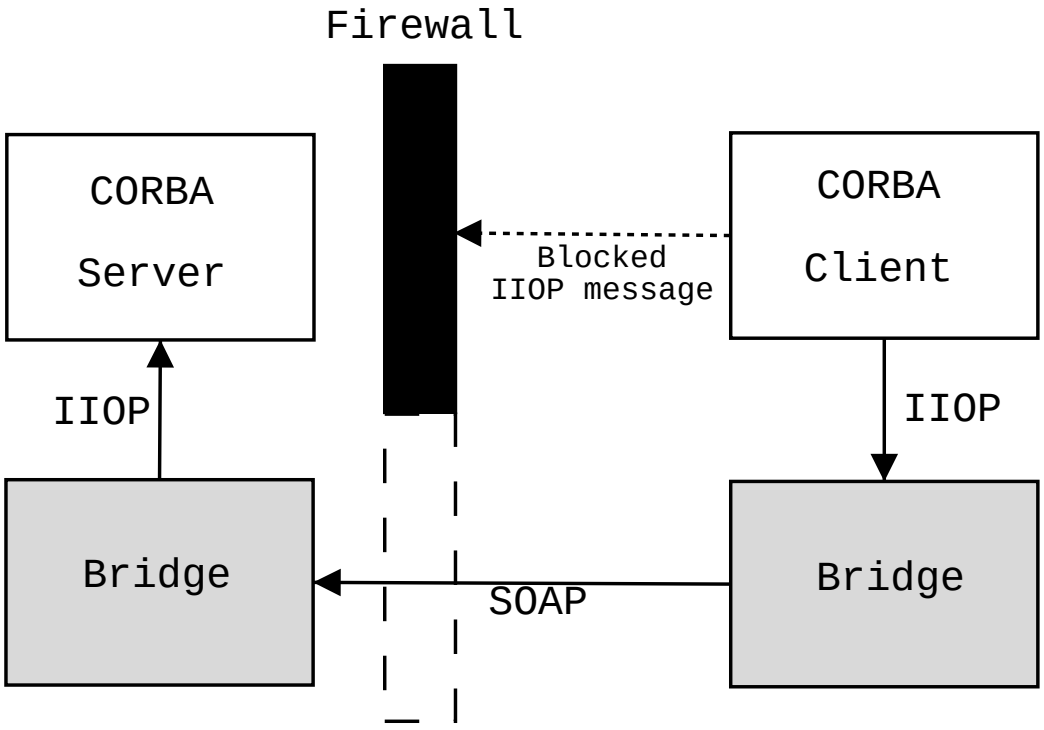

Figure 2.5: Getting through the firewall via the bridge

typed information in a distributed environment using XML. What it is interesting about SOAP is that it uses HTTP as its transport protocol. As a consequence, the firewall does not prevent a SOAP client and SOAP server from communicating.

Let's suppose that a CORBA client wants to communicate with a CORBA server that is behind a firewall (Figure 2.5). The CORBA client, and more specifically the stub, will make an IIOP message and try to send it to the CORBA server using a TCP/IP connection. As mentioned earlier, this IIOP message won't get through the firewall.

The role of an IIOP to SOAP bridge is to intercept IIOP messages, to convert them to SOAP messages and to send them over the network to a SOAP server. On the server side, the bridge must convert each received SOAP message to an IIOP message, and send it to the CORBA server to which the IIOP message is destined.

This is the idea behind the bridge.

Chapter 3

Bonobo

3.1 GTK+

The Bonobo distribution consists of a collection of CORBA IDL interfaces and their default implementation. The implementation is in C and is based on the GTK+ library. More specifically, it uses the GTK+ object system which allows writing C programs with an object oriented programming approach.

GTK+ stands for the GIMP[1] Toolkit. Originally, it was developed for the GIMP, but now it is being used in many applications and constitutes the basis for GNOME's user interface [2].

GTK+ is essentially a library for creating graphical user interfaces. It is built on top of the GLib[2] library which enhances programs portability. GTK+ is completely written in C, but many language bindings are available including C++, Perl and Python. GTK+ was written with an object oriented approach based on the GTK+ object system.

GTK+ was adopted by GNOME mainly because of its flexible and convenient

[1] GIMP stands for GNU Image Manipulation Program

[2] See section 3.1.1 for details on the GLib library.

programming interface, but also because of its licensing terms. Besides, programmers who are familiar with GTK+ will find it easy to learn Bonobo.

3.1.1 The GLib library

The GLib[3] library is written in C and contains some convenient functions and definitions. For instance, the GLib library defines basic types, memory allocation routines and useful macros like those dealing with type conversion and byte order. It also provides useful data structures like arrays, hash tables and linked lists, and includes utility functions like those related to strings.

GLib replaces many standard and common C language constructs. These replacement functions and definitions make applications easier to write and insulate them from the details of different operating systems. In other words, the advantages of using the GLib library are to increase the portability of applications and libraries, and to provide some utility functions.

Here are examples of data types defined by GLib[4]:

- gint8, gint16 provide respectively 8-bit and 16-bit integers independently of the platform. guint8 and guint16 provide unsigned integers with guaranteed size. These new definitions allow to enhance the programs portability.
- gboolean is defined for making code more readable.
- gpointer is simply a pointer to void (void *).

And here are examples of some GLib functions:

- gpointer g_ malloc(gulong size);

[3]See `http://www.gtk.org/tutorial/ch-glib.html` and `http://developer.gnome.org/doc/GGAD/cha-glib.html` For more information about the GLib library.

[4]For the GLib API see `http://developer.gnome.org/doc/API/glib/index.html`.

- void g_free(gpointer mem);
- g_new0(type, count);
- gint g_snprintf(gchar* buf, gulong n, const gchar* format, ...);

3.1.2 The GTK+ object system

GtkObject forms the basis of the GTK+ object system. It plays a similar role as the *Object* class in Java. Every GtkObject is made up of two structures, one structure that represents an object, and another that represents a class.

A **Gtk object** is an object that inherits from GtkObject. Sometimes, when there is no confusion, a Gtk object will simply be called an **object**. We will explain the inheritance mechanism further in this section. BonoboObject, GtkWidget and GtkButton are all examples of Gtk objects. Figure 3.1 shows a hierarchical tree of some Gtk objects, starting from the first object which is GtkObject. The white rectangles represent some Gtk objects used in the Bonobo distribution.

The two structures representing a GtkWidget are[5]:

```
typedef struct _GtkWidget        GtkWidget;
typedef struct _GtkWidgetClass   GtkWidgetClass;

struct _GtkWidget
{
  GtkObject object;

  guint16 private_flags;
  guint8 state;
```

[5]Usually the object and class structures are defined in a header file. Exceptionally, the first line of this sample code is in GtkStyle.h (it is a forward declaration), the rest is in GtkWidget.h.

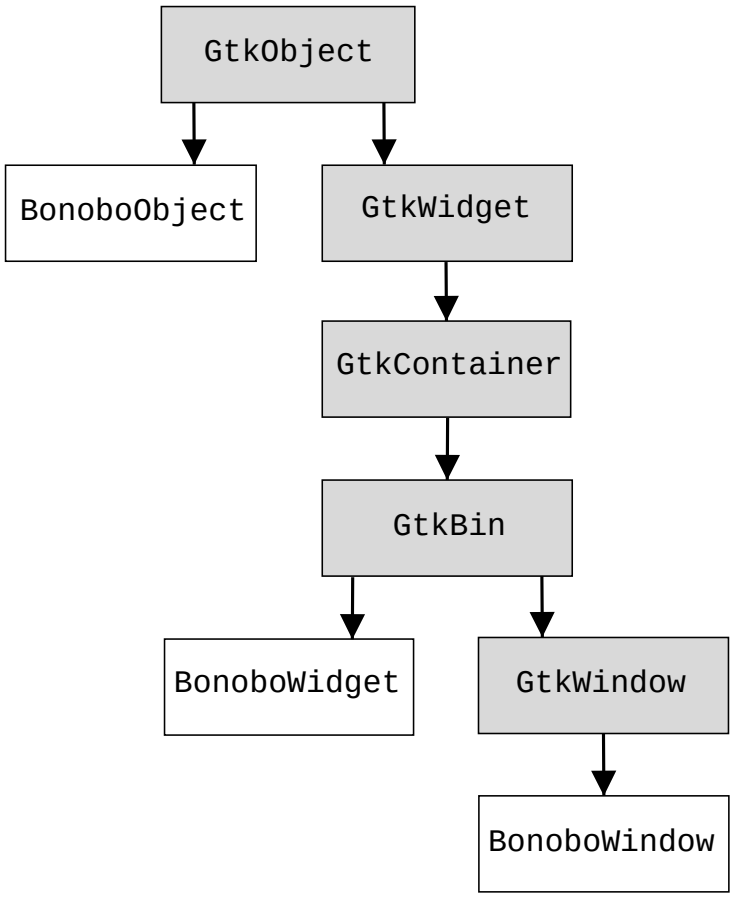

Figure 3.1: A hierarchical tree of some Gtk objects

```
  /* Other members that are not shown in this example */
  GtkWidget *parent;
};

struct _GtkWidgetClass
{
  GtkObjectClass parent_class;

  guint activate_signal;
  guint set_scroll_adjustments_signal;
  void (* show)            (GtkWidget      *widget);
  void (* realize)         (GtkWidget      *widget);
  /* Other members that are not shown in this example */
}
```

This example shows that the GtkWidget class inherits from the GtkObject class. In the GTK+ object system, inheritance is accomplished by using nested structures. The GtkObject structure needs to be the first element in the GtkWidget structure in order for the object mechanism to work correctly. This allows a GtkWidget pointer to be cast to a GtkObject pointer. The same applies for GtkWidgetClass and GtkObjectClass. In fact, the mechanism works well because the C language guarantees that a pointer to a structure is also a pointer to the first member of that structure.

In general, the object structure contains object members and the class structure contains class methods represented by function pointers. The class structure may also contain class members.

The GTK+ object system makes it possible for subclasses to override class methods:

```
static void
gtk_button_class_init (GtkButtonClass *klass)
{
  GtkObjectClass *object_class;
  GtkWidgetClass *widget_class;
  object_class = (GtkObjectClass*) klass;
  widget_class = (GtkWidgetClass*) klass;

  object_class->set_arg = gtk_button_set_arg;
  widget_class->activate_signal = button_signals[CLICKED];
  widget_class->realize = gtk_button_realize;
  /* Only part of the function implementation is shown */
}
```

In the example above, the GtkButton[6] class overrides class methods inherited from GtkObject and GtkWidget. This is possible after casting the GtkButtonClass into GtkObjectClass and GtkWidgetClass.

3.1.3 The GTK+ type system

GTK+ has also a type system that is used by the object system to check the validity of casts. Every Gtk object has a type, and every type is associated with a unique identifier represented by an unsigned integer. The identifier is assigned by the type system when the *gtk_type_unique* function is called.

The *gtk_type_unique* function registers the new type with the type system. It takes the type of the parent object as its first argument. This allows the type system to make a hierarchy of types which can be used to check the validity of

[6]GtkButton is a subclass of GtkWidget which is a subclass of GtkObject.

casts. The validity test is done by calling the *gtk_type_is_a* function which takes two identifiers as arguments and verifies that one identifier is the other's ancestor.

When writing a Gtk object, it is common to have a function that registers the Gtk object with the system. The registration is made on the first call of the function. On subsequent calls, the function simply returns the type's identifier of the Gtk object. Here is what this function looks like for BonoboObject:[7]

```
GtkType
bonobo_object_get_type (void)
{
  static GtkType type = 0;
  if (!type) {
    GtkTypeInfo info = {
      "BonoboObject",
      sizeof (BonoboObject),
      sizeof (BonoboObjectClass),
      (GtkClassInitFunc) bonobo_object_class_init,
      (GtkObjectInitFunc) bonobo_object_instance_init,
      NULL, /* reserved 1 */
      NULL, /* reserved 2 */
      (GtkClassInitFunc) NULL
    };
    type = gtk_type_unique (gtk_object_get_type (), &info);
  }
  return type;
}
```

Note that the *info* structure that is passed to the *gtk_type_unique* function contains

[7] The GtkType represents the type's identifier. It's a typedef for a guint.

two function in its members. These functions allow the initialization of the object and class structures.

The Gtk objects that inherit from BonoboXObject define their *get_type* function by using the *BONOBO_X_TYPE_FUNC_FULL* macro, which is defined in the *bonobo_xobject.h* file.

Having defined the *get_type* function, the creation of a BonoboObject instance is very simple:

```
BonoboObject *object;
object = gtk_type_new (bonobo_object_get_type ());
```

In the object's header file, it is common to have five macros, four of which check the validity of casts. Here are the macros defined in BonoboObject:

```
#define BONOBO_OBJECT_TYPE       (bonobo_object_get_type ())
#define BONOBO_OBJECT(o)
    (GTK_CHECK_CAST ((o), BONOBO_OBJECT_TYPE, BonoboObject))
#define BONOBO_OBJECT_CLASS(k)
    (GTK_CHECK_CLASS_CAST((k), BONOBO_OBJECT_TYPE,
                          BonoboObjectClass))
#define BONOBO_IS_OBJECT(o)
    (GTK_CHECK_TYPE ((o), BONOBO_OBJECT_TYPE))
#define BONOBO_IS_OBJECT_CLASS(k)
    (GTK_CHECK_CLASS_TYPE ((k), BONOBO_OBJECT_TYPE))
```

The second and third macros check the cast validity for objects and classes respectively. The fourth and fifth are used in preconditions to check whether an object is of a certain type. These macros are often used when writing Gtk or Bonobo based applications.

3.2 Basis of Bonobo

3.2.1 The Bonobo_Unknown interface

The Bonobo distribution includes a set of IDL interfaces that we will call **Bonobo interfaces**. The Bonobo interfaces constitute the basis of Bonobo.

The *Bonobo_Unknown* interface is the base interface from which all the other Bonobo interfaces inherit. The Bonobo_Unknown interface is made up of three methods. *ref* and *unref* deal with lifetime management of objects, and *queryInterface* finds out whether or not an object supports an interface and possibly returns a reference to this interface.

Here is the complete Bonobo_Unknown interface:

```
module Bonobo {
     interface Unknown {
          void ref ();
          void unref ();
          Unknown queryInterface (in string repoid);
     };
};
```

The Bonobo_Unknown interface is inspired from the IUnknown interface of Microsoft's OLE2. Indeed, all OLE interfaces are derived from IUnknown.

Here is a simplified version of IUnknown:

```
class IUnknown
{
public:
     virtual HRESULT QueryInterface(REFIID iid,
                                    void** ppvObj) = 0;
```

```
        virtual ULONG AddRef() = 0;
        virtual ULONG Release() = 0;
    };
```

The *AddRef* and *Release* methods in the IUnknown interface correspond respectively to the *ref* and *unref* methods in the Bonobo_Unknown interface. The three methods have a similar role in both interfaces.

Objects lifetime management

Bonobo keeps track of objects by using a system of reference counting. Every time a program starts using an object, the object reference count is incremented by one. When a program finishes using an object, it should release a reference by calling the *unref* method. This causes the object reference count to be decremented by one. When the reference count of an object reaches zero, which means that the object is not being used, the object is simply destroyed.

The queryInterface method

The *queryInterface* method allows to get a different interface on an object that implements the Bonobo_Unknown interface. In other words, *queryInterface* allows a dynamic discovery of the interfaces supported by an object.

As an example, we suppose that we have a reference to a graphical component, which in fact is a reference to a Bonobo_Control object. *queryInterface* can be used on that object to know whether the graphical component supports the Bonobo_Zoomable interface. The returned value is either a reference to a Bonobo_Zoomable object or CORBA_OBJECT_NIL. If it is a reference to a Bonobo_Zoomable object, it means that the graphical component supports the Bonobo_Zoombable interface[8].

[8]It also means that the the reference count of the Bonobo_Zoombale object has been incre-

Here is an interesting use of the *queryInterface* method[9]:

```
Bonobo_Unknown bu1, bu2;
control = bonobo_widget_new_control
     ("file:/home/squallia/Photo.jpg", BONOBO_OBJREF (uic));
bu1 = bonobo_widget_get_objref(BONOBO_WIDGET (control));
bu2 = Bonobo_Unknown_queryInterface
     (bu1, "IDL:Bonobo/Zoomable:1.0", &ev);
Bonobo_Zoomable_setLevel(bu2, .25, &ev);
```

In this example, an image is converted to a graphical component, also called a control, and is zoomed to 1/4 of its original size.

The *queryInterface* has to be an equivalence relation, which means that it should be reflexive, symmetric and transitive[10].

Let's A, B and C be three Bonobo CORBA objects, and IA, IB, IC the interfaces they implement. The queryInterface implementation should guarantee the following properties.

- Reflexive: Querying object A about interface IA should succeed.
- Symmetric: If querying object A about interface IB succeeds, then querying object B about interface IA should succeed.
- Transitive: If querying object A about interface IB succeeds and querying object B about interface IC succeeds, then querying object A about interface IC should succeed.

mented. When the program finishes using the object, it should explicitly release a reference by calling the *unref* method.

[9] To know more about controls see chapter 3.3.1 on page 39.

[10] The *congruence modulo n* relation on the integers is an example of an equivalence relation.

3.2.2 BonoboObject

Bonobo defines a base Gtk+ object called *BonoboObject*. BonoboObject allows the wrapping of CORBA servers as Gtk+ objects. It also implements the Bonobo_Unknown interface. Therefore, it contains the implementation of the *ref, unref* and *queryInterface* methods.

BonoboObject inherits directly from GtkObject (Figure 3.3), which is the base object for the Gtk+ object. It also wraps a Bonobo_Unknown CORBA object. To make this clear, here is the BonoboObject structure:

```
typedef struct {
     GtkObject            base;

     Bonobo_Unknown       corba_objref;
     gpointer             servant;
     BonoboObjectPrivate *priv;
} BonoboObject;
```

Figure 3.2 shows that a BonoboObject wraps a Bonobo_Unknown CORBA object. Obviously this remains true for every object inherited from BonoboObject. In particular, a BonoboControl object wraps a Bonobo_Unknown CORBA object. More precisely, it wraps a Bonobo_Control CORBA object.

An object that inherits from BonoboObject will be called a **Bonobo object**. Therefore, a Bonobo object wraps at least a Bonobo_Unknown CORBA object. All the Bonobo interfaces are implemented as Bonobo objects[11].

[11] Note that some of the Bonobo objects do not implement any Bonobo interface, and that some of the objects included in the Bonobo distribution are not Bonobo objects.

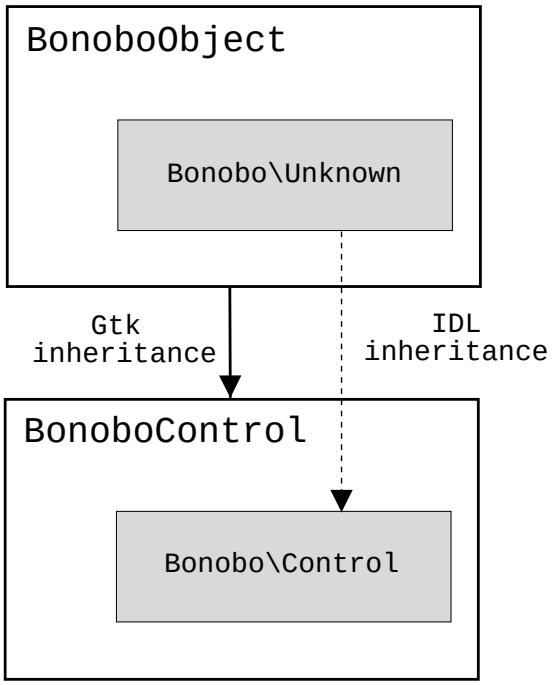

Figure 3.2: Wrapping CORBA objects

Useful functions defined by BonoboObject

BonoboObject provides an important programming interface which is available to all the Bonobo objects[12]. We will examine two functions *bonobo_object_corba_objref* and *bonobo_object_add_interface*.

The *bonobo_object_corba_objref* function takes a BonoboObject as an argument and returns a reference to the wrapped CORBA object. Here is a use of that method:

```
BonoboListener *listener;
Bonobo_Listener corba_listener;
listener = bonobo_listener_new (a_callback_function, NULL);
corba_listener = bonobo_object_corba_objref
                                ((BONOBO_OBJECT)listener);
```

In the example above, we created a BonoboListener object, cast it into a BonoboObject by using the *BONOBO_OBJECT* macro and got a reference to the CORBA object it wraps. The *BONONO_OBJREF* macro is usually used instead of the last line of code:

```
corba_listener = BONOBO_OBJREF (listener);
```

The *bonobo_object_add_interface* function provides a way to associate objects together. Therefore, having a reference to one object makes it possible to get a reference to all its associated objects. The prototype of *bonobo_object_add_interface* is as follows:

```
void  bonobo_object_add_interface (BonoboObject *object,
                                   BonoboObject *newobj);
```

[12] Since by definition, a Bonobo object inherits from BonoboObject

Actually, this function adds the list of interfaces supported by *newobj* to the list of interfaces supported by *object*. Here is a sample code that shows a use of this function[13]:

```
/* EogControl inherits from BonoboControl,
   so it supports the Bonobo_Control interface */
EogControl *control;
/* BonoboZoomable supports the Bonobo_Zoomable interface */
BonoboZoomable *zoomable;
/* The type of the variables are shown above */
bonobo_object_add_interface
            (BONOBO_OBJECT (control),
             BONOBO_OBJECT (control->priv->zoomable));
```

The *control* object is now an *aggregate object* because it combines two BonoboObjects into a single one. This means that an aggregate object wraps at least two CORBA servers. The *control* object will support not only the Bonobo_Control interface but also the Bonobo_Zoomable interface. Therefore, having a reference to the *control* object makes it possible to obtain a reference to a Bonobo_Zoomable object. As a result, the zoom of the object can be changed.

```
Bonobo_Zoomable zoomable;
zoomable = bonobo_object_query_interface
    (BONOBO_OBJECT (control), ''IDL:Bonobo/Zoomable:1.0'');
```

3.3 Bonobo objects

In this section, we will investigate some Bonobo objects that will be used in the implementation chapter. Figure 3.3 shows the hierarchy of some Bonobo objects.

[13]The code was taken from the EOG (Eye Of Gnome) image viewer.

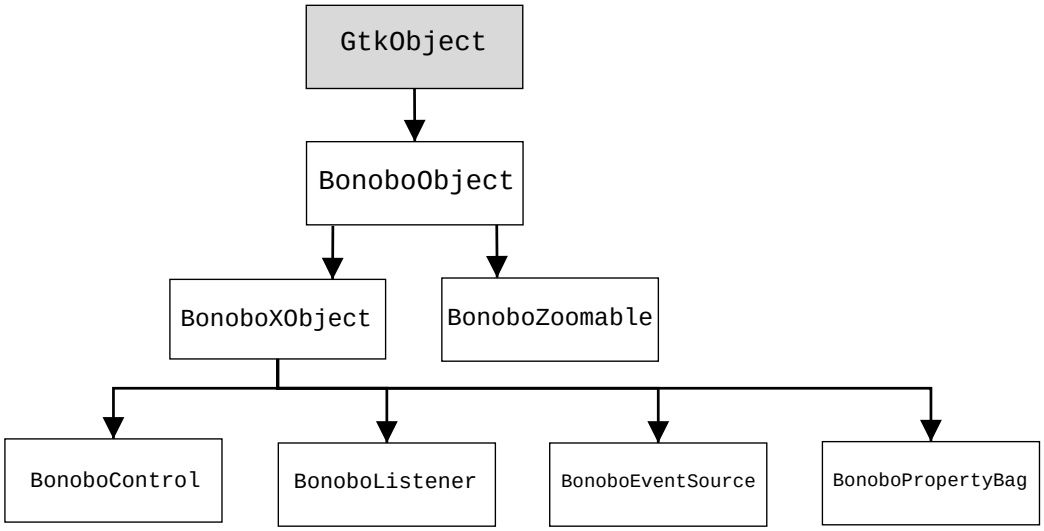

Figure 3.3: A hierarchical tree of some Bonobo objects

It is also useful to have a look at figure 3.1 on page 27, which shows the hierarchy of some Gtk objects.

3.3.1 Controls

Controls are graphical components that are used by container applications. They are created from Gtk widgets, which are user interface elements like buttons. However, unlike the use of Gtk widgets, the use of controls does not require source code nor header files. Controls are used dynamically and allow a dynamic discovery of their properties.

Creating a control

Creating a control from a widget is very simple. Here is the code that allows to create a clock control:

```
BonoboControl *control;
```

```
GtkWidget *clock;
clock = gtk_clock_new (GTK_CLOCK_INCREASING);
gtk_widget_show (clock);
control = bonobo_control_new (clock);
```

Basically, there are three steps for creating a control:

- Creating a widget.
- Calling the *gtk_widget_show* function[14].
- Passing the widget to the *bonobo_control_new* function.

Accessing a control through a CORBA client

A BonoboControl object wraps a CORBA object that implements the Bonobo_Control interface. Since the *bonobo_control_new* function returns a pointer to a BonoboControl object, it is possible to obtain a reference to a Bonobo_Control CORBA object with the following code lines:

```
Bonobo_Unknown *objRef;
objRef = BONOBO_OBJREF (control);
```

Getting the object reference from the control makes it accessible via the network. As a CORBA object, a control can be invoked from a CORBA client program. The program can be run in any operating system and written in any programming language[15]. This is possible since CORBA is the communication layer of Bonobo.

[14]It is important to call this function here in the control implementation. Otherwise the control won't show up when embedded in the container even if the container calls the *gtk_widget_show_all* function.

[15]Provided that the ORB being used has the corresponding bindings for that language

3.3.2 Property bags

A property bag allows a control to expose its functionalities. Consequently, it is a means for a control to communicate with its container. Most of the controls would not be useful without property bags. Another way for accessing a control is to make it support another interface, and then communicate with the CORBA object implementing this interface.

The PropertyBag class provides an implementation of the Bonobo_PropertyBag interface. Therefore, a PropertyBag object wraps a Bonobo_PropertyBag CORBA object. Using this CORBA object it is possible to access a PropertyBag object through a CORBA client.

Creating a property bag

As its name suggests, a property bag contains a list of properties. These properties can be accessed through a get and set callback functions. At the creation of a PropertyBag object the callback functions and a user data object should be passed as parameters. Every time the callback functions are called, the user data object will be passed to them.

The following code shows how to create a property bag:

```
BonoboPropertyBag *pb;
pb = bonobo_property_bag_new (get_prop, set_prop, clock);
```

In this example, *get_prop* and *set_prop* represent the callback functions and clock represents the user data object. Every time the callback functions are called the clock object will be passed to them as a parameter.

Adding properties

After creating a property bag, we should add some properties to it. The *bonobo_property_bag_add* function allows to do that. For example, we can add a property called ”running” whose argument id is 0 and whose type is boolean:

```
bonobo_property_bag_add (pb, "running", 0,
                         BONOBO_ARG_BOOLEAN, NULL, "whether
                         or not the clock is running", 0);
```

The third argument is the argument id. It is common to use an enumeration to represent all the argument ids. It makes the code easier to read and allows the use of switch statements in the implementation of the callback functions. The fifth argument represents the default value of the property, the following one is a documentation string, and the last one represents flags that determine the access rights of the property.

Connecting the control

Now that we have a property bag, we need to connect it to a control:

```
bonobo_control_set_properties (control, pb);
```

Callback functions

To complete the example, we need to write the callback functions. The prototype of the *get_prop* function is as follows[16]:

```
get_prop (BonoboPropertyBag *bag, BonoboArg *arg,
          guint arg_id, CORBA_Environment *ev,
          gpointer user_data)
```

[16]Note that a gpointer is a pointer to void.

At this stage, there is an important feature of property bags that should be mentioned. A property bag does not hold the data of the properties, it is only a model for the properties. This means that the property bag pointer passed to the get_prop function is not sufficient to get the property value, because the property bag does not hold the value. A property value may for example represent a data member of a Gtk widget. This data member would be accessed by calling some function on the widget.

Here is a sample implementation of the *get_prop* function[17]:

```
GtkObject *clock = user_data; /* Casting the user data */
switch (arg_id) {
case 0:
{
    /* The data (whether the clock is running) is held by
       the clock not the bag */
    gboolean running = gtk_clock_is_running (clock);
    /* Setting the out argument arg. */
    BONOBO_ARG_SET_BOOLEAN (arg, running);
    break;
}
}
```

When a property value is requested, the get callback function supplied at the property bag creation time is called.

The set callback function is very similar to the get one. Only two points are worth mentioning:

[17] The is_running function used in this example is not in the clock's API. To have the same result as this fictitious function, we need to make the clock object carry a table of associations.

- The BonoboArg pointer that is passed to the *set_prop* function does not get changed, because the argument is a *const*. The BonoboArg points to the value we want to set.

- The *BONOBO_ARG_GET_BOOLEAN* macro is used in the set callback implementation instead of the *BONOBO_ARG_SET_BOOLEAN* macro.

3.3.3 Containers

A container embeds controls and may interact with them. A container represents the client side of an application.

Embedding controls

BonoboWidget makes embedding controls very easy. The Gtk widget corresponding to the control can be created in one line of code:

```
GtkWidget *control;
control = bonobo_widget_new_control
     ( "OAFIID:Bonobo_Clock", BONOBO_OBJREF (uic) );
```

control can then be used as a normal GtkWidget exactly like a GtkButton or a GtkWindow. The previous code deserves some explanations:

- The real type of *control* is a BonoboWidget, not a GtkWidget. In fact, *control* was cast to a GtkWidget before being returned by the *bonobo_widget_new_control*. The cast is valid because BonoboWidget inherits GtkWidget[18]:

[18]The tree hierarchy is as follows: GtkObject, GtkWidget, GtkContainer, GtkBin, BonoboWidget. See figure 3.1.

- The first argument passed to the *bonobo_widget_new_control* function is a string representation of a moniker object. It is sufficient to know that this string allows the activation of the clock control.

- *uic* is a BonoboUIContainer object. This object should be created and associated with a BonoboWindow object before calling the *bonobo_widget_new_control* function:

```
BonoboUIContainer *uic;
uic = bonobo_ui_container_new ();
bonobo_ui_container_set_engine (uic, bonobo_window);
```

Accessing a property

A BonoboWidget also simplifies the access to the control's property bag. Getting the value of the *running* property is easy:

```
CORBA_boolean state;
bonobo_widget_get_property (BONOBO_WIDGET (control),
                            "running", &state, NULL);
```

Since a property data type might change on the server, it is better for the client to use another method to access a property. First, we get the control frame associated with the bonobo widget, and then we get the property bag from the control frame[19]. Here is how to get the value of the *running* property using the approach described above:

```
Bonobo_PropertyBag pb;
BonoboControlFrame *cf;
```

[19]When creating a BonoboWidget object, the control (which is linked to the property bag) was bound to the control frame. That's why it is possible to get the property bag from the control frame.

```
CORBA_boolean state;
cf = bonobo_widget_get_control_frame
     (BONOBO_WIDGET (control));
pb = bonobo_control_frame_get_control_property_bag
     (cf, NULL);
state = bonobo_property_bag_client_get_value_gboolean
     (pb, "running", NULL);
```

Both ways of accessing properties cause one of the property bag callback functions to be called on the server.

3.3.4 Event Sources and Listeners

Sometimes clients would like to be notified when something happens in the server so that they can perform some action. For example, a container application might want to react when one or more properties of a control have changed. The mechanism that allows this relies on event sources and listeners.

Clients that wish to be notified when an event occurs have to create a listener and register it with an event source. Obviously, it is possible that many listeners register with the same event source.

Let's suppose that there are two containers embedding the same control, and that these containers are both registered with the event source of the control's property bag. Here is what happens when a property changes. The event source is immediately notified of the change. It notifies all the listeners attached to it. As a result, every attached listener emits an *event-notify* signal. Therefore, the callback function corresponding to the signal is called. This means that the clients have finally been notified of the property change.

It is possible to attach a listener to an event source so that it listens to all the events emitted by an event source or only to certain events. For example, a listener

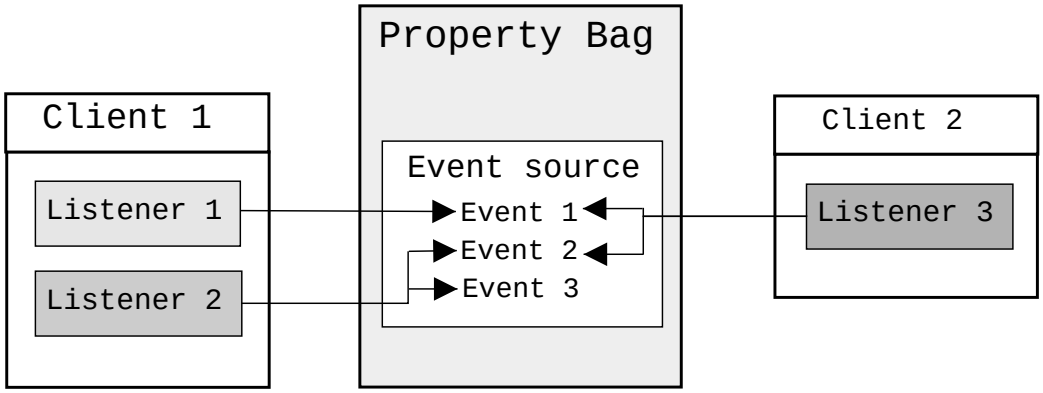

Figure 3.4: Registering listeners with an event source

can be notified when one or more property values have changed. Figure 3.4 shows listeners from two clients registering with some events of the event source. When an event occurs, the attached listeners are notified.

Next, we will see an example that shows the use of event sources and listeners.

Event sources

Typically, the server creates event source objects. These objects can be accessed through CORBA clients. The creation of an event source objects is straightforward:

```
BonoboEventSource *es = bonobo_event_source_new();
```

When creating a property bag, an event source object is created automatically and associated with the property bag. Thus, once a client gets a reference to a property bag, it can get a reference to the event source object associated with it:

```
Bonobo_Unknown es; /*event source*/
/* pb is a Bonobo_PropertyBag CORBA object and ev
   is an initialized CORBA environment */
es = Bonobo_PropertyBag_getEventSource (pb, &ev);
```

Another way to get the same reference is to use the queryInterface function of the Bonobo_Unknown interface:

```
es = Bonobo_Unknown_queryInterface
     (pb, "IDL:Bonobo/EventSource:1.0", &ev);
```

An event source is responsible for notifying all the listeners attached to it when an event occurs. For example, when a property value changes by a set callback function[20], the *bonobo_property_bag_notify_listeners* function is called and calls in its turn the *bonobo_event_source_notify_listeners* function. Consequently, all the registered listeners will be notified. However, when a property value changes by another method, the *bonobo_property_bag_notify_listeners* function should be called explicitly. Otherwise, the property bag will not be notified of the change of the property value[21].

Listeners

In order for the notification mechanism to work, the client needs to create one or more listeners and register them with the event source:

```
BonoboListener *listener;
Bonobo_Listener corba_listener;
/* prop_change_cb is a callback function and clist is the
   the user data that will be passed to it */
listener = bonobo_listener_new (prop_changed_cb, clist);
/* Requesting the CORBA object
 of the 'listener' Bonobo object */
```

[20] See section 3.3.3, page 45 on accessing properties.

[21] Remember that a property bag is just a model for the properties. It does not hold the data itself. The data might be changed from other programs. See section 3.3.2, page 41 for details on property bags.

```
corba_listener = BONOBO_OBJREF (listener);
Bonobo_EventSource_addListenerWithMask
     (es, corba_listener,
      "Bonobo/Property:change:running", &ev);
```

As in the example above, a listener should always be bound to a callback function, so that the listener is able to call the callback function whenever it is notified of an event (by the event source). Here is the prototype of a listener callback function:

```
void prop_changed_cb (BonoboListener *listener,
                      char *event_name,
                      CORBA_any *any,
                      CORBA_Environment *ev,
                      gpointer user_data);
```

where *any* is the value that is passed by the event source, and *event_name* is the event name that has been emitted by the event source.

The *addListenerWithMask* function specifies that the listener wants to be notified of certain events only. In the example above, the listener wants to be notified when the *running* property changes. The string in the third argument is called the event mask. It is a comma-separated list of event names the listener is interested in receiving.

Event names are generally made up of three parts. The first part consists of the interface that triggered the event. Its purpose is to create a unique namespace. The second part indicates what happened, and the third part is used to make events more specific.

Here is the code that allows a listener to be notified when one of the *running* or *test* property change:

```
Bonobo_EventSource_addListenerWithMask
     (es, corba_listener,
      "Bonobo/Property:change:running,
       Bonobo/Property:change:test", &ev);
```

The *addListener* function should be used in order to notify a listener of all the events emitted by the event source:

```
Bonobo_EventSource_addListener (es, corba_listener, &ev);
```

Chapter 4

Design

In this chapter, we will examine the design issues of the IIOP to SOAP bridge. We will discuss the different possibilities of building such a bridge and will justify the decisions being taken.

4.1 Requirements and assumptions

The bridge should be designed so that a CORBA client and a CORBA server would communicate seamlessly over HTTP. Particularly, no modifications are allowed in the CORBA client or server in order for the communication to take place.

We will ignore the bootstrapping problem which consists of getting the IOR of the naming service[1], and we will assume that the CORBA client has access to the IOR of the object it wants to invoke operations on[2].

[1] The IOR of the naming service is usually used for getting IORs of other objects.

[2] The object may be the Naming Service itself.

4.2 Intercepting an IIOP message

Once a CORBA client has the IOR of an object and knows the IDL interface of that object, it can invoke operations on it. The client, and more precisely the stub, will use the information encapsulated in the IOR in order to create a TCP connection to the object server. It will then create an IIOP message and send it over this connection.

On the client side, the purpose of the bridge is to intercept the IIOP message, convert it to a SOAP message and send it over HTTP. There are two ways to intercept the outgoing IIOP message:

- Changing the stub so that it establishes a TCP connection with the bridge instead of the CORBA server. Here, the problem is addressed at the end: the CORBA client communicates with a modified stub.

- Changing the object's IOR so that it points to another endpoint address[3]. A server in the appropriate machine will be listening to the appropriate port number waiting for IIOP messages. We will call this server the **proxy server**, because it acts like a proxy to the clients wishing to contact CORBA servers. Here, the problem is addressed at the beginning: the CORBA client reads a modified IOR. Figure 4.1 shows that the CORBA client reads IOR' with endpoint address E2 instead of the IOR generated by the CORBA server.

Both solutions involve a change to be made:

- The first approach presents a big disadvantage, because it changes something that is ORB specific namely the stub. Indeed, different ORBs generate different stubs for the same IDL interface. In addition, the stub is language

[3]By endpoint address, we mean an IP address and a port number.

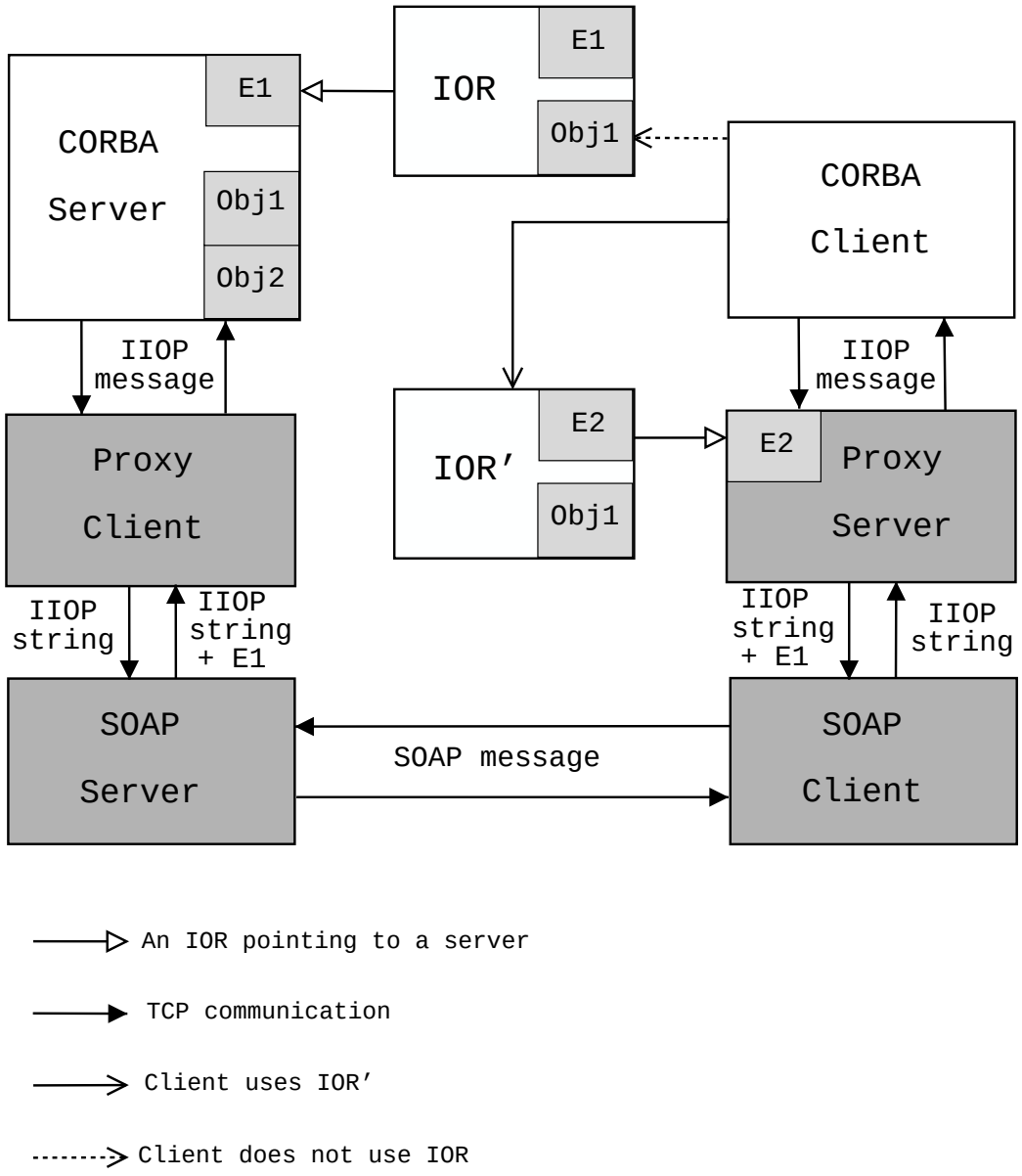

Figure 4.1: IIOP to SOAP bridge

dependent: the same ORB will generate different stubs for the same IDL interface depending on the language binding chosen. This means that a parser should be written for every ORB and for every language binding supported by that ORB. Furthermore, a slight modification in the ORB, and more precisely in the IDL compiler, means that the parser should be changed. And finally, this option will not be able to deal with new ORBs, except by writing new parsers.

- The second approach is more logical, because it relies on changing something that is standard namely the IOR. Once an IOR parser has been written, it will work for every ORB, even for new ones.

For the reasons above, we will adopt the second approach. The mechanism described therein makes the client think that the CORBA object is on the proxy server. Therefore, the client will create a connection to the proxy server and will send an IIOP message over this connection. The IIOP message is indeed intercepted by the proxy server.

4.3 The Proxy Server

The proxy server should take an IIOP message, which is a sequence of bytes, and convert it to a string containing a decimal representation of the bytes. Here is an example of such a string:

```
071 073 079 080 001 000 000 000 000 000 000 232 000 000 000 002
000 000 000 001 000 000 000 012 000 000 000 000 000 001 000 032
000 001 001 000 000 000 000 006 000 000 000 140 000 000 000 000
000 000 000 040 073 068 076 058 111 109 103 046 111 114 103 047
083 101 110 100 105 110 103 067 111 110 116 101 120 116 047 067
```

```
111 100 101 066 097 115 101 058 049 046 048 000 000 000 000 001
```

The first four bytes represent the word GIOP[4]. Every IIOP message starts with the GIOP characters.

In brief, the proxy server is responsible for:

- Changing the IORs so that they point to the endpoint address of the proxy server.
- Listening to a TCP port for incoming IIOP messages.
- Converting every IIOP message to a string and sending it to the SOAP client.

Changing the IORs might equally be done on the server side, before the IORs are made available to the client.

4.4 From the SOAP server to the CORBA server

SOAP makes it possible to do a Remote Procedure Call over HTTP. The SOAP client passes the string representation of an IIOP message to the SOAP server as an argument of a method. The remote method invokes what we will call the **proxy client**. When the proxy client receives the string representation of the IIOP message, it should convert it back to a byte stream, exactly like the original IIOP message that has been intercepted by the proxy server. Then, it should forward the IIOP message to the appropriate CORBA server. The CORBA server gets the IIOP message and thinks that it is coming from a CORBA client. Therefore, it prepares an IIOP response message and sends it to what it thinks is the CORBA client. Actually, the message is sent to the proxy client.

[4]GIOP stands for General Inter-ORB Protocol. GIOP is the protocol used for the communication between two ORBs. It is defined without any mention to the underlying network protocols. IIOP is the implementation of GIOP over TCP/IP.

Note that on the server side, the bridge is made up of the SOAP server and the proxy client, and that on the client side, the bridge is made up of the proxy server and the SOAP client (See figure 4.1).

4.5 Problems and their solutions

In this section, we will discuss three problems relating to the bridge and their solutions.

Problem 1

A problem remains in the system described so far. Upon receiving an IIOP message, the proxy client should know the endpoint address of the CORBA server to which the IIOP message is destined.

This piece of information is included in the original IOR: the one that has been created by the CORBA server. Therefore, before changing the IOR, the proxy server should extract the endpoint address from it and store it somewhere. When receiving an IIOP message from the CORBA client, the proxy server should pass it along with the endpoint address to the SOAP client. Finally, the SOAP client will send these two elements (ie the IIOP message and the endpoint address) to the proxy client.

Problem 2

The current system works perfectly as long as there is only one server. If there are two servers, the proxy server as it was described, has no way of telling if an IIOP message is destined to one server or to the other.

The problem may be solved by making the proxy server not only store the endpoint address, but also the object key of every IOR. The proxy server should

maintain a hashtable containing an object key and its corresponding endpoint address.

Every IIOP request message includes an object key that allows the CORBA server to know which object is concerned by the invocation. Therefore, the solution to the problem stated above is to extract the object key from the IIOP message, and to look up its corresponding endpoint address in the hashtable. After that, the endpoint address and the IIOP message should be forwarded until they arrive to the proxy client.

Problem 3

In all what has been said so far, we assumed that the client has access to an initial IOR. It may be the IOR of the naming service or any other CORBA object.

Having a first IOR that enables it to access a first object, the client may get other IORs by invoking methods on that object, and is more likely to do so. The next IORs the client receives will be included in the IIOP reply messages. Inside these IORs, there will be the endpoint address of the real CORBA server, not that of the proxy server. With the current system, the client will use the bridge until it receives an IOR included in an IIOP reply message. At that point, the client will try to connect directly to the server.

This problem can be solved the same way IIOP messages are intercepted. When the proxy server receives an IIOP message from the SOAP client, it should check if there is an IOR inside the IIOP message. If it is the case, the IOR should be extracted and modified so that it points to the endpoint address of the proxy server. Then, the proxy server should add a new entry to its hashtable for taking the new IOR into account. And finally, it should put the modified IOR back into the IIOP message and send it to the CORBA client.

In addition to the three points mentioned in section 4.3, the proxy server is

also responsible for:

- Maintaining a hashtable from all the parsed IORs. There should be one entry for every parsed IOR. And every entry should contain an object key and its corresponding endpoint address .
- Extracting the object key from an IIOP request message and looking up its corresponding endpoint address in the hashtable.
- Sending not only the IIOP message but also the endpoint address to the SOAP client.
- Modifying every IIOP reply message that includes an IOR by changing that IOR, and adding a new entry to the hashtable accordingly.

In fact, practically all the tasks of the proxy server can be carried out by the proxy client. In this case, the proxy server's responsibilities will be reduced: it will have to listen to an endpoint address for incoming IIOP messages, to convert them to strings and to send them to the SOAP client. All the other tasks should be performed by the proxy client (the third responsibility described in the previous paragraph becomes irrelevant).

Chapter 5

Implementation

This chapter is about implementing and testing the IIOP to SOAP bridge. First, we will examine a Bonobo application. Then, we will see how to access it through CORBA clients. And finally, we will look into some implementation issues of the bridge.

5.1 Bonobo application

In this section, we will investigate the implementation details of a Bonobo application. This application is a modified version of a Bonobo sample program that is part of the bonobo-1.0.4 distribution. The Bonobo application is written C[1]. Figure 5.1 shows a screenshot of the Bonobo container application.

The application consists of two executables:

- *bonobo-controls* that acts like a server. It is a control factory, which means that it is capable of producing controls on request. It can produce two types of controls: a calculator control and a clock control.

[1]Bonobo has only a C API.

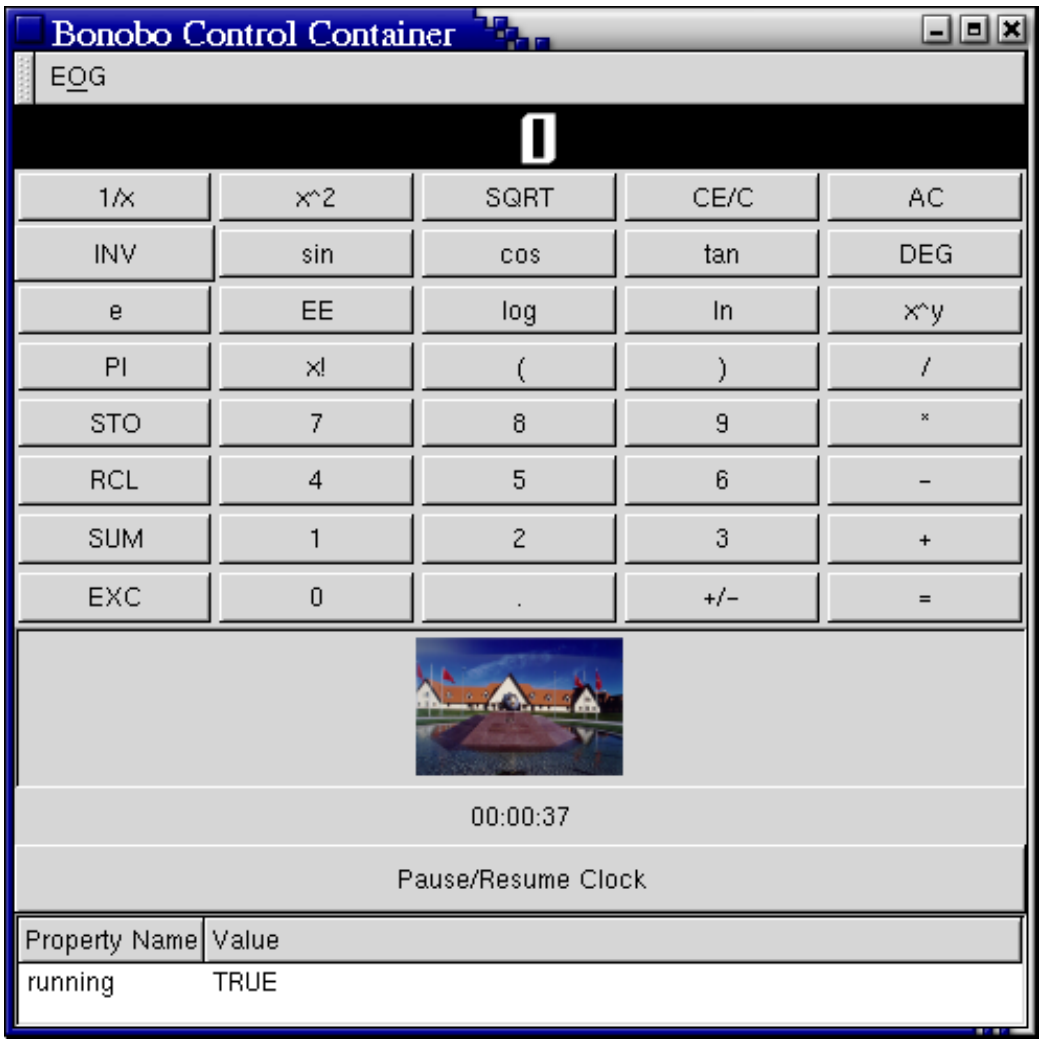

Figure 5.1: Screenshot of a Bonobo container application

- *control-container* that acts like a client. It is the container application. It uses the bonobo-controls server in order to create a calculator control and a clock control. It also creates an image control.

Many containers can use the same control factory, the same way many clients can use the same server. However, every container has a different instance of a control.

5.1.1 The server's controls

Initialization steps

The bonobo-controls program starts up by initializing GNOME, OAF and Bonobo. Then it registers the factory with OAF.

During start up, most applications will have to initialize GNOME, OAF, Bonobo and register their Bonobo object factory with OAF. As these steps are used frequently, two macros were defined for that purpose. The BONOBO_OAF_FACTORY macro is used by a factory that produces one type of controls, while the BONOBO_OAF_FACTORY_MULTI macro is used by a factory that produces many types of controls. Both macros define the *main* function of the program.

Here is the use of such a macro in the bonobo-controls program:

```
BONOBO_OAF_FACTORY_MULTI
            ("OAFIID:Bonobo_Sample_ControlFactory",
             "bonobo-controls", "1.0.4",
             control_factory, NULL)
```

The first argument is a string representation of a moniker. The second last argument is a gnome factory callback function. The last argument is the user data that is passed to it. The second and third argument are the description and version of the program.

The gnome factory callback function returns a pointer to a BonoboObject. The object returned depends on which type of the two controls you want to create. Here is what this function looks like in our program:

```
static BonoboObject *
control_factory (BonoboGenericFactory *this,
                 const char           *object_id,
                 void                 *data)
{
   BonoboObject *object  = NULL;
   if (!strcmp (object_id, "OAFIID:Bonobo__Clock"))
       object = bonobo_clock_control_new ();
   else
   if (!strcmp (object_id, "OAFIID:Bonobo_Calculator"))
       object = bonobo_calculator_control_new ();
   return object;
}
```

Clock control

The clock control represents a running clock. We want the container application to be able to stop and start the clock. The usual way to do that is to attach a property bag to the control. Then, for this particular case, we need to add a boolean property to the property bag. We will call this property the *running* property.

As the clock widget does not have a data member that shows whether the clock is running, a pointer to a variable should be associated with it (before converting it to a control). Actually, every Gtk object can carry around a table of associations from strings to pointers. Here is the way to do it:

```
gtk_object_set_data (GTK_OBJECT (clock), ‘‘Running’’,
```

```
                                GUINT_TO_POINTER (1));
```

Except from this detail, the implementation of the clock control is as specified in section 3.3.1 on “Controls”.

Calculator control

We should be able to use the calculator control remotely from a CORBA client. Therefore, pressing a button on the client calculator Graphical User Interface (GUI) should have a similar effect on the calculator control, as if the button was pressed on the local container GUI. In order to do that, we need to slightly modify the gnome calculator[2]. We need to add a function to its interface to simulate the click of a specific button. This function is called *gnome_calculator_press*. It takes an index of a button as an argument, and makes the corresponding button emit the *clicked* signal.

Now, we can add two properties to the control’s property bag. One will represent a calculation result, and the other will represent a button index. Finally, here is the set callback function of the property bag:

```
switch (arg_id) {
case RESULT:
{
   gnome_calculator_set (gc, BONOBO_ARG_GET_DOUBLE (arg));
   break;
}
case BUTTON_INDEX: {
   gnome_calculator_press(gc, BONOBO_ARG_GET_INT (arg));
   break;
```

[2]The gnome calculator is a Gtk widget

```
}
}
```

5.1.2 The container

The container starts up by initializing GNOME, OAF and Bonobo. Then, it creates a window and three controls: a calculator control, a clock control and an image control. These are created as described in section 3.3.3 on “Containers”. Next, the window inserts the controls via a Gtk box and every control’s IOR is printed to a file.

Here is a way to print a control’s IOR:

```
Bonobo_Unknown bu;
char *ior;
bu = bonobo_widget_get_objref(BONOBO_WIDGET (control));
ior = CORBA_ORB_object_to_string(orb, bu, &ev);
```

It is worth mentioning that two types of string representation of monikers are used when creating the controls. One type for the calculator and clock controls like *OAFIID:Bonobo_Calculator* and *OAFIID:Bonobo_Clock*, and the other for the image control like *file:/home/squallia/Photo.jpg*. The first type activates the bonobo-controls server and the second activates the image viewer Eye Of GNOME (EOG). The activation is automatic due to the OAF architecture.

Before the container application shuts down, it should release all the object references. If no other client programs are using EOG and bonobo-controls, both applications will shut down automatically. In the case of our example, we should release the reference to the Bonobo_Unknown object that was used for printing the IORs of the controls:

```
bonobo_object_release_unref(bu, NULL);
```

The container application has also two features:

- A button that is used to stop and start the clock.
- A text field that shows whether the clock is running.

Starting and stopping the clock means accessing the property bag of the clock. And having a text field that shows the state of the clock means that a listener should be attached to the event source of the property bag[3].

5.2 CORBA clients

The Bonobo application analysed in the previous section is written in C, runs on a Linux operating system and uses the ORBit ORB. The objective is to write Java programs running on a Windows platform and using the Java IDL ORB in order to access the Bonobo application.

The task is actually not as hard as it seems, the two ORBs will do most of the work. However, the bootstrapping between the two ORBs is either impossible or very hard to achieve. In the case of our example, bootstrapping is about getting the IOR of the ORBit's naming service. In order to avoid this problem, we suppose that the Java programs have access to the files where the IORs of the three controls were written[4]. Each Java program will act as a CORBA client while the Bonobo application will act as a CORBA server.

The first thing to do is to use the Java IDL compiler *idlj* in order to generate the stubs corresponding to the Bonobo interfaces[5].

[3] See section 3.3.4 on page 46 for the use of event sources and listeners.

[4] The assumption made means that ORBit's naming service won't be used. It is possible to have a minor assumption which consists of only getting the IOR of the naming service.

[5] Some IDL directives like *ifdef* should be removed from the IDL files since they generate compilation errors.

The three client programs start with initializing the ORB. Then, they read the string representation of a control's IOR from the appropriate file, and convert it to an IOR.

Now, we'll examine each of the three CORBA clients:

- **Clock:** Having the IOR of the clock control, we use the *getProperties* operation to get a reference to the control's property bag. Next, we call the *getPropertyByName* operation to get the reference of the *running* property. Finally, we get the boolean value, change it in order to stop or run the clock, and release the reference to the property bag.

- **Calculator:** The program presents a calculator GUI. Its logic is similar to the clock one. Every time a button is pressed, the *button_index* property is set with the new index. Then, we get the value of the *result* property so that the calculation result can be shown on a text field.

- **Image:** Having a reference to the image control, we use the queryInterface operation on the *IDL:Bonobo\Zoomable:1.0* interface. Now that we have a reference to a zoomable object, we can call the *setLevel* operation where the zoom level is specified as an argument.

The three CORBA clients are interacting with the Bonobo application using IIOP messages. Integrating the bridge should enable them to interact in the same manner but using SOAP messages instead.

5.3 IIOP to SOAP Bridge

The IIOP to SOAP bridge consists of two parts. One part is responsible for communicating with the CORBA client and the other is responsible for communicating with the CORBA server. The first part is made up of the proxy server and the

SOAP client, and the second part is made up of the SOAP server and the proxy client (See figure 4.1). Because the SOAP implementation used has a Java API, all the bridge was implemented in Java.

5.3.1 From the SOAP Client to the SOAP Server

To implement a SOAP client and server, we need a SOAP implementation. The SOAP implementation used is Apache SOAP.

As far as the bridge is concerned, the SOAP client and server are responsible for making an RPC over HTTP. On the client side, the SOAP API provides a way to make SOAP requests and interpret SOAP responses. On the server side, the Apache SOAP installation requires a Web application server that supports servlets and JSPs. Apache Tomcat is the one that has been used. Both on the server and the client side an XML parser such as Apache Xerces is needed.

Writing a SOAP client for making an RPC is actually quite easy. Basically, we will have to create a Call object[6] and set some of its elements:

- Setting the target Uniform Resource Identifier (URI) which will enable the Call object to identify the SOAP service being invoked.
- Setting the name of the method that should be invoked. Obviously, the method has to be exposed by the SOAP service.
- Setting the parameters of the method.

Then, we should call the *invoke* method on the Call object. The endpoint address of the SOAP service should be passed as a parameter to the *invoke* method. A Response object returned by this method allows to extract the result of the RPC.

[6]The Apache SOAP Call object is the main interface to the underlying SOAP RPC code.

Writing the SOAP server is also straightforward. First, we will have to create a Java class which contains the methods to be exposed as a SOAP service. Then, we should create a deployment descriptor for the SOAP service. The descriptor consists of the name of the Java class providing the SOAP service implementation and the name of the methods to be exposed.

Now the first element of the bridge has been built, but there are still two other elements: the proxy server which will communicate with the SOAP client and the proxy client which will communicate with the SOAP server.

5.3.2 The Proxy Server

The proxy server provides all the functionalities mentioned in the design chapter. It is made up of four classes:

- The IOR class allows to extract and modify the object key and endpoint address of a string representation of an IOR. The byte order is an important factor in parsing and modifying the IOR.
- The IIOP class allows to know the byte order of the machine sending the IIOP message and to check whether this message carries an IOR.
- The Parser class parses all IOR files in a certain directory, put their object keys and endpoint addresses in a hashtable, and creates new IOR files by changing the endpoint address of previous ones. The Parser class is also responsible for changing IORs coming within IIOP messages and adding entries to the hashtable accordingly. Finally, it provides a method that returns the endpoint address corresponding to an object key.
- The Bridge class listens to the port number to which the IIOP messages will be redirected. It contains the *main* method and makes use of the three classes described above as well as the SOAP client.

5.3.3 The Proxy Client

The SOAP client passes an IIOP string and an endpoint address to the proxy client. The proxy client turns the IIOP string into an IIOP message and send it to the specified endpoint address. Again, the byte order is important while reading the IIOP message.

Chapter 6

Conclusion

6.1 Work achieved

As stated in the introduction section, the goal of the project was to build an IIOP to SOAP bridge and vice versa. This was motivated by the fact that the bridge will enable us to access Bonobo components behind a firewall, and that security features are easier to build on top of HTTP.

The project goal has been achieved since a fully Java implemented IIOP to SOAP bridge has been built. Furthermore, the bridge has been tested with three CORBA clients and three Bonobo controls embedded in a Bonobo container. The test showed that all the communication between the CORBA clients and the Bonobo controls is routed via the bridge. In other words, the HTTP protocol is used instead of IIOP.

In addition to that, this thesis provides a good starting point for understanding many aspects of the Bonobo technology.

6.2 Difficulties encountered

The most difficult element of the project was understanding the Bonobo technology. Indeed, there is a complete lack of Bonobo documentation. Apart from a short ubiquitous tutorial on writing Bonobo controls[1] and some Bonobo sample applications shipped with the Bonobo distribution, there is practically no other technical resources[2]. Even the Bonobo API is only partially documented. This lack of information is mainly due to the fact that Bonobo is a new and free technology. As a result, examining Bonobo code was necessary to comprehend the technology.

On the other hand, understanding the Bonobo sample code was not easy since it contains GTK+[3], ORBit and Bonobo code.

Finally, there were two implementation problems. The first one was about changing the string representation of IORs, so that they point to a specific endpoint address. And the second one was about dealing with the byte order.

6.3 Further work

The work that has been achieved in this thesis can be extended in many ways:

- Currently, the bridge only supports one CORBA client at a time. Other CORBA clients wishing to use the bridge have to wait until the first CORBA client finishes its communication. Thus, an important feature to the bridge would be to enable it to deal with many CORBA clients at the same time.

- The performance of the bridge can be increased. Indeed, the size of the

[1] The tutorial is in `http://www.djcbsoftware.nl/projecten/bonobo_controls/`. It was written on the 15/04/2001 and updated afterwards.

[2] However, there are a few general papers on the Bonobo component architecture.

[3] The code uses the GTK+ library including the GTK+ object and type systems.

SOAP message being sent across the network can be halved. This can be achieved by using the hexadecimal representation of an IIOP message instead of the decimal one and by removing all spaces from it.

- SOAP messages can be encrypted using SSL and SOAP Security Extensions. This will provide important security features such as confidentiality and non-repudiation.

A C implementation of the bridge would be an interesting project since it will greatly enhance the performance. The project would be easier for two reasons. Firstly, many SOAP implementations based on C are becoming mature. Secondly, the principles behind the bridge have been laid out in this thesis.

Finally, accessing and manipulating a Gnumeric spreadsheet from the Web would also be a useful project with many practical applications. The project will rely on the bridge in order to use SOAP for communicating with Bonobo components. The project will be especially interesting since the Bonobo technology is being more integrated into the Gnumeric spreadsheet.

Bibliography

[1] http://orbit-resource.sourceforge.net/.

[2] http://www.gtk.org.

[3] Soap frequently asked questions. http://www.develop.com/soap/soapfaq.htm.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. http://www.w3.org/TR/SOAP/, May 2000.

[dI99] Miguel de Icaza. Bonobo. http://www.ximian.com/devzone/tech/bonobo.html, 1999.

[Gwy00] Telsa Gwynne. Gnome frequently asked questions. http://www.gnome.org/faqs/users-faq/, 2000.

[LZM] Todd Graham Lewis, David Zoll, and Julian Missig. Gnome frequently asked questions. http://canvas.gnome.org:65348/gnomefaq/html/x703.html.

[Obj] Object Management Group, Inc. http://www.omg.org/.

[Pop98] Alan Pope. *The CORBA Reference Guide, Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1998.

[She00] John R. Sheets. *Writing GNOME Applications*. Addison-Wesley, September 2000.