Wonderwall Administrator's Guide

IONA Technologies PLC
December 1997

IONA Technologies PLC
The IONA Building
8–10 Lr. Pembroke St.
Dublin 2
Ireland

Phone: +353-1-662 5255
Fax: +353-1-662 5244

IONA Technologies Inc.
60 Aberdeen Ave.
Cambridge, MA 02138
USA

Phone: +1-617-949-9000
Fax: +1-617-949-9001

IONA Technologies Ptv. Ltd.
Ashton Chambers, Floor 3
189 St. George's Terrace
Perth WA 6000
Australia

Phone: +61 9 288 4000
Fax: +61 9 288 4001

Support: support@iona.com
Training: training@iona.com
Orbix Sales: sales@iona.com
IONA's FTP site: ftp.iona.com
World Wide Web: http://www.iona.com/

Contents

Chapter 1
Getting Started

The Internet Inter-ORB Protocol (IIOP), introduced as part of the CORBA 2.0 General Inter-ORB Protocol (GIOP), has paved the way for using distributed CORBA objects over the internet. This opening up of the internet, however, brings its own problems and risks. It is a potentially dangerous environment—there will always be a few users willing to exploit any security loopholes to cause damage to your system. Some level of security is necessary to keep these intrusions at bay. A popular approach to internet security is to use a *firewall* to restrict access to hosts on your local network. The basic model is to direct all traffic to and from the internal network through a single point which can monitor and control every transmitted message

There are firewalls currently available which restrict access to a local network in a variety of ways—for example, access to certain hosts and certain commands can be limited. However in a distributed object environment, such as CORBA, it is important that security be object oriented. Experience has shown that it is a bad policy to implement security which is too coarse-grained. Users presented with a choice between two levels of security, one which is too restrictive and another which is too permissive, will inevitably choose the permissive level of security on occasion—and consequently a breach appears in the network defences.

The Wonderwall was developed according to the firewall model in order to address the security issues arising from using CORBA over the internet. It provides a flexible approach to security and is object-oriented, allowing control of access to individual objects even down to the level of individual operations on objects.

1.1 Introduction

The usual approach to building a firewall involves restricting internet access to a single IP port on a single host for each service. This host will be the only host which is physically connected to the internet and the restriction to using a single well known IP port provides an additional safeguard.

A popular refinement of this model involves making the firewall host a dedicated, secure host, known as a *bastion host*. The bastion host is dedicated exclusively to the role of gateway to the internet and its configuration can be toughened to make it extra secure against unwanted incursions. This approach has the clear advantage that much of the security effort can be focused on this one machine. For example, many directories on the bastion host can be made read only to `root` without inconveniencing anyone.

The firewall is usually implemented as a proxy server process which runs on the bastion host (as is done, for example, with HTTP and SMTP firewalls). Wonderwall follows this pattern and implements an IIOP proxy server. The role of the server process is to listen to incoming messages on the well-known IP port and to pass on these messages to the internal network, after subjecting them to close scrutiny. When any potentially hostile or forbidden messages are encountered, these are blocked and not passed on to the internal network.

The Wonderwall has the following features which contribute to strengthening the security of the internal network:

- Use of bastion host facilitated.
- All messages filtered.
- Filtering of messages based on Request header.
- Fine-grained control of security.
- Support for message encryption.
- Message logging.
- Messages blocked unless specifically allowed.
- Simplicity of proxy server.

The model on which the Wonderwall is built supports the use of a bastion host as the basis of your firewall. You have only to install the Wonderwall server on the bastion host and it will act as a liaison between the outside world and your internal network. Alternatively, you can install Wonderwall on a regular host if you prefer.

All messages arriving on the server's well known port will be filtered. To be specific, the Wonderwall is not just a facility to monitor initial connections to CORBA objects, it will continue to monitor (and potentially block) all messages which pass between an external client and the internal CORBA object.

A number of message types are defined for the IIOP protocol and any or all of these can be blocked if necessary. The most important group of incoming messages are the Request messages which are used to invoke methods on CORBA objects. The Wonderwall provides comprehensive filtering of these messages based on the content of the Request message header. This header provides all of the information needed to provide effective filtering. For example, the identity of the target object and

the intended operation name. Request messages can be checked rapidly and passed to the internal network with little performance overhead.

Wonderwall provides the kind of fine-grained control of security which is needed for a distributed object environment. It allows you to control access to individual objects and, moreover, to allow or deny access to specific methods defined on that object. There are a number of other criteria which can be checked as will be seen later.

Message encryption is an essential feature needed for the exchange of private messages over the internet. Wonderwall supports the exchange of encrypted IIOP messages using the Orbix *transformer* mechanism. This allows the programmer to encrypt a complete IIOP message using a custom encryption algorithm.

The logging facility of Wonderwall (which can be configured to focus on particular kinds of events) is a powerful facility for tracing the history of suspicious message exchanges. It is also broadly useful as a debugging and monitoring facility.

Some general features of good security practice are observed in the Wonderwall. The approach taken to filtering, for example, is that everything is forbidden unless it is expressly allowed. Another principle is that a proxy server ought to behave simply and predictably. Too many security loopholes have arisen when a large, complex application is connected directly to the internet. The Wonderwall server, by contrast, is a simple stand-alone process, which requires no special privileges, forks no processes and interacts with the bastion host in a simple manner.

The rest of this chapter explains how to set up and configure the Wonderwall with the minimum of fuss. It takes, as an example, the case of an OrbixWeb client talking to an OrbixWeb or Orbix C++ server. The Wonderwall is also fully interoperable and the issues associated with interoperability are discussed in Chapter 3.

Before learning how to use Wonderwall, it is necessary to have an elementary understanding of the IIOP protocol itself.

1.2 Overview of IIOP

The IIOP protocol specifies the way in which CORBA messages are encoded for transmission. In particular, it specifies a universal format for the transmission of operation invocations across the internet. This makes it possible for clients of one ORB to send operation invocations to any ORB across the internet, and also to correctly interpret any return values received.

When an IIOP client sends a message to a remote object, it requires an Interoperable Object Reference (IOR) which stores the addressing information for that object. For the IIOP protocol, an IOR will include the following information:

- The name of the host on which the object resides.
- The port it listens to.
- Its object key (a string of bytes identifying the object).

When an IIOP client has the remote object's IOR, it opens a TCP connection to the host and port named therein and can send and receive messages along this connection. If multiple objects use the same host and port, the client can use the same connection to communicate with the other objects.

The IIOP model is based around two main message types: a Request and a Reply. Clients send Requests, and servers send Replies. There's also a set of message types used to handle unexpected error conditions or timeouts. See section 3.4 on page 24 for information on these.

In the same way that a filtering router can filter packets based on the packet header, the Wonderwall filters incoming Requests based on the following information gleaned from the message header:

- The message type.
- The IP address of the client.
- The object key of the object being accessed.
- The name of the operation being invoked.
- The principal of the client's invoker.
- Any IOP Service Contexts.

See section 2.2 on page 12 and Appendix A for more details of the filtering mechanism and how it's specified. The body of Request messages cannot be filtered without knowledge of the Interface Definition Language (IDL) used to define the operations and parameters for each object, so only the message header parameters can be used in a filter.

In the present version of Wonderwall, any Reply messages which pass from the internal server out to the client are not filtered.

The basic component of Wonderwall is the executable `iiopproxy`. This process is intended to run on the bastion host listening for IIOP requests on a specified TCP port. Any requests which arrive on this port from external hosts are filtered so that access can be restricted to certain CORBA objects or operations behind the firewall.

You can control the filtering of packets by editing the configuration file `iiopproxy.cf`. This file allows you to specify a flexible set of rules for either allowing or denying access to certain objects or operations.

Once a given request has been allowed through the firewall, the process `iiopproxy` will forward it to the proper location on the internal network. The `iiopproxy` does this by looking up its own database of IORs which include all the externally accessible CORBA objects.

In this chapter, an example of a configuration file is given and the database of IORs set up so that the firewall can pass on requests to a couple of objects on the internal network.

1.3 Grid Example

The example of a simple grid interface is considered in order to illustrate the operation of the Wonderwall. This tutorial example assumes that the client, Wonderwall, and server all run on the

same host. In a realistic situation these processes would run on three separate hosts. Details on setting up a real installation are given in your install guide.

The grid interface on which the example is based is as follows:

```
// IDL
// Definition of a 2-D grid.

interface grid {
   // height of the grid
   readonly attribute short height;

   // width of the grid
   readonly attribute short width;

   // set the element [n,m] of the grid, to value:
   void set(in short n, in short m, in long value);

   // return element [n,m] of the grid:
   long get(in short n, in short m);
};
```

This defines the interface for a two-dimensional grid of long integers whose size is given by the `height` and `width` attributes. Two operations `set()` and `get()` can be invoked to respectively modify or read a single element of the grid.

The details of implementing a grid object need not be considered here. It is assumed that there is a server called `grid` which implements at least one grid object. Likewise it is assumed there is a client that makes use of the object. Both server and client use the IIOP protocol.[1] In this example the grid client represents an external, possibly hostile, process which wishes to use objects in the server. The server itself is to be protected by the Wonderwall.

1.4 OrbixWeb Client

An OrbixWeb client can use the Orbix `_bind()` mechanism to connect to objects behind the Wonderwall. It is assumed that the client has been compiled with the relevant options to ensure that it uses the IIOP protocol.[2] For the client programmer, all that is needed to connect to a grid object behind the Wonderwall is something like the following:

```
// Java
package gridtest;

import IE.Iona.Orbix2._CORBA;
```

1. Orbix is shipped with a demo `grid_iiop` which provides an example implementation of the grid using the IIOP protocol.
2. This is the default behaviour in OrbixWeb. See the *OrbixWeb Programming Guide* for more details.

```
import IE.Iona.Orbix2.CORBA.SystemException;

public class Client {
   ...
   public static void main(String args[]) {
       _gridRef gRef = null;

       try {
          gRef = grid._bind("grid1:GridSrv",
                            "wwallHost");
       }
       catch (SystemException se) {
          System.out.println(se.toString());
       }
       ...
   }
};
```

The `_bind()` call will contact the Wonderwall proxy and establish an IIOP connection. The first argument to `_bind()` is of the form "*`marker`*`:`*`server`*", where *`marker`* is a string identifying the object within a particular *`server`*.[3] The second argument "*`host`*" specifies the host where the Wonderwall proxy is running. The advantage of using `_bind()` is that the client does not need to have an Interoperable Object Reference (IOR) for `grid1` before making the connection.

If you are using a non-OrbixWeb client or if, for some reason, you do not wish to use `_bind()`, then you will have to understand the concepts underlying IORs and the process of *proxification* of IORs. See section 3.2 for details.

1.5 The Configuration File

The first stage in setting up the firewall is to create the configuration file `iiopproxy.cf`. An example configuration file for the grid example is the following:

```
#########################################################
# A sample Wonderwall configuration file.
port 1570
orbixd-iiop-port 1571 # Use the Orbix IIOP port.
domain your.domain.com
log requests replies
http-port 0
http-files /
#########################################################
# Database of Objects.
object grid_1 bind("grid1:GridSrv","gridHost") interface grid
object grid_2 bind("grid2:GridSrv","gridHost") interface grid
```

3. The identifiers *`marker`* and *`server`* are Orbix specific concepts.

```
allow-unlisted-objects on

#######################################################
# On to the access control list!
# Disallow any IOP Service Contexts, at least until we need
# them... who knows what could be put in here?
#
deny servicecontexts *
# Allow general access to grid_1,
# except for the "set" operation.
#
allow object grid_1 op _get_height
allow object grid_1 op _get_width
allow object grid_1 op get
# Allow access to grid_2 from our link to a semi-trusted
# network, but log any "set" operations.
#
allow object grid_2 ipaddr 10.23.67.1 op _get_height
allow object grid_2 ipaddr 10.23.67.1 op _get_width
allow object grid_2 ipaddr 10.23.67.1 op get
allow object grid_2 ipaddr 10.23.67.1 op set log
# File ends here -- if the message hasn't matched a rule
# until now, it'll be denied automatically.
#######################################################
```

The file `iiopproxy.cf` is read in by the firewall server `iiopproxy` when it starts up. Subsequent changes made to `iiopproxy.cf` will affect new clients (but any existing client sessions will not be affected by the changes). This file is at the heart of Wonderwall's operation. A brief explanation for each line of the above example is given here (full explanations of these fields are given in Appendix A).

1.5.1 Basic Configuration and Ports

Lines beginning with a '#' character are comments, and trailing comments on a line are also allowed.

The first specification, of the form `port 1570`, specifies that Wonderwall listens for requests on TCP port 1570.

The next port specified is `orbixd-iiop-port 1571`. This refers to the port where the Orbix daemon listens for IIOP messages on the internal network. It is essential to specify this port number if you are going to be using the Orbix daemon. The Wonderwall needs to know which port the Orbix daemon is listening on, in order to interact with it.

The entry `domain your.domain.com` gives the DNS domain name of the host where Wonderwall is running. The next entry `log requests replies` tells Wonderwall to log all IIOP request and reply messages.

The entries `http-port` and `http-files` are used to configure the HTTP server capability of the Wonderwall. They are discussed in detail in section 1.7.

1.5.2 Object Specifiers

The next section of the configuration lists all of the objects which might be made available through Wonderwall. The Wonderwall proxy uses this list to construct an internal table of known objects. The general form of these entries is:

```
object tag [wild wildcardflags] object-specifier
```

This entry declares a *`tag`* which is used to refer to the specified object throughout the configuration file. The optional `wild` field is used to refer to *categories* of objects, rather than a single object, and is discussed in Appendix B.2. The *`object-specifier`* can be specified in a number of ways (see Appendix A).

At present the Wonderwall supports four different forms of object-specifier:

- An object-specifier beginning with the keyword `"bind"` is used to specify the object using a pseudo-bind syntax (which closely resembles the syntax of `_bind()` as used by a regular OrbixWeb client).
- An object-specifier that begins with the characters `"IOR:"` introduces an IOR coded as a standard CORBA stringified object reference.
- An object-specifier that begins with the characters `"RXR:"` introduces an IOR encoded using the readable-hex-representation (explained in detail in section 2.3).
- An object-specifier that begins with a `"/"` is assumed to be the absolute pathname of a file where the IOR is stored (either in `"IOR:"` or `"RXR:"` format).

All of these forms of object-specifier are explained in detail in section 2.3 and section B.2. The simplest specifier to use is the bind format. This format requires that the Wonderwall is able to contact an Orbix or OrbixWeb daemon in order to locate the server. If you are using a non-Orbix server, you will have to read Chapter 3 on interoperability and use one of the three alternative object-specifiers.

Assuming you are using an Orbix or OrbixWeb server, it is possible to use the `bind` format as given in the above configuration file, for example:

```
object grid_1 bind("grid1:GridSrv","gridHost") interface grid
```

The pseudo `bind` function has a similar format to `_bind` in the OrbixWeb client. This example specifies an object with marker `grid1`, held by the server named `GridSrv`, found on host `gridHost`. The trailing fields `interface grid` (which must be present) specify that the object is of type `grid`.[4]

The last entry of this section, `allow-unlisted-objects on`, gives you a powerful mechanism for extending the list of known objects. When it is set to `on` (the default setting) then any time a

4. It is assumed that the server `GridSrv` has been registered with the Orbix daemon in the usual way.

client attempts to access an unlisted object, the Wonderwall will automatically update and add the object reference to its table of known objects. This considerably relieves the burden of administration required for a minimal configuration of the Wonderwall. Note that, just because an object gets automatically listed in this way, does not mean that the client has permission to connect to the object. That is determined by the Access Control List.

In some high security networks, the administrator may prefer to switch this option to `off`.

1.5.3 Access Control List

The last section of the configuration file is the Access Control List. This consists of a list of rules which begin with either one of the keywords `allow` or `deny`. Whenever a request arrives at the Wonderwall server, these rules are checked in sequence until a rule is found which definitely denies access or definitely allows access to the target object.

The first rule given here is `deny servicecontexts *`. A service context is a mechanism which allows extra information to be added to an IIOP request (or reply) for use by the CORBA services. In keeping with the firewall philosophy of "anything not expressly permitted is denied" it is considered safer to forbid all requests with a service context attached. Note that the core specification of CORBA does not make use of service contexts.

The next few rules have a form similar to the following:

```
allow object grid_1 op _get_height
```

This states that the request is allowed if it is to be invoked on object `grid_1` *and* the operation name is `_get_height`. Note that the operation name `_get_height` derives from the attribute name `height`. For every attribute, such as `height`, there are associated with it two operation identifiers `_get_height` and `_set_height`. If the attribute is declared `readonly`, there will be just one operation `_get_height`.

The rules applying to the object `grid_2` are specified in a slightly different way, for example:

```
allow object grid_2 ipaddr 10.23.67.1 op _get_height
```

This stipulates that if the request is to invoke on object `grid_2` *and* the IP address of the invoking host is 10.23.67.1 *and* the operation is `_get_height`, then the request is allowed.

The last line of the Access Control List is the following:

```
allow object grid_2 ipaddr 10.23.67.1 op set log
```

This specifies that the operation `set` is allowed on object `grid_2` when the host has an IP address 10.23.67.1. In addition, the final keyword `log` specifies that all such requests should be logged (in this example the logging is superfluous since all incoming and outgoing requests and replies will be logged anyway).

It is important to understand how Wonderwall parses the Access Control List. It starts at the beginning of the list, reading each rule in sequence, until it finds a rule which unambiguously

allows or denies a request. Wonderwall then stops and does not read any more rules. This approach makes it easy to predict how Wonderwall will interpret the Access Control List.

A non-intuitive side effect of this algorithm is that it is permissible to have contradictory rules. The resolution of any conflict is simple: The first rule takes precedence.

1.6 Factory Objects

One of the interesting features of CORBA is that it allows you to pass back and forth object references inside Request or Reply messages, where they might appear either as parameters or return values. This provides a powerful mechanism for clients to obtain references to new objects. The term *Factory Interface* is applied to any interface which can create a new object and return a reference to this object. Individual instances of a Factory Interface are known as *Factory Objects*.

Consider the following example of a Factory Interface:

```
// IDL
typedef string MarkerString;

interface GridFactory {
   // Make an object of type ‘grid’
   // and return the object’s marker.
   MarkerString makeGrid();
};
```

This particular interface is an Orbix specific example of a Factory (because it returns an Orbix marker instead of an Interoperable Object Reference). The marker gives an OrbixWeb client enough information to find the object using the pseudo `_bind()` mechanism.

The existence of Factory Objects poses special problems for the Wonderwall administrator. Object level security is based on the idea that a finite number of objects are listed and it is known whether they may safely be accessed from outside. A Wonderwall administrator must consider, not only whether the Factory Object is safe, but also whether objects created by the Factory can be considered safe. This also applies to the related idea of *Finder Objects*, which do not actually create new objects, but could return object references not listed in the Wonderwall configuration.

Notwithstanding, there are compelling reasons for making use of both Factory and Finder objects. For example, if you wished to access a database through the firewall, and this database represents its records in the form of CORBA objects: The number of objects is likely to be huge and it would not be practical to list them in the Wonderwall configuration file. A finder object is a practical way of providing access to the records.

Assume that there is a given Factory Object, such as GridFactory above, which needs to be used through the firewall. This implies that Wonderwall must provide a means of accessing both the Factory Object and objects created by that Factory.

Wonderwall provides the following form of entry in the configuration file for specifying Factories[5]:

```
server tag object-specifier
```

The server keyword is used to define a *tag* which refers to all of the objects on a particular server. The object given by the *object-specifier* refers to an object which can be used to make the initial connection to the server. Usually this will be the factory object. For example, the `GridFactory` object can be listed as follows:

```
server gridFactory \
   bind(":FactorySrv","gridHost") interface GridFactory
```

The tag `gridFactory` can now be used to refer to all objects on the `FactorySrv` server, irrespective of marker, or interface name. Therefore a line such as the following in the Access Control List can be used to give away access to all objects on that server:

```
allow object gridFactory
```

Note that because the object type of the tag `gridFactory` is wildcarded, it is legal to specify a rule such as the following:

```
allow object gridFactory operation _get_height
```

The operation `_get_height` does not appear in the interface `GridFactory`, only in interface `grid`. The appearance of `interface GridFactory`, in the above object specifier, is just a placeholder. Any operation at all, from any interface, may be specified in a rule with a server tag.

Note that the Wonderwall's support for factories is dependent on using Orbix or OrbixWeb objects, as it needs to understand the object key format. For more details on the object key format, see "Orbix/OrbixWeb Object Key Format" on page 17.

1.7 HTTP Server

The Wonderwall proxy normally listens for all IIOP messages on a single dedicated port. It monitors this port and redistributes IIOP Request messages to servers behind the firewall.

However, an IIOP port is not yet a standard feature of most firewalls. Until this port becomes established in firewalls, it will be necessary to use *HTTP tunnelling* to smuggle IIOP messages through the HTTP port.

This approach requires the complicity of the HTTP server, which is required to recognise that some HTTP messages may contain data which is meant to be interpreted as an IIOP message. For this reason, the Wonderwall proxy has had the full functionality of a HTTP server added to it.

---

5. This is equivalent to the following construction:
`object tag wild marker ifmarker object-specifier`
The `server` keyword is provided as a convenience for defining Factory objects (see Appendix B.2).

This functionality of the Wonderwall is illustrated in Figure 1.1. The process `iiopproxy` is capable of listening on two ports: one of these is a dedicated IIOP port and the other is a HTTP port (usually port 80).

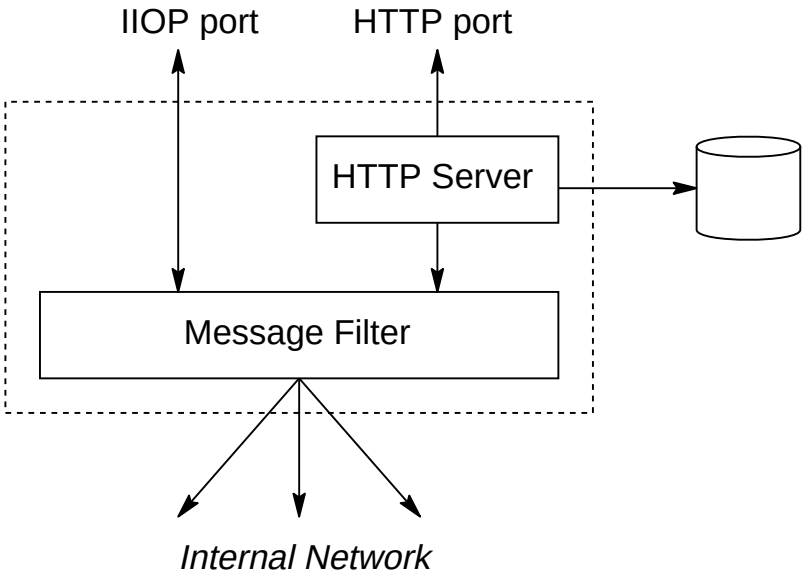

Figure 1.1: *Internal architecture of the Wonderwall proxy server.*

When `iiopproxy` listens on the HTTP port, it functions as a full-function HTTP server. Any normal HTTP requests that arrive will cause it to search a designated directory, and return a copy of the requested Web page (if it can be found). However, this HTTP server also has the intelligence to recognise when a tunnelled IIOP message arrives via HTTP. In that case, it extracts the IIOP message and passes it on to the IIOP gateway.

It does not matter to the gateway whether an IIOP message arrives through the dedicated port or by way of HTTP. The message is still subject to the same filtering mechanism regulated by the configuration file, as described in section 1.5.

The configuration of the HTTP server only requires two parameters to be set in the configuration file:

```
http-port port
http-files directory
```

The `http-port` is used to set the *`port`* where the `iiopproxy` listens for HTTP requests. The keyword `http-files` is used to specify the *`directory`* where files can be retrieved to service ordinary HTTP requests.

If you specify `http-port 0`, then HTTP functionality is not enabled and `iiopproxy` will listen only on the dedicated IIOP port for ordinary IIOP messages.

1.8 Logging Output

The log from the Wonderwall server `iiopproxy` is sent by default to the standard output. Usually the user will redirect this output to a log file. It is possible to specify what goes into the log file by editing the configuration file and Wonderwall gives you quite an amount of flexibility in this respect. In this example configuration, the line `log requests replies` ensures that all IIOP requests and replies passing in or out through Wonderwall will be logged. The log records essentially all of the information available in the request or reply headers.

The logged output, when an IIOP message is forwarded, generally takes the following format:

```
forwarded: <client> -> <servername>: [Message v1.x: <size>
   bytes: Request <request id>, op [ObjectKey "<object
   key>"]::[<operation>] from "<principal>", respond?
   <response expected>]

forwarded: <client> <- <servername>: [Message v1.x: <size>
   bytes: Reply <request id>, reply status <reply status>]
```

Consider the sample log output generated by a client invoking on the grid via Wonderwall:

```
IIOP connection opened: [ultra:64023]
starting server for activated object "grid"
forwarded: [ultra:64023] -> [grid]: [Message v1.0, 82 bytes:
   Request 0, op [ObjectKey
   "RXR::%5cultra.dublin.iona.ie:grid:0::IR:grid_"]
   ::[_get_height] from "RXR:jmason", respond? y]
forwarded: [ultra:64023] <- [grid]: [Message v1.0, 14 bytes:
   Reply 0, reply status NO_EXCEPTION]
forwarded: [ultra:64023] -> [grid]: [Message v1.0, 82 bytes:
   Request 1, op [ObjectKey
   "RXR::%5cultra.dublin.iona.ie:grid:0::IR:grid_"]
   ::[_get_width] from "RXR:jmason", respond? y]
forwarded: [ultra:64023] <- [grid]:[Message v1.0, 14 bytes:
   Reply 1, reply status NO_EXCEPTION]
forwarded: [ultra:64023] -> [grid]:[Message v1.0, 84 bytes:
   Request 2, op [ObjectKey
   "RXR::%5cultra.dublin.iona.ie:grid:0::IR:grid_"]::[set]
   from "RXR:jmason", respond? y]
forwarded: [ultra:64023] <- [grid]:[Message v1.0, 12 bytes:
   Reply 2, reply status NO_EXCEPTION]
forwarded: [ultra:64023] -> [grid]:[Message v1.0, 78 bytes:
   Request 3, op [ObjectKey
   "RXR::%5cultra.dublin.iona.ie:grid:0::IR:grid_"]::[get]
```

```
   from "RXR:jmason", respond? y]
forwarded: [ultra:64023] <- [grid]:[Message v1.0, 16 bytes:
   Reply 3, reply status NO_EXCEPTION]
   IIOP connection closed: [ultra:64023]
```

The logging facility also allows the full request and reply bodies to be logged. The rules for the Access Control List also let you dictate that requests or replies be logged only in specific circumstances. For full details of the logging options see Appendix A.

Chapter 2
IORs and IIOP

Wonderwall provides firewall security for applications that communicate using the IIOP protocol. An elementary understanding of the IIOP protocol is therefore indispensable for the proper use of Wonderwall and will help you to appreciate the issues which affect security.

In terms of security implications for the client side, IIOP is not another Java. It does not download executables onto the client machine and it is quite benign. It provides a protocol which enables a client to contact a remote server and call remote functions on this server. Data may pass between client and server, in the form of parameters, however nothing is sent by the server to be executed on the client side.

The server, on the other hand, is in need of some protection because it allows clients to remotely invoke operations which run on the server's host. The Wonderwall provides protection for servers which might expose security loopholes and it also restricts access to certain operations which the server does not wish to make available to remote clients.

The CORBA interoperability specification defines both the mechanism by which clients establish communications with a server and the details of message formats and data coding. The key concept which CORBA uses to enable clients to connect to servers is the Interoperable Object Reference (IOR) as discussed in the following section. The message formats and data coding are discussed in section 2.4 on page 20.

2.1 IOR Format

In order to identify objects in a distributed object system CORBA uses the concept of an *object reference*. Once an application has an object reference, it has all the information it needs to connect to the object and make remote invocations on the object's methods.

The notion of an object reference is an abstract one. To the application CORBA programmer it can be represented simply as a C++ pointer. Individual ORB vendors can have their own proprietary representation of an object reference.

However, as part of the infrastructure for an interoperability protocol, CORBA also specifies a universal format for object references known as the Interoperable Object Reference (IOR). This enables the information about an object reference to be either stored or communicated directly to clients in a form which is universally understood. All ORB vendors are required to support this form of object reference.

The information encoded in an IOR (as used in conjunction with the TCP/IP protocol) consists of the following pieces of information:

- The type of the object.
- The host where the object may be found.
- The port number of the server for that object.
- An object key (a string of bytes identifying the object).

The type of the object is equivalent to the name of the IDL interface which is used to define the object. For example in section 2.1 on page 11 an IDL interface was defined for objects of type `grid`. The host and port together give us the connection information required to contact the server. Finally the object key is used by the server itself to locate the object.

Figure 2.1 outlines the format of an IOR in greater detail and is intended to give a schematic view of the information held in an IOR. The upper part of Figure 2.1 shows the overall format of the IOR. It begins with the string `type_id` which gives the type of the object, equivalent to the name of the interface defining the object[1]. There follows a sequence of profiles, preceded by a `profile_count`. In the Figure 2.1 are shown two profiles preceded by a `profile_count` of 2. A profile contains essentially all the information which is needed to find an object. The facility to specify more than one profile in an IOR is a useful feature which will allow future extensions to the use of IORs. For example, an IOR can specify a number of possible locations for an object. If a client does not succeed in connecting to the location specified in first profile, the client can try the next profile in the sequence instead. Wonderwall supports the use of IORs with multiple profiles.

In the lower part of Figure 2.1 are shown the details contained in a single profile. Note that the connection information stored in a profile is specific to a particular underlying protocol. For this reason the first field is a `protocol_tag` and, in this example, the tag value is zero to indicate a TCP/IP transport protocol. This is followed by the `Version` fields which consists of a `major` and a `minor` version number. The next two fields provide the `host` and IP `port` needed to establish communication with the remote server. The `object_key` is a field which will be used by the

1. To be precise this field holds the `RepositoryId` for the type of object.

remote server to locate the object being accessed. There can also be additional fields at the end but these are currently not used and are reserved for future expansions to the protocol.

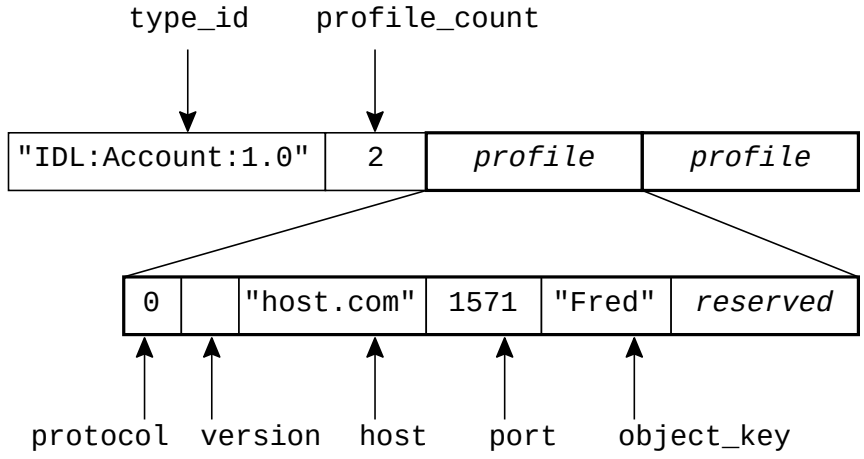

Figure 2.1: *The format of an Interoperable Object Reference and profile.*

It may seem surprising that the format of an `object_key` is not specified by CORBA. However this fact does not affect interoperability nor make the IOR any less portable. The `object_key` is used only by the *server* to identify the object referred to. The client needs to have a copy of the `object_key` but does not need to interpret it in any way. As far as the client is concerned the key is just an opaque code (in fact a sequence of bytes) which it passes to a server in order to identify an object. The server, which originally assigned the `object_key`, then makes active use of the key to find the object.

The outline of an IOR given here is only intended to be schematic although it does include all of the essential information which is supplied in a typical IOR. The formal specification of an IOR is given in terms of IDL data types. For the complete specification of an IOR refer to the CORBA interoperability specification.

2.2 Orbix/OrbixWeb Object Key Format

Orbix and OrbixWeb object keys in IORs have the same format as Orbix-protocol object references. They take the following the form:

```
:\host:serverName:marker:IR_host:IR_Server:interfaceMarker
```

These fields are as follows:

`host`	The host name of the target object.
`serverName`	The name of the target object's server as registered in the Implementation Repository and also as specified to `CORBA::BOA::impl_is_ready()`, `CORBA::BOA::object_is_ready()` or set by `setServerName()`. For a local object in a server, this will be that server's name (if that server's name is known), otherwise it will be the process' identifier. Note that the server name will be known if the server is launched by Orbix; or if it is launched manually and the server name is passed to `impl_is_ready()` or if the server name has been set by `CORBA::ORB::setServerName()`.
`marker`	The object's marker name. This can be chosen by the application, or it will be a string of digits chosen by Orbix.
`IR_host`	The name of a host running an Interface Repository that stores the target object's IDL definition. Normally, this is blank.
`IR_server`	The string "IR" or "IFR", depending on the version of Orbix or OrbixWeb in use.
`interfaceMarker`	The target object's interface. If called on a proxy, this may not be the object's true (most derived) interface—it may be a base interface.

2.3 Representations of an IOR

A portable representation of an IOR is a basic requirement. Generally speaking, an IOR is created by the server which supports the corresponding object. The IOR is publicised in some way so that it is available to prospective client processes. Once a client obtains a copy of the IOR it will then be able to connect to the object.

For convenience in publishing an IOR it must be possible to convert it to a string format which is not subject to any conversions when communicated from place to place. For this reason, CORBA specifies a standard string format for IORs. The following is an example of such a string:

```
IOR:000000000000000d49444c3a677269643a312e300000000000000001
000000000000004c000100000000015756c7472612e6475626c696e2e69
6f6e612e6965000009630000002 83a5c756c7472612e6475626c696e2e69
6f6e612e69653a677269643a303a3a49523a67726964003a
```

It consists of the characters `IOR:` followed by a lot of hexadecimal numbers. Every byte of the original IOR is translated into a two-digit hexadecimal number. This representation has the advantage that it is simple and resistant to corruption.

Unfortunately, the standard string format has the disadvantage that it is difficult to interpret the content of the IOR. A typical IOR is not really as opaque as this. To make IORs more comprehensible, the Wonderwall can use its own format known as the Readable Hex Representation RXR. The RXR format is a hybrid format which mixes plain ASCII characters with hexadecimal numbers. As an example, consider the RXR representation of the object reference just given:

```
RXR:_______%0dIDL:grid:1.0_______%01_______L_%01_____%15ultr
a.dublin.iona.ie__%09c___(:%5cultra.dublin.iona.ie:grid:0::I
R:grid_:
```

The RXR format is provided in order to provide readable logging messages and a convenient way to specify strings of octets. It incorporates concepts from the URL encoding for HTTP (RFC 1738). RXR format strings are written as follows:

```
RXR:<version><string>
```

The 4-character upper-case string `RXR:` must be present at the start. The `<version>` specifier is optional and can be omitted. If it is present, it takes the form `%vX` where the `X` character encodes a format identification character ranging from 0 to 9, a to z, and A to Z. If this version specifier is not present, version 0 is assumed. This document describes RXR format version 0.

Each octet of the octet string is stored, in order, in the `<string>` specifier. Octets must be encoded if they have no printable representation in the US-ASCII coded character set, if the use of the corresponding character is *unsafe*, or if the corresponding character is reserved for some other interpretation within this representation format.

The octets which must be encoded are as follows (the values are specified in hexadecimal, and ranges are inclusive): any octet from 00 to 20, octets 22, 23, 25, 27 and 3B, octets between 5B and 60, and octets from 7B to FF. Here is an annotated list of ostensibly-printable octets deemed unsafe:

OCTET VALUE	SPECIAL USE
`%`	used to signify octet-encoding
`_`	used to signify null-encoding
`# ;`	may be used as a comment
`' " (space)`	may be used as a string delimiter
`` ` [ ] | { } ~ \ ^ ``	may be corrupted by gateways or shells

Table 2.1: *Characters considered special by the RXR format.*

The encoding methods are as follows. For non-NUL (hex 00) octets, a '%' (percent) character is stored in the string, followed by the high-order nibble of the octet encoded in hexadecimal, followed by the low-order nibble encoded in the same way. The character '_'(underscore) is used to encode a NUL (hex 00) character. Here is an example of an RXR encoded IOR from OrbixNames:

```
RXR:_______%20IDL:CosNaming/NamingContext:1.0
____%01_______W_%01_____%10192.122.221.136_a%eb_____7:%5cult
ra.dublin.iona.ie:NS:::IR:CosNaming%5fNamingContext_
```

2.4 Internet Inter-ORB Protocol (IIOP)

The IIOP protocol is just a special case of the General Inter-ORB Protocol (GIOP). The GIOP specification provides a general framework for protocols to be built on top of specific transport layers. The IIOP protocol is the specialisation of GIOP which is built on top of TCP/IP.

Many aspects of IIOP discussed in this section apply equally well to any GIOP protocol but no attempt will be made to distinguish the different elements of the specification here.

In general the IIOP specification has three main elements:

- Transport management requirements.
- Definition of CDR coding.
- IIOP message formats.

The transport management requirements give a high level view of the semantics of setting up and ending connections. The roles of client and server and the respective functions of each are outlined at this level. The protocol described is connection oriented with well-defined roles for client and server.

The second element of the specification is the Common Data Representation (CDR). This transfer syntax specifies a coding for all IDL types: including basic types, structured types, object references (in the form of IORs) and pseudo-object types such as `TypeCodes`. The CDR coding translates IDL types into a series of bytes to make up an octet stream (the CORBA name for a raw memory buffer). A feature of CDR is its ability to deal with the different kinds of byte ordering required by different hardware types: both big-endian and little-endian byte ordering is supported. The convention adopted is that the *sender* of a message sends data using its native byte ordering (and sets a flag in the message header to indicate the ordering used). The receiver of a message is obliged to detect the byte ordering used and carry out any conversion, if it is required. The advantage of this convention is that when both sender and receiver use the same byte ordering, no conversion is required resulting in considerable gain in efficiency.

2.5 IIOP Message Formats

The IIOP protocol defines seven types of message format. The messages allow clients to pass invocations to servers and receive replies which can be either normal or indicate some error status. Some additional messages are available to help manage the connection.

The two most important message formats are the Request and Reply message formats. An operation which has been declared in the IDL interface for an object will be invoked by a client using a Request message. The client will usually wait for a Reply message from the server (unless the operation has been declared to be `oneway`) which will normally contain a return value, or possibly an error condition.

The other five messages are called CancelRequest, LocateRequest, LocateReply, CloseConnection and MessageError. They are all concerned with managing some aspect of the connection and their role is discussed in more detail below.

The IIOP messages fit into one of three formats:

- A GIOP message header only.
- A GIOP message header followed by a message header specific to the message type.
- A GIOP message header followed by a specific message header and message body.

Note that in all cases a message will begin with a GIOP header. The GIOP message header has the form illustrated in Figure 2.2. The fields in the header can be described as follows:

- The four characters “GIOP” which serve to identify the protocol.
- The GIOP version number (major and minor) used to create the message.
- A flag byte which is currently only used to indicate the byte ordering.
- An integer used to indicate the message type.
- The message size (excluding the GIOP header itself).

This summarises all of the information which is sent in the GIOP header. For a formal specification of the exact header format you can consult the CORBA specification.

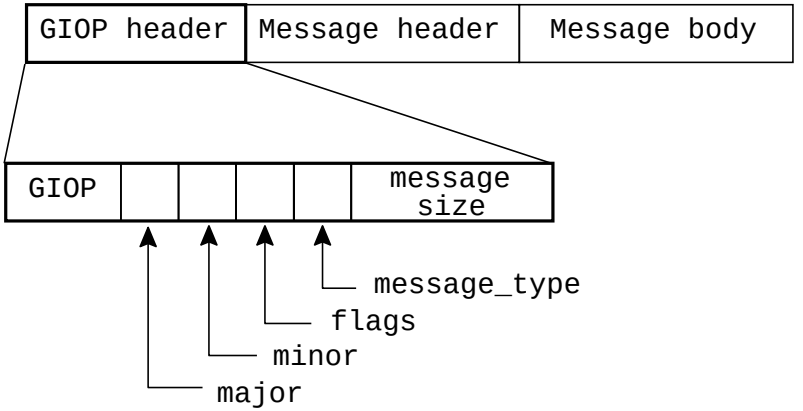

Figure 2.2: *The format of a GIOP message and message header.*

The purpose and usage of the different IIOP message formats is explained now. This guide is only intended to explain them sufficiently to use Wonderwall effectively. For complete details of the message formats you should consult the CORBA specification.

Request Message

A Request message allows a client application to invoke an operation on a remote server. The message contains all the information which is needed for the invocation including the identity of the object, the operation name and any parameters associated with the operation. Note that a Request message is designed specifically to invoke operations which have been declared in an IDL interface. The message format is thus designed to support all of the syntax which can appear in an IDL operation definition.

The message consists of a Request header followed by a Request body. An outline of the Request header is shown in Figure 2.3. It consists of the following fields:

- The service contexts allow service specific context information to be passed along with a Request. These are intended for use in conjunction with the CORBA services

to carry extra information along with the Request.[2] However, the service contexts are not needed in the core specification of CORBA.

- The `request_id` is used to uniquely identify a Request emanating from a client so that the client can later match a received Reply with its corresponding Request (the corresponding Reply will be tagged with the same `request_id`).
- The `response_expected` flag is used to indicate whether or not the Request is `oneway` or not. A normal Request has `response_expected` set equal to TRUE.
- The next field is an array of three bytes reserved for future use.
- The `object_key` is used at the server end to identify the object which is being invoked.
- The `operation` field is simply a string giving the name of the operation being invoked.
- The `requesting_principal` identifies the user making the request. In other words it is simply the user name of the person running the client.

This is all of the information available in the Request header. The Request header is of particular importance to the operation of Wonderwall. The Wonderwall carries out its filtering based upon the contents of the Request header. For example, if you look ahead to Appendix A you can verify that all of the rules for filtering requests are based on the contents of this header.

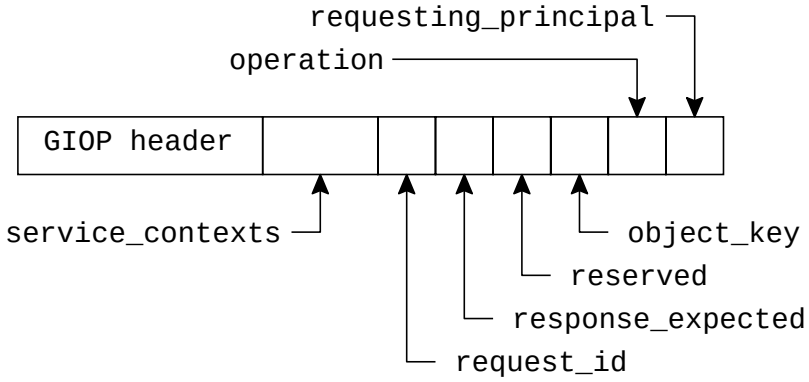

Figure 2.3: *The format of a Request message header*.

2. This field is used by the Transaction Service, for example.

The Request also has an associated Request body. The body of the Request consists essentially of a list of the operation parameters followed by any `context` strings for the operation.[3] It is possible for the body of the Request to be empty—for example if the Request was made for an operation which took no parameters and omitted a `context` clause.

Since the filtering done by Wonderwall is based entirely on the Request header there is no need for it to parse, or alter in any way, the Request body. This fact simplifies the filtering process significantly and ensures that the filtering and forwarding of request messages can be done simply and efficiently.

Reply Message

A Reply message is normally sent by a server in response to a client Request message. The Reply message consists of a GIOP header followed by a Reply header and a Reply body. The usual intent of a Reply message is to pass back a return value for an operation and to indicate the completion status for the operation.

The Reply header does not pass as much information as in a Request header and it consists basically of the following three fields:

- The `service_context` which is similar to the service context described in connection with a Request message.
- The `request_id` which is used to match this Reply to the client Request which gave rise to it. In other words, all Replies are paired off with their corresponding Request and the `request_id` is a unique (per client) identifier used to match Request and Reply.
- The `reply_status` is used to indicate whether this is a normal Reply or if some error condition occurred in the server.

In fact the `reply_status` is used to toggle between a number of different Reply types so that a Reply message is almost like four messages rolled into one. The possible values for `reply_status` are the following:

`NO_EXCEPTION`	This is the normal Reply type. The body contains any return, `out` or `inout` parameters which have been declared in the IDL for the operation.

3. These `context` strings have nothing to do with service contexts. They are effectively middleware environment parameters and they will only be passed if a `context` clause appears at the end of an operation definition in IDL.

`USER_EXCEPTION`	This status indicates that a user exception has been raised in the server. The body of this Reply type contains the details of the user exception.
`SYSTEM_EXCEPTION`	This status indicates that a system exception occurred. The body of the Reply will indicate the kind of system exception raised.
`LOCATION_FORWARD`	This is a special kind of Reply which a server can use to let a client know that it does not hold the object to which the Request refers. The body of a `LOCATION_FORWARD` reply contains a new IOR for the object. The client can use the new IOR to resend the Request to the new location (this is done transparently as part of the IIOP protocol).

The LOCATION_FORWARD Reply is routinely used by the Orbix daemon to dynamically allocate a port to a server process which has been automatically forked by the daemon.

CancelRequest Message

A CancelRequest message is sent by the client to the server to indicate that the client is no longer interested in receiving a Reply to a particular message (however it is not an error if the server should send the Reply anyway).

LocateRequest Message

A LocateRequest message can be sent from client to server to probe for the location of a remote object. It may be advantageous to send this message before sending a large Request on a connection which has just been opened.

LocateReply Message

A LocateReply message is sent from server to client in response to a LocateRequest message. There are three kinds of LocateReply message which the server can send:

- The UNKNOWN_OBJECT response indicates that the server does not hold the object and neither does it know where to find it.
- The OBJECT_HERE response indicates that the server holds the object and communication can proceed as normal.
- The OBJECT_FORWARD response indicates that the server does not hold the object but it does know of a forwarding location for the object. In this case, and in this case only, the LocateReply message has a body. This LocateReply body contains the new IOR.

CloseConnection Message

A CloseConnection Message is sent by the server to the client to tell the client that it intends to close the connection.

MessageError Message

A MessageError message can be sent by either the client or the server. It is used within the IIOP protocol to indicate that the last message received was either corrupted or incorrectly formatted in some way. It consists only of a GIOP header with the message type set to MessageError.

Chapter 3
Interoperability and Details

3.1 Object References

The IIOP protocol was introduced to facilitate interoperability between ORBs supplied by different vendors. For the most part, the use of this protocol is transparent to the user—the main difference you notice is that your ORB is able to talk to many different ORBs, as a result of sharing a common protocol.

However there is one aspect of IIOP of which the user should be aware. The information required to make an initial connection to an ORB must be passed around by some means other than using the IIOP protocol. A connection is established between client and remote server with the help of an Interoperable Object Reference (IOR), which details the location of the object and the information needed to connect to the server.

There are two main formats of IOR: Firstly, an encoded form of IOR which is used to transmit IORs inside an IIOP message. Secondly, a stringified form of IOR which is used to communicate an IOR by any convenient means (see section 2.3).The second form of IOR is typically given to a client to allow it to bootstrap the initial connection to a server. Subsequent IORs may be obtained from the server via the IIOP protocol itself.

Normally the server, which holds the object, creates an IOR for the object and makes this public in some way. There are four main ways in which an IOR can become known to a client:

- The server can create a stringified IOR and write this IOR to a file in a well-known location which is accessible to the client.
- The server can register the IOR with the CORBA Naming Service. The Naming Service, as detailed in the CORBA specification, is basically a database that

associates names with object references. (A client requires just a single bootstrap reference to the Naming Server in order to access all of the IORs stored there).

- An IOR can be sent inside an IIOP message. This applies to any IDL operation which features an interface name as a parameter or return type.
- The Orbix-specific bind mechanism (used for an Orbix or OrbixWeb client talking to an Orbix or OrbixWeb server). The client initially makes contact with the Orbix daemon and the daemon helps the client to determine the IOR of the required object. In this case, the client does not need an IOR to get started—bind provides an alternative bootstrap mechanism.

Of these four methods, the first two provide the most general interoperable way of bootstrapping initial connections.

The operation of Wonderwall is based on the use of two IORs for each object: The *real* IOR and the *proxified* IOR. The real IOR is used by servers operating behind the firewall. Whereas, the proxified version of the IOR is publicised and made generally available outside the firewall. This is explained in the following sections.

3.2 Proxification

A general convention on the internet is that frequently used servers are assigned to a dedicated port. For example, most HTTP servers operate on port 80, most internet mail servers operate on port 25, and so on. Whenever you contact a remote host, you can connect to a particular service by opening a socket on its well known port. The Wonderwall fits this convention by providing a single dedicated port for IIOP messages.

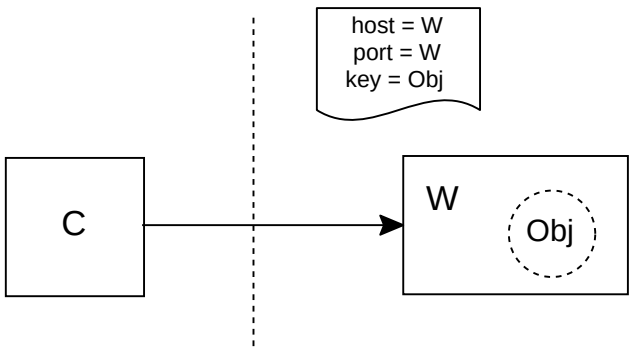

Figure 3.1: *Apparent location of object, in Wonderwall proxy server.*

A port number is embedded directly into every IOR and may have any value. If a large number of CORBA servers are active on a given host then a large number of ports may be in use for IIOP communications. This makes sense, from the perspective of CORBA, since each of these processes is a dedicated server, carrying out a specific sort of task.

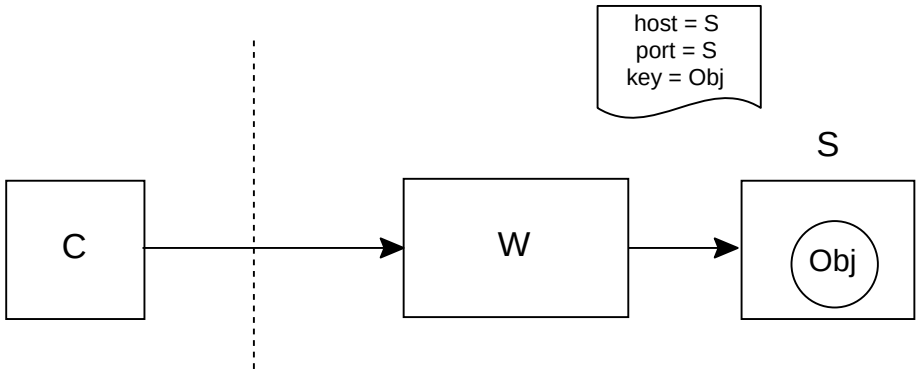

Figure 3.2: *Actual location of object, in server S.*

However the use of multiple IIOP ports poses difficulties from the perspective of a firewall. It is undesirable, from a security point of view, to allow the use of multiple ports on the bastion host. Firewall practice is based on collating all messages of a single protocol type, and passing them through a single port.

The Wonderwall does use a single IIOP port on the bastion host. Any IORs which are used remotely should point at the Wonderwall host and port. The IORs generated by servers on the internal network, however, feature a range of hosts and ports, depending on where they were generated. These IORs are good for use on the internal network since they allow direct IIOP connections to be established behind the firewall. However, we do not want to give them away to users on the internet, because they facilitate direct connections to internal hosts, and a properly constructed firewall would make them unusable across the internet anyway.

It is necessary to modify the real IORs before making them available on the internet, a process referred to here as the *proxification* of the IOR. The principle of proxification is illustrated by figures (Figure 3.1 and Figure 3.2) above.

When a client is communicating with Obj, it has the illusion that the object lives on the Wonderwall server. The IOR which is used to contact this object must have the host and port of server W (the Wonderwall proxy server) embedded, along with the `object_key` for Obj.

In reality, the object lives behind the firewall and is located on server S in the internal network. The real IOR for this object has the host and port of server S embedded in it, along with the `object_key` for Obj. The Wonderwall acts as a proxy for this server, forwarding any messages it receives from the client (subject to filtering by the Access Control List).

The only difference between the real and the public IOR is the value of the embedded host and port. The host and port embedded in the real IOR must be changed. The resulting IOR is a *proxified* IOR.

Proxification can be carried out using the utility `iortool` which comes with Wonderwall. First the real IOR for Obj on server S is written to a file, say `real.ref`, in stringified form[1]. Then the real IOR is proxified with the following command:

```
% iortool -ior -proxy \
   -host wonderwall_host -port wonderwall_port \
   real.ref > proxy.ref
```

This takes the IOR stored in file `real.ref` (which may be either in `IOR` or `RXR` format) and replaces the current host and port embedded in the IOR by *`wonderwall_host`* and *`wonderwall_port`* instead. The result of this proxification process is written to the file `proxy.ref` in `IOR` format. Note that `IOR` format is the portable string representation, as defined by CORBA.

3.3 Non-Orbix Client

If you are using a non-Orbix client to connect to a server via the Wonderwall then you will not have the option of using the bind mechanism. The interoperable approach is based on the use of proxified object references, as described in the previous section. First of all a proxified object reference is obtained as described above and then this proxified reference is publicised using one of the methods discussed in section 3.1. If the client has access to this IOR in string format, then a connection may be established to the server using code similar to the following. The example assumes that the remote object is of type `grid`:

```
// C++
main () {
   ...
   char *proxifiedIOR;
   CORBA::Object_var objVar;
   ...
   // Read the proxified IOR into a string buffer pointed
   // to by ‘proxifiedIOR’.
   ...
   // Convert the string to an object reference.
   objVar = CORBA::Orbix.string_to_object(proxifiedIOR);
   ...
   // Assume that this is the proxified IOR for a
   // ‘grid’ object. Perform a ‘_narrow()’
   grid_var myGridVar = grid::_narrow(objVar);
   ...
```

1. See section 3.2 and consult your ORB programming guide for instruction on how to generate a stringified version of the real IOR.

```
}
```

At the end of these few lines of code, a reference `myGridVar` has been obtained to the desired `grid` object. Note that error handling has been omitted from this example for clarity, but in a real example it is imperative to enclose the calls in a `try/catch` clause.

It is generally more difficult for a client to get a reference to its first object, whether it be via a stringified object reference, as above, or via the Name Server. If this first object is a kind of Finder or Factory object, then subsequent object references can be obtained more easily with its help.

3.4 Non-Orbix Server

For non-Orbix servers, the main restriction is that they are not able to respond to the Orbix bind mechanism. This affects the Wonderwall proxy, because it is the Wonderwall proxy which attempts to make direct connections with servers behind the firewall. The Wonderwall will not be able to specify objects using the bind syntax in its configuration file.

You will not be able to include a line such as:

```
object grid_1 bind("grid1:GridSrv","gridHost") interface grid
```

in your configuration file, when you want to make an object known to the Wonderwall. Instead, you will have to carry out the following steps:

- Obtain a copy of the real IOR for the object in CORBA string format (consult your ORB programming guide for instructions on how to do this). This IOR is needed anyway if you want to generate a Proxified IOR for a non-Orbix client.
- Copy this string to a file in a convenient location, for example you could place it in the file `/etc/iors/grid1.ref`.
- Make this object known to Wonderwall by putting the following line in the configuration file:

```
object grid_1 /etc/iors/grid.ref
```

There are a few alternatives you can use for the *`object-specifier`* field (see Appendix B.2). However, the approach outlined here is probably the most convenient for the interoperable case.

3.5 Connection Establishment

This section explains some of the steps involved in establishing a connection through the Wonderwall. By involving the daemon in the process of connection establishment, it is possible to have servers launched automatically. This means that the server is contacted in a sequence of steps, beginning with an initial connection to the daemon. It is for this reason that the configuration keyword `orbixd-iiop-port` must be set equal to the value of the daemon port—knowledge of this port is needed to facilitate communication with the daemon process.

A Normal IIOP Connection

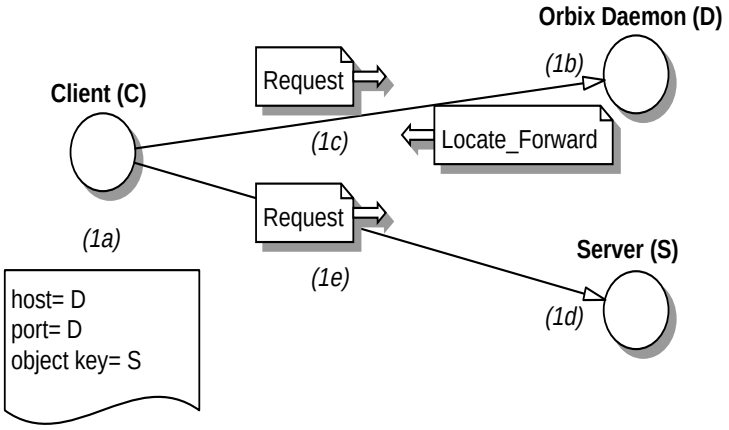

Figure 3.3: *Establishing a normal connection.*

Client C gets the IOR for server object S[2]. For example a Java applet could get this as from an applet tag. This contains the connection details of the activation agent, for example, the Orbix daemon on the server's host.

(1a,b) Client C opens a TCP/IP connection to host D, port D.

(1c) C sends the request message to D. D responds with a LOCATION_FORWARD Reply message containing the real location of S.

(1d) C opens a TCP/IP connection to host S, port S.

(1e) C sends the request message to S.

This example assumes the use of a non-persistent server. Persistent servers do not require the presence of an activation agent (for Orbix, this is the Orbix daemon). In that case the IOR in the first step would contain the connection details of S rather than D and the second and third steps would not be necessary.

2. An OrbixWeb client can use the `bind` mechanism as an alternative.

An IIOP Connection Through The Wonderwall

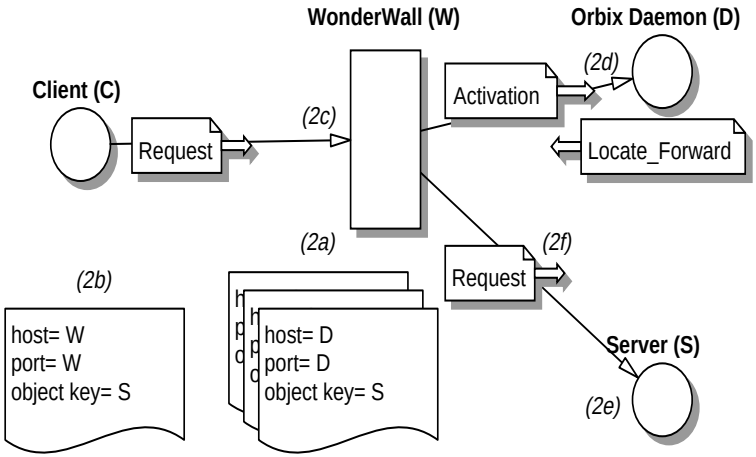

Figure 3.4: *Establishing a connection through Wonderwall.*

Wonderwall W gets the IOR for server object S. This is copied into the configuration file by the system administrator.

(2a,b) Client C gets the proxified IOR for server object S using the same methods as the previous example.

(2c) C opens a TCP/IP connection to host W, port W and sends the request message to W.

(2d) W reads the request, finds the IOR in its configuration that matches the object key used in the request, and opens a connection to host D, port D. An *activation request* is sent to the daemon, causing the server S to start up. D responds to the activation request with the connection details for S.

(2e) W opens a connection to S using the details from D.

(2f) W forwards the request message to S, forwards any replies back to C and so on until the connection closes.

Again, this assumes the use of non-persistent servers. Persistent servers use the connection details of S rather than D, and the fourth step is skipped.

A More Complicated Example - The Use of Object Factories

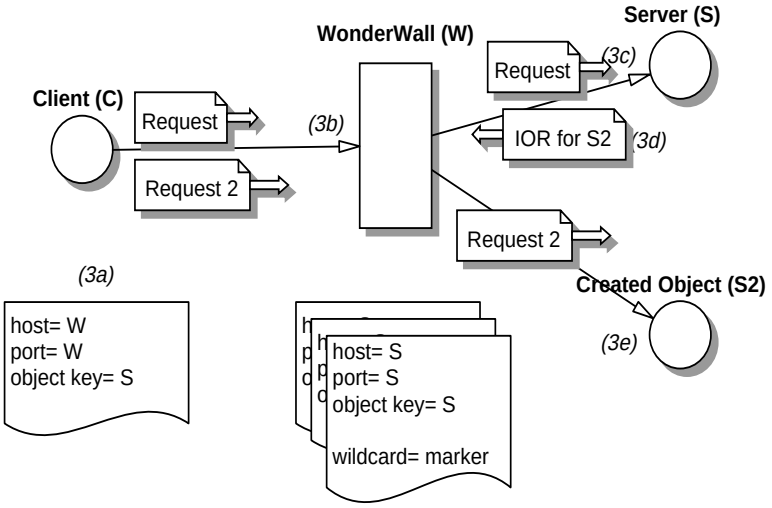

Figure 3.5: *Establishing a connection to a new object through Wonderwall.*

Factory objects are server objects which create objects to handle requests. This diagram also applies to servers which return IORs so that clients can bind to objects.

Wildcard flags are used to indicate that an IOR in the Wonderwall configuration file can be used to match in an approximate manner. Different wildcard flags may be required to support other situations.

To avoid cluttering the diagram the server-activation stage has been omitted.

(3a) Client C gets the proxified IOR for server object as in the previous examples.

(3b) C opens a TCP/IP connection to host W, port W and sends a Request message to W.

(3c) W reads the Request and examines the object key. Since no IOR in its configuration exactly matches the object key, it runs through its list of *wildcard* IORs. It finds the IOR that approximately matches, opens a connection to host S, port S and forwards the Request.

(3d) S creates the object S2 and sends an IOR for it back to W which forwards it on to C.

(3e) C makes a second request and this time it invokes on object S2. W reads the Request and examines the object key. Since S2 uses the same host, port, interface and server name, the wildcard IOR used in (3c) matches this Request as well. A connection is opened to host S, port S (the addressing information in this IOR) and the Request is passed to object S2.

This initial version of the Wonderwall requires that objects created by the factory are represented by proxified IORs. OrbixWeb has an API call to ensure this.

Note that objects S and S2 might not share the same connection details. In that case, a separate wildcard IOR would be necessary listing the known details of S2.

3.6 Factory Objects and IORs

In the general case, Factory objects have a method which is used to create a CORBA object on the server and then return an interoperable object reference to this object. For example, a general version of a `GridFactory` could be defined as follows:

```
// IDL
interface GridFactory {
   // Make an object of type ‘grid’
   // and return an IOR.
   grid makeGrid();
};
```

The single operation `makeGrid()` returns an object reference to an object of type `grid`.

This type of interface introduces a complication for the firewall. Typically, the default behaviour of an operation such as `makeGrid()` is to generate an object reference which points directly at the object on the server itself. But this object reference is not useful on the internet because it points at a server on the internal network, behind the firewall. The operation of the firewall is designed to prevent *direct* access to such internal servers.

The solution is to change the default behaviour of the server, so that any object references it returns refer to the Wonderwall host and port instead. In other words, the Factory object should generate *proxified* object references instead of real object references. You will need to consult the programming manual for your ORB to find out how to do this.

In addition to implementing the `GridFactory` interface, you will need to add to the Wonderwall configuration file, to allow external access to objects generated by the Factory object. A typical approach is the following:

- Generate a real IOR for the `GridFactory` object and store it in a convenient location, for example `/etc/iors/GridFactory_real.ref`.
- Declare a tag in the Wonderwall configuration which refers to *all* of the objects on the same server as `GridFactory`:

  ```
  server GridFactory /etc/iors/GridFactory_real.ref
  ```
- Use this tag in a rule to allow access to all objects in the `GridFactory` server:

  ```
  allow object GridFactory
  ```

This configuration allows any proxified object references, generated by `GridFactory`, to be used by the external network, irrespective of marker or interface type. The use of the `server` keyword,

to generate a tag, allows you to regulate permissions, a server at a time. Currently, this form of wildcarding is supported only for Orbix and OrbixWeb.

3.7 Implications for Developers

From a developer's perspective, the use of Wonderwall has minimal impact. Once the server is ready to be made available to the internet, the IOR and the list of required operations are passed on to the firewall administrator who will assess the security of the server and update the Wonderwall's configuration file. Alternatively, an Orbix or OrbixWeb server allows you to use the `bind` form of an *`object-specifier`* in the configuration file (as explained in Appendix B.2).

On the client side, in general, it is merely necessary to ensure that the client receives a copy of the proxified IOR so that it can establish an initial connection. In the special case of an OrbixWeb client, the procedure is simplified so that an OrbixWeb client will transparently connect to the server via the Wonderwall, if a direct connection is blocked by a firewall.

Callbacks

A firewall unfriendly feature of the IIOP protocol is its use of dynamic port assignment. Firewalls are based around the idea of ports, protocols and services mapping to one another. For example, the SMTP protocol for internet mail runs on port 25, therefore a connection from a client to a server on port 25 is used for sending mail. Since IIOP dynamically creates and assigns ports, this no longer applies so the usual paradigm of opening up a single port to support a particular protocol cannot be assumed.

This is of particular relevance for callbacks. The callback mechanism in the current IIOP specification is unlikely to work successfully unless the client's site is not protected by a firewall. This is because it relies on the server opening a TCP connection to the client using a dynamically assigned port. If the client's site is protected by a firewall, this connection will be blocked. The typical scenario of opening a well-known port does not apply here.

One possibility would be to extend the existing IIOP specification to allow the callback to use the same connection as that used for the initial incoming invocation from the client. Until this is standardised, however, the above model will be restricted to the usual WWW client-driven paradigm. In CORBA terminology this means that invocations can only be issued from the client site to the backend service behind the firewall. Objects resident in a client application/applet behind a firewall cannot act as CORBA servers receiving requests from objects resident in the backend service. Note that although callbacks are not supported, the Requests can of course be two way.

Chapter 4
Transformers

Several internal layers of Orbix separate a simple remote invocation, as seen by the application level programmer, from the final construction and transmission of a message via TCP/IP. Orbix offers the user a chance to customise its behaviour by providing hooks at a number of levels: For example, when an Orbix operation is called on the client side it can be intercepted straight away using a Smart Proxy to customise its behaviour. Orbix provides another hook in the form of Filter objects which can inspect (and modify) a Request object. Finally, after the full contents of a Request have been marshalled into a raw buffer, Orbix provides access to the buffer via the mechanism known as a *Transformer*.

Transformers are useful for a number of purposes. One of the main uses for a Transformer is to allow encryption of the message prior to transmission via the TCP/IP protocol. This provides the user with an added level of security which is desirable in many situations. If your site has a policy of encrypting all messages prior to transmission, then you will find that the support provided by Wonderwall for transformers allows you to insert a firewall with no disruption to the encryption process.

4.1 Transformer Architecture

An outline of a typical Transformer setup, in the absence of a Wonderwall firewall, is shown in Figure 4.1. The shaded blocks shown at the edge of the client process C and server process S represent the transformers at either end of the connection. The heavily shaded line connecting client and server is used to represent the transmission of an encrypted IIOP message. Note that the

Transformers in this figure are not implemented as CORBA objects. We will refer to these Transformers as *integral Transformers* since they are built in to the client or server process.

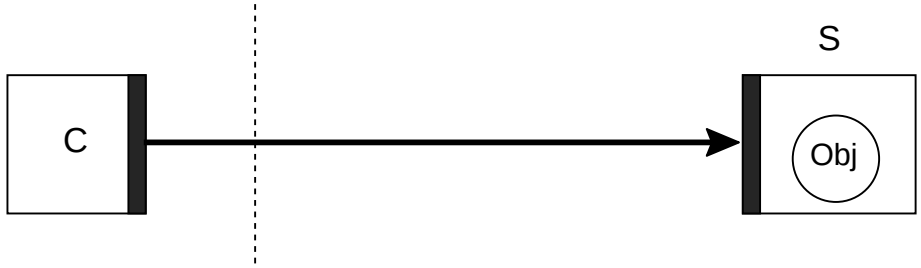

Figure 4.1: *Encrypted link using integral transformer*

Consider the problem of interposing a firewall proxy between this client and server. The firewall cannot deal directly with encrypted messages, nor can it properly monitor and filter messages while they are in encrypted form. In Figure 4.1, decryption is done by the server's integral Transformer. Logically, however, the point at which decryption should occur is just before the packet passes through the Wonderwall.

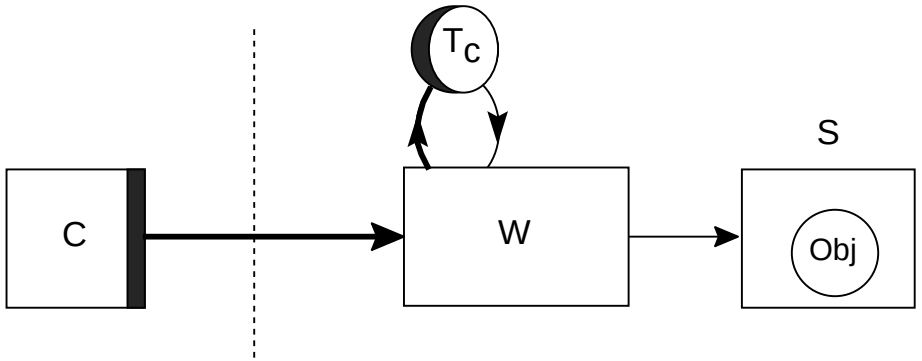

Figure 4.2: *Encrypted link using Wonderwall.*

Wonderwall offers two alternative solutions which enable you to insert the firewall even when encryption is being used. Both of these solutions are based on the definition of *external Transformers* which Wonderwall uses to encrypt and decrypt messages. In contrast to integral Transformers, these Transformers are implemented as CORBA objects.

The first solution is shown schematically in Figure 4.2. This model may be appropriate when the main perceived risk to security is the external network. A single *client Transformer* T_C is implemented to handle encrypted messages arriving from remote clients. When an encrypted Request message arrives from the client, the Wonderwall first sends the message out to the external, client Transformer T_C. The transformer returns a decrypted message, indicated by a thin line, to the Wonderwall. The Wonderwall proxy is then able to monitor and filter this Request message as normal and, if allowed by the Access Control List, the Request will be forwarded to the Server S. When the server responds to the client with a Reply message, the reverse procedure is followed. The unencrypted message is sent from server to Wonderwall, which might log the message, then passed to the client Transformer T_C for *encryption*, then relayed by Wonderwall back to the client in encrypted form. In this case, the messages which circulate on the internal network are left in unencrypted form. The Wonderwall provides a single point of decryption and encryption for all messages entering and leaving the internal network.

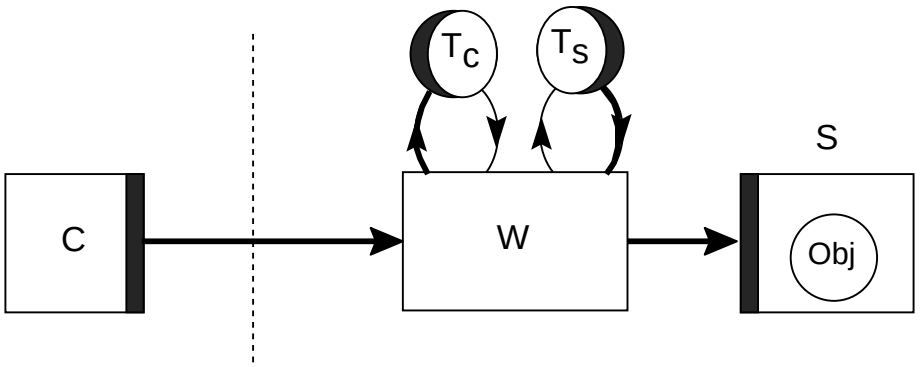

Figure 4.3: *Wonderwall inserted into encrypted link.*

The second solution is shown in Figure 4.3. In this model, encypted messages are circulated on both the internal and external networks. The advantage of deploying Wonderwall in this way is that the firewall can be inserted where an encrypted link already exists. No disruption is caused, and the server needs no modification. This requires the use of two external Transformers: the first Transformer is the client Transformer T_C which exchanges encrypted messages with the client, and the second Transformer is the *server Transformer* T_S which exchanges encrypted messages with the server. Sandwiched between these two Transformers is the monitoring and filtering portion of the Wonderwall performing all its operations on unencrypted messages.

4.2 Usage

The external Transformers, which the Wonderwall uses, are defined as CORBA objects. They are not built in to the Wonderwall server process. You are free to implement these Transformer objects (both client and server Transformer) in whatever way you like. Wonderwall defines an IDL interface for the Transformer objects which you must use when writing your implementation. The Wonderwall can be configured so that it automatically calls your Transformer objects as needed. This is explained in the following sections.

4.2.1 IDL

The external Transformers used by Wonderwall, both client T_C and server T_S, are instances of the CORBA interface `IT_WonderwallReqTransformer`. The IDL interface is as follows:

```
// IDL
//
// Wonderwall client/server Transformer interface:

typedef sequence<octet> iiopMessage;

interface IT_WonderwallReqTransformer {

   exception TransformFailedException {
       string reason;
   };

   void transform (inout iiopMessage data,
                   in string host,
                   in boolean sending)
       raises (TransformFailedException);
};
```

The interface features a single operation `transform()` which is called whenever Wonderwall needs a message to be transformed. The first argument `iiopMessage` is the message to be transformed. An `iiopMessage` is declared to be of type `sequence<octet>` which is the data type that CORBA typically uses for buffers of bytes. It is perfectly permissible to pass back a transformed buffer which is a different size to the one received from the Wonderwall. The next argument is the `host` which sent the Request (or is about to receive the Reply). The argument `sending` is used to indicate whether encryption or decryption is required. When a message is sent out of Wonderwall, this flag is true, and encryption is required. When a message is sent inwards, this flag is false, and decryption is required.

There is a single exception `TransformFailedException` which can be raised by the user to abort the message. The user can decide under what circumstances the exception is raised. Wonderwall will react to this error by sending a MessageError error message back to the client, in the case of a

client Transformer, or by sending a MessageError error message to the server, in the case of a server Transformer.

4.2.2 Implementation

The implementation of both client and server Transformers is flexible. Apart from keeping to the rule that you should encrypt and decrypt messages when Wonderwall expects, you have complete freedom in implementing the Transformers. The Transformers can be implemented in any convenient language for which a CORBA mapping exists. They may be implemented in a standalone server, or built into some existing server on the internal network.

As an example, we outline here the skeleton of a client Transformer which is implemented in the Java programming language with the help of OrbixWeb:

```
// IDL
package wwallXformerBoth;

import IE.Iona.Orbix2.CORBA.* ;
import java.util.Random;

public class Transformer extends IT_reqTransformer
{
   public boolean transform (_sequence_Octet data,
                             String host,
                             boolean is_send)
   {
       if (is_send) {
          //
          // Implement an algorithm to encrypt ‘data’
          //
          ...
       }
       else {
          //
          // Implement an algorithm to decrypt ‘data’
          //
          ...
       }
       return true ;
   }
}
```

Note that the encryption algorithm is allowed to change the size of the sequence buffer and return a transformed sequence of a different length. If you wish to do this you will have to remember reallocate the size of the sequence buffer before completing the transformation. A full example of an OrbixWeb client Transformer is available in the demos directory of your Wonderwall

distribution. The demo is based on the architecture of Figure 4.2, where the client Transformer is implemented in server S.

4.2.3 Configuration

Configuring the Wonderwall to use either a client Transformer, or a client and server Transformer is quite straightforward. Once a client and server Transformer have been implemented, insert two lines like the following into the configuration file (typically called `iipproxy.cf`):

```
################################################
# Configure client and server Transformers...
#
client-transformer \
   bind (":myServer","internalHostA") \
   interface IT_WonderwallReqTransformer
server-transformer \
   bind (":myServer","internalHostA") \
   interface IT_WonderwallReqTransformer
```

This setup is appropriate when both a client and server Transformer object have been implemented in server `myServer`, on host `internalHostA`. If you do not wish to use the Orbix bind mechanism, you can substitute any form of *`object-specifier`* (as described in Appendix B.2).

Note that since the encrypted Requests will be sent from the Transformer servers to the Wonderwall in unencrypted form, care should be taken that these connections cannot be intercepted. For example, the Transformer server could be run on the Wonderwall machine itself. As long as the firewall protects the Transformer, this is safe.

Chapter 5
Using the Wonderwall with OrbixWeb

OrbixWeb, IONA's CORBA-on-Java product, contains built-in support for the Wonderwall. This allows you to use the Wonderwall either as a simple intranet request-routing server which passes IIOP messages from your applet, via the web server, to the target server, or as a full-blown firewall proxy which can filter, control and log your IIOP traffic. This makes the Wonderwall much easier to use and administer, as IORs no longer need to be proxified to be used with OrbixWeb, and in low-security configurations the need to list server objects in the Wonderwall configuration file is removed.

OrbixWeb 3 has built-in support for the Wonderwall. However, if you use OrbixWeb 2.0.1, you should contact `support@iona.com` in order to receive an up-to-date patch for it which supports the Wonderwall.

5.1 Using the Wonderwall with OrbixWeb as an Intranet Request-Router

If you simply wish to provide a way for your OrbixWeb applets to contact servers which reside on hosts other than the one your web server is running on, and you don't care about security issues, then the Wonderwall will provide this capability as an intranet request-router for IIOP. The file `intranet.cf` is used in this configuration, so the Wonderwall command line is as follows:

```
iiopproxy -config intranet.cf
```

This mode of operation requires no configuration, apart from setting your daemon port and domain name – any server can be connected to using the Wonderwall, and any operation can be called.

5.2 Using the Wonderwall with OrbixWeb as a Firewall Proxy

To run the Wonderwall in a traditional secure mode, as detailed elsewhere in this manual, use the file `secure.cf`. The Wonderwall command is as follows:

```
iiopproxy -config secure.cf
```

This mode of operation requires that the target objects and operations be listed in the configuration file. For more details, see Appendix B, which provides a guide to using the Wonderwall access control lists and object specifiers.

5.3 OrbixWeb Configuration Parameters used to support the Wonderwall

OrbixWeb now has automatic support for the Wonderwall built-in to its client-side. This allows OrbixWeb to transparently attempt to connect to any IIOP servers via the Wonderwall if a connection attempt fails using the default direct socket connection mechanism. It also means that the Wonderwall can be used to

- provide HTTP tunneling for OrbixWeb-powered Java applets.
- provide automatic intranet routing capability for OrbixWeb-powered java applets, in order to avoid browser security restrictions.
- use OrbixWeb applications and applets with the Wonderwall, with no code changes.

5.4 Configuring OrbixWeb to use the Wonderwall

In order for OrbixWeb to use the Wonderwall, it must be configured with the Wonderwall's location. The following configuration parameters are used for this purpose:

```
OrbixWeb.IT_IIOP_PROXY_HOST
OrbixWeb.IT_IIOP_PROXY_PORT
```

`IT_IIOP_PROXY_HOST` should contain the name of the host on which the Wonderwall is running, and `IIOP_PROXY_PORT` should contain its IIOP port. These parameters may be set using any of the supported configuration mechanisms – see the release notes of the OrbixWeb version you're using for details. For example, here's a fragment of a HTML file which uses applet parameters:

```
<applet code=GridApplet.class height=300 width=400>
  <param name="OrbixWeb.IT_IIOP_PROXY_HOST"
value="wwall.iona.com">
  <param name="OrbixWeb.IT_IIOP_PROXY_PORT" value="1570">
</applet>
```

5.5 Configuring OrbixWeb to use HTTP Tunneling

"HTTP tunneling" is a mechanism for traversing client-side firewalls. Each IIOP Request message is encoded in HTTP base-64 encoding, and a HTTP form query is sent to the Wonderwall, containing the IIOP message as query data. The IIOP Reply is then sent as a HTTP response.

Using HTTP tunneling allows your applets to be used behind a client's firewall, even when a direct connection (or even a DNS lookup of the Wonderwall's hostname) is impossible.

In order to use HTTP tunneling, you must use the new `ORB.init()` API call to initialize OrbixWeb, whether you're using OrbixWeb 2.0.1 or 3.0. Again, see the release notes for whatever version you're using for details on how to use this.

This is necessary as it allows OrbixWeb to retrieve the **codebase** from which the applet was loaded. The codebase is then used to find the Wonderwall's interface for HTTP tunneling – a pseudo-CGI-script called `"/cgi-bin/tunnel"`. For more information on what the codebase is used for in Java, look at `http://www.javasoft.com/`.

The Wonderwall should be used as the web server which provides the applet's classes, as an untrusted Java applet is only permitted to connect to the server named in the codebase parameter. However, it is permissible to provide your main web site's HTML and images from another web server, such as Apache, IIS or Netscape, and simply refer to the Wonderwall web server in the applet tag, as follows:

```
[ in the file http://www.iona.com/demo.html ]

<applet code=GridApplet.class
codebase=http://wwall.iona.com/GridApplet/classes
height=300 width=400>
</applet>
```

With this setup, your HTML, images, and so on are loaded from the main web site (`www.iona.com`), but your applet code is loaded from `wwall.iona.com`, and as a result the applet can open connections to that host. For greater efficiency, it's advisable to make a ZIP, JAR and/or CAB file containing the classes used by your applet, and store them on the Wonderwall site as well (in fact, regardless of whether you're using the Wonderwall, this is generally a very good idea).

It is also feasible to provide a Wonderwall set up to support HTTP tunneling on the same machine as the real HTTP server, by using a different port number from the default port 80; however, bear in mind that some sites may only support HTTP traffic on port 80, the standard port, so this may restrict your applets' potential audience slightly.

You should ensure that the applet's classes are available in the directory you named in the codebase URL – in the example above, this would be `GridApplet/classes`. This directory path is relative to the directory named in the Wonderwall configuration file's `http-files` parameter.

If you want an application to use HTTP tunneling, or would prefer to override an applet's HTTP tunneling setup [1], three more configurable parameters are provided:

```
OrbixWeb.IT_HTTP_TUNNEL_HOST
OrbixWeb.IT_HTTP_TUNNEL_PORT
OrbixWeb.IT_HTTP_TUNNEL_PROTO
```

`IT_HTTP_TUNNEL_HOST` should contain the name of the host on which the Wonderwall is running, `IT_HTTP_TUNNEL_PORT` should contain its HTTP port, and `IT_HTTP_TUNNEL_PROTO` should contain the protocol used – currently the only protocol value supported for HTTP tunneling is "*http*".

We have observed an issue relating to the use of HTTP tunneling in OrbixWeb 2.0.1. The OrbixWeb runtime classes attempts to retrieve a file from the web server in order to test if HTTP tunneling can be used; normally, the runtime can differentiate between the exception thrown when the connection fails and the exception thrown when the target file does not exist. However, some implementations of the HTTP support classes (`java.net.URLConnection`) in the different web browsers throw the same exception for "file not found" as for "connection failed" – and hence OrbixWeb treats this as an unusable connection.

If you observe OrbixWeb failing to use HTTP tunneling when it should be, this problem is most likely occurring. The workaround is to create a file in the root directory of the Wonderwall's `http-files`[2] directory hierarchy, named `robots.txt`[3] – insert the following line in the file (it needs to be non-empty due to another bug in one of the browsers):

```
User-Agent: *
```

This workaround is no longer necessary with OrbixWeb 3.0.

The Wonderwall supports HTTP 1.1 and HTTP 1.0's Keep-Alive extension. This means that TCP connections between the client and the Wonderwall (or between a HTTP proxy and the Wonderwall) will be "kept alive", i.e. more than one HTTP request can be sent across them. This greatly increases the efficiency of HTTP.

---

1. Note that OrbixWeb 2.0.1 will use the applet's CODEBASE URL from the HTML applet tag, and will not allow you to override it with these parameters.
2. `http-files` is a configuration file parameter which specifies the directory the web server serves documents from; for example, `http-files /var/web` means that a request for `/robots.txt` is served from the file `/var/web/robots.txt`.
3. The file `/robots.txt` is used by web-crawler robots to find control information about the site in question. In the case of HTTP tunneling, it is used because an access to this URL is of no importance to the site administrators, and rather than worrying people unduly about accesses to strange URLs, we decided an access to a moderately strange, but well-documented, URL would be preferable. The robot commands provided act as a no-op, by the way, so any robots who drop by your site will be entirely unaffected.

5.6 Manually Configuring OrbixWeb to Test Tunneling

In order to test HTTP tunneling or IIOP via the Wonderwall, two more configurable parameters are provided; these are

```
OrbixWeb.IT_IIOP_PROXY_PREFERRED
OrbixWeb.IT_HTTP_TUNNEL_PREFERRED
```

If either of these are set to `true`, then that connection mechanism will be tried first, before the direct connection is attempted.

Appendix A
iiopproxy and iortool

The Wonderwall is shipped with just two binary files, `iiopproxy` and `iortool`. The `iiopproxy` implements the firewall itself, while `iortool` is a useful utility for manipulating object references. Both of these commands each have a number of options as detailed below.

A.1 The iiopproxy process

The command `iiopproxy` is usually run as a daemon process to monitor both the dedicated IIOP port and the HTTP port on the bastion host. It is the key component of the Wonderwall and combines the functionality of IIOP gateway with a full HTTP server. The `iiopproxy` will generally be launched automatically using the `inetd(8)` on Unix (see the install notes) and remains permanently active, monitoring the designated ports, until it is explicitly killed. The syntax of iiopproxy is as follows:

```
iiopproxy [options]
```

The *options* may consist of one or more of the following switches:

`-config` *file*	Specify the pathname of the Wonderwall configuration file (see Appendix B). When iiopproxy is run as a daemon process this should be an absolute pathname.
`-debug` *n*	The debug level can be set to three values 0, 1 or 2. The value 0 means no diagnostics, 1 means minimal diagnostics and 2 means full diagnostics. The default is 1.
`-fg`	Debugging option meaning don't fork when a new connection arrives. This is of limited usefulness unless you're debugging, and may cause trouble.
`-help`	Give usage information on these command switches.
`-httpport` *port*	Specify the port to listen on for HTTP requests (this can also be specified in the Wonderwall configuration file—but the value specified by `-httpport` takes precedence).
`-inetd`	Specify that the `iiopproxy` is running from `inetd(8)` as a daemon process. This causes the `inetd` process to listen to the ports on behalf of the Wonderwall. When this flag is not specified, the Wonderwall listens on these ports itself. Wonderwall uses the current user ID as an identification when binding to servers. Hence the servers must be registered using `putit`, and `chmodit`'ed so that the user running the Wonderwall has invoke and launch rights to the servers.
`-log` *logfile*	Send the log output into the named file, rather than to `stderr`, the standard error file descriptor.
`-port` *port*	Specify the dedicated IIOP port for the Wonderwall (this can also be specified in the Wonderwall configuration file—but the value specified by `-port` takes precedence).
`-user` *username*	Run as a specified user. If the Wonderwall needs to use a privileged port (one under 1024), it is safer to run as a normal, unprivileged user once the port is acquired, so you should use this switch. Wonderwall uses the current user ID as an identification when binding to servers. See the `-inetd` switch.
`-v`	Print version information for `iiopproxy`.

Typically, when testing the Wonderwall the `iiopproxy` can be run from the command line as in the following example:

```
% iiopproxy -debug 2 -config iiopproxy.cf >& iiop.log
```

where the configuration file is called `iiopproxy.cf` and the output is logged to the file `iiop.log`. When running `iiopproxy` from `inetd(8)` you would typically use a command line like the following:

```
iiopproxy -inetd -config /etc/iiopproxy.cf
```

The Wonderwall sends its output to the system log by default.

See the install guide for recommendations on how to set up the `iiopproxy` to run as a daemon process from `inetd(8)`.

A.2 The iortool Utility

The Wonderwall requires a certain amount of manipulation and use of IORs. In particular, the administrator of the Wonderwall needs to maintain a database of objects both in their original form (for use behind the firewall) and in their proxified form (for use by remote clients). To make this task easier, the Wonderwall product is shipped with a utility `iortool` to help you read and edit IORs.

The iortool utility is a general purpose tool which allows you to view, edit or even create IORs. It may be used with IORs generated by any ORB but some of its features are specific to Orbix. The syntax of the iortool utility is:

```
iortool {-ior | -rxr | -view} iorfile
iortool {-ior | -rxr | -view} \
   [-proxy [-host host] [-port port]] iorfile
iortool {-ior | -rxr | -view} -manual
```

There are three basic forms of iortool usage. The first form is used to view the IOR which is stored in iorfile. The second form is used to edit the IOR in iorfile. The third form is used to create a new IOR where the user is prompted for input at each stage in the creation of the IOR. The options are as follows:

`-host` *`host`*	Used, in conjunction with the `-proxy` option, to specify the new proxy *`host`* which will be embedded in the IOR. If this option is not specified, the host in the IOR is left unchanged.
`-ior`	Specifies that the IOR should be printed in CORBA standard stringified format.
`-manual`	Used to create an IOR interactively. The `iortool` proceeds to create a new IOR and the user is prompted along the way to provide all of the information needed.
`-port` *`port`*	Used, in conjunction with the `-proxy` option, to specify the new proxy *`port`* number which will be embedded in the IOR. If this option is not specified, the port in the IOR is left unchanged.

`-proxy`	Used to specify that the `iortool` is being used to edit the IOR in *`iorfile`*. This option is intended to be followed by one or both of `-h` *`host`* or `-p` *`port`*. It allows the user to easily create a proxified IOR from the given *`iorfile`*. By specifying both the proxy host (using `-h host`) and the proxy port (using `-p port`)
`-rxr`	Specifies that the IOR should be printed in (Wonderwall specific) readable hex representation RXR.
`-view`	Specifies that the IOR should be printed in a human readable format. This option only works when applied to IORs generated by Orbix.

The first way of using the `iortool` utility is as a tool for translating IORs between different formats. There are three output formats which may be selected using one of the flags `-ior`, `-rxr` or `-view`. For example, given an IOR stored in the file `/tmp/gridiiop.ref`, it may be printed out in the standard IIOP stringified format using the `-ior` flag:

```
% iortool -ior /tmp/gridiiop.ref
IOR:000000000000000d49444c3a677269643a312e300000000000000001
000000000000004c000100000000015756c7472612e6475626c696e2e69
6f6e612e6965000009630000000283a5c756c7472612e6475626c696e2e69
6f6e612e69653a677269643a303a3a49523a67
```

The `-rxr` option writes out the IOR in readable hex format RXR (see section 3.2 on page 21 for the details of this format). Note that the RXR format is specific to the Wonderwall. An example of RXR format is obtained as follows:

```
% iortool -rxr /tmp/gridiiop.ref
RXR:_______%0dIDL:grid:1.0_______%01_______L_%01_____%15ultra.dub
lin.iona.ie__%09c___(:%5cultra.dublin.iona.ie:grid:0::IR:grid_
```

The `-view` option writes the IOR in human readable format. Note that this option can only be used to parse Orbix IORs. For example:

```
% iortool -view /tmp/gridiiop.ref
[IOR: type "IDL:grid:1.0": [IIOP1.0 host=ultra.dublin.iona.ie port=2403 \
[ObjectKey "RXR::%5cultra.dublin.iona.ie:grid:0::IR:grid_:"]]]
```

The second way of using an IOR is to edit an existing IOR. This is done via the option `-proxy` (in addition to the output format specifier which is one of `-ior`, `-rxr` or `-view`) which is used in conjunction with the option `-host` and `-port`. For example:

```
% iortool -x -h host.iona.com -p 1570 -i /tmp/gridiiop.ref
IOR:000000000000000d49444c3a677269643a312e3000000000000000000100000
0000000048000100000000001270726f7879686f73742e696f6e612e696500062
2000000283a5c756c7472612e6475626c696e2e696f6e612e69653a677269643a3
03a3a49523a6772696400
```

The new IOR will have the specified host and port number embedded in it. These become the host and port which the client will attempt to connect to when it uses the new IOR.

Finally the `-manual` option can be used to create an IOR from scratch. For example, if you enter the command

```
% iortool -manual -ior
```

you will be prompted to enter each of the requisite fields of an IOR (as specified by the IIOP standard) and the resulting IOR will be written to the standard output as an IIOP stringified IOR, as specified by the `-ior` option. Note that this is not the standard way of producing IORs, and it is recommended that novice users avoid this option altogether.

Appendix B
Configuration

The heart of the Wonderwall's configuration is its configuration file `iiopproxy.cf`. This is read when the Wonderwall is started. It will also be read again if the file changes during the Wonderwall's operation (although only for new clients that connect after the file is read, not for existing client sessions).

The file takes the format of a standard UNIX configuration file. It is read line-by-line, with anything between a '#' (number sign) and the end of line is ignored as a comment, and entries can be continued onto the next line using the '\' (backslash) character. The configuration entries are case-sensitive. A sample configuration file can be found in section 2.2 on page 12.

It is convenient to divide the contents of the configuration file into three parts:

- Basic Settings.
- List of IORs.
- Access Control List.

This appendix provides a complete description of the configuration settings.

B.1 Basic Settings

`activated-port-range` *`lo hi`*

The TCP/IP port range (inclusive) that an internal server can use. The default lower range is 1024, and the default upper bound is 65535. If you wish to tighten security, you may want to restrict the port range to whatever is used in the internal hosts' `Orbix.cfg` files, in the IT_DAEMON_SERVER_BASE parameter.

`allow-binary-principals` *`boolean`*

If the value of *boolean* is `on`, then principals (usernames attached to incoming IIOP requests) are sanitized by the Wonderwall for server activation purposes, and any non-username characters cause the principal to be replaced with the string `iiopproxy`. Non-username characters are alphanumeric characters, plus the characters _, -, +, =, . and ,.

`domain` *`dns-domain`*

The DNS domain name used for the Wonderwall's hostname should be specified here.

`hostname` *`hostname`*

Specifies the hostname (or IP address) of the machine the Wonderwall is running on. If this is specified on a machine with multiple IP interfaces, then the Wonderwall will bind to the interface with that hostname. If it is left unset, the hostname will be determined automatically.

`http-files` *`directory`*

Sets the directory under which the Wonderwall proxy searches for files when behaving as a HTTP server. The files are fetched in response to HTTP requests incoming on the port specified by `http-port`.

`http-idiosyncrasy` *`user-agent idiosyncrasy [...]`*

Unfortunately, the HTTP support built into the various browser's Java runtimes are not always bug-free; as a result, the Wonderwall may need to be informed of bugs present in certain versions.

The user-agent string indicates the value used by the Java runtime to identify that particular browser, and is usually (but not always) in the format BrowserName/version, for example, `JDK/1.1`. You can use simple glob-style pattern matching can be used here, so `JDK/*` matches all versions of the JDK.

The idiosyncrasy parameters use the following keywords:

KEYWORD	DESCRIPTION
`none`	No idiosyncrasies; full HTTP 1.0 or HTTP 1.1 compliance.
`newline-after-post`	Expect to see a redundant newline after every HTTP post operation. Most JDK 1.0.2-derived Java runtimes do this.
`no-keepalive`	Inhibit the use of HTTP Keep-Alive even if the browser says it supports it.

Table 5.1:

`http-keepalive-timeout` *`timeout`*

Specifies how long, in seconds, the Wonderwall should wait for a new message to arrive from a HTTP 1.1 connection before closing it down, requiring the client to reconnect. The default value is 600 seconds.

`http-port` *`httpport`*

Sets the port number which the Wonderwall uses to receive HTTP requests and HTTP tunnelled IIOP messages. Typically, this port is set to the standard value 80. If it is set to 0, the HTTP server capability of Wonderwall is disabled.

`iiop-idle-timeout` *`timeout`*

Specifies how long, in seconds, the Wonderwall should wait for a new message to arrive from a client connection before closing it down, requiring the client to reconnect. The default value is 3 hours.

`log [requests] [replies] [request-bodies] [reply-bodies]`

Specify what additional messages should be sent to the syslog.

FLAG	INFORMATION LOGGED
`requests`	Headers of messages sent by client.
`replies`	Headers of messages sent by server.
`request-bodies`	Contents of all request messages.
`reply-bodies`	Contents of all reply messages.

Table 5.2:

`log-facility` *`facility`*

Specifies the name of the `syslog(3)` facility which the Wonderwall should log its output to. The facility parameter should be set to the name of the facility without the leading `LOG_` prefix, in lowercase; for example, to cause the Wonderwall to log using the `LOG_DAEMON` facility, use `log-facility daemon`.

`orbixd-iiop-port` *`port`*

If you are using the Wonderwall with Orbix or OrbixWeb servers on the internal network, and you wish to use the `bind` form of `object-specifier` in the Wonderwall configuration file, then it is necessary to specify this port. The port is set to the port which the Orbix daemon uses, on the internal network, to receive IIOP messages. (In the default configuration, this port is 1571. It may also be set explicitly via the environment variable IT_IIOP_PORT, in the environment in which the Orbix daemon process is run.)

`port` *`port-number`*

This allows you to specify which *`port-number`* the Wonderwall will listen on. This value can also be specified using the `-port` command-line parameter when starting the proxy.

`pseudo-orbixd` *`boolean`*

This option only needs to be set if the Wonderwall receives messages from certain older versions of OrbixWeb clients. It specifies whether the Wonderwall should emulate specific aspects of an Orbix daemon (`orbixd`) in order to allow clients to connect to Wonderwall-protected servers using `bind()`. The value of *`boolean`* can be set to either `on` or `off`. The default setting is `off`.

`server-open-timeout` *`timeout`*

Specifies how long, in seconds, the Wonderwall should retry connecting to a server which has been activated by its `orbixd`. The default value is 15 seconds.

`strict-host-matching` *`boolean`*

If the value of *`boolean`* is `on` then hostnames will be matched using string comparisons (this is the default). If it is `off`, hostnames will be matched using DNS name resolution.

B.2 List of IORs

`allow-unlisted-objects` *`boolean`*

If the value of *`boolean`* is `on`, it allows the Wonderwall to dynamically update, and add to, its internal table of known objects. For example, if a client attempts to connect to an unknown IOR (not registered using one of `object`, `server` or `persistent-object`) the Wonderwall will automatically add this IOR to its internal list of known objects, assuming *`boolean`* is `on`. The default value of *`boolean`* is `off`.

Note that, just because an IOR is automatically added to this list, does not mean that the client is necessarily granted access. All messages must still be filtered by the Access Control List, in the usual way.

`object` *`tag`* [`wild` *`wildcardflags`*] *`object-specifier`*

This entry is used to define a *`tag`* which is used throughout the configuration file to refer to an object or group of objects. An entry is made in a runtime table which records all objects known to Wonderwall.

At present the Wonderwall supports four different forms of *object-specifier*:

- An object-specifier beginning with the keyword "bind" is used to specify the object using a pseudo-bind syntax (which closely resembles the syntax of _bind() as used by a regular OrbixWeb client—see section 1.5.2).
- An object-specifier that begins with the characters "IOR:" introduces an IOR coded as a standard CORBA stringified object reference (see section 2.3).
- An object-specifier that begins with the characters "RXR:" introduces an IOR encoded using the readable-hex-representation (explained in detail in section 2.3).
- An object-specifier that begins with a "/" is assumed to be the absolute pathname of a file where the IOR is stored (either in "IOR:" or "RXR:" format).

If the wild parameter is specified, any attempts to match this object with a request from the internet will ignore that aspect of the object key. Since this requires examination of the object key, it is not interoperable and depends on support in the Wonderwall code for the server ORB vendor's object key format. Currently, Orbix and OrbixWeb are supported. The supported parameters for wild are as follows:

WILDCARD FLAG	EFFECT
host	Ignore the hostname used in the object key.
server	Ignore the server name used in the object key.
marker	Ignore the object marker used in the object key.
ifmarker	Ignore the interface marker used in the object key.

Table 5.3:

Note that the object entry is suitable for listing an IOR whether the respective server is activated or persistent. When an object is listed as an object entry, Wonderwall will use the facilities provided by the IIOP protocol to check, first of all, the host and port where the server is currently located. The Wonderwall will use this new host and port information to forward messages to the server.

This ensures that Wonderwall functions correctly with activated servers. In Orbix, for example, such servers are started automatically and have host and port assigned by the Orbix daemon process. When Wonderwall contacts the daemon via IIOP, it will be told the current host and port of the particular server.

`persistent-object` *`symbol`* [`wild` *`wildcardflags`*] *`ior`*

The `persistent-object` entry can *only* be used when the object is in a persistent server. It is almost identical to the `object` entry. The only difference is that, in this case, Wonderwall assumes the listed *`ior`* specifies a direct connection to the server. A precautionary message, to determine the actual host and port, is not sent in advance of the real request. The advantage is a gain in efficiency when used in conjunction with persistent servers.

`server` *`tag object-specifier`*

Exactly equivalent to the entry:

`object` *`tag`* `wild marker ifmarker` *`object-specifier`*

It generates a tag which can be used to refer to all of the objects common to a particular server (hence the name). It is provided as a standalone keyword because it is useful for tagging Factory objects (see section 1.6).

`client-transformer` *`object-specifier`*

Specifies the object which implements an external client transformer for the Wonderwall. A client transformer will not be used unless this line is present in the configuration file (see section 4.2.3).

`server-transformer` *`object-specifier`*

Specifies the object which implements an external server transformer for the Wonderwall. A server transformer will not be used unless this line is present in the configuration file (see section 4.2.3).

`use-ipaddr-in-iors` *`boolean`*

Specifies whether IORs created by the Wonderwall should contain the host's IP address or its hostname. If the hostname the Wonderwall is running on will not be resolvable using DNS to hosts outside the Wonderwall's domain, then this should be set to true.

B.3 Access Control List

The most important part of the configuration is the Access Control List (ACL). This specifies which operations can be accessed on which objects, along with extra conditions and flags.

The ACL is read from the first rule encountered to the last, and is processed by the ACL testing code in that order. This means that you can specify broad filters first, to remove potentially dangerous or unknown features such as Service Contexts, and then go on to allow specific operations on objects after that.

There is no limit to the number of rules in the ACL. If no rules match the message, it is blocked.

Each line is treated as one rule. Longer rules can be continued onto consecutive lines using the '\' (backslash) character.

`allow [keyword [parameter] ] [keyword [parameter]] ...`

This entry will allow any request which matches the given rule. Each specified *keyword* on the line must match for the rule to match (sometimes a *keyword* also has an associated *parameter*). If a *keyword* is not specified on a rule line, that aspect of the message is ignored for purposes of the ACL match.

`deny [keyword [parameter] ] [keyword [parameter]] ...`

This entry will deny any request which matches the given rule. Each specified *keyword* on the line must match for the rule to match (sometimes a *keyword* also has an associated *parameter*). If a *keyword* is not specified on a rule line, that aspect of the message is ignored for purposes of the ACL match.

Keywords Used in Rules

The following keywords may only appear as parameters to the `allow` or `deny` rules:

`...ipaddr` *ipaddress*`[/`*mask*`]`

Match if the client IP address equals the given *ipaddress*. The optional *mask* parameter specifies a bit-wise mask. Only these bits will be used to compare the IP addresses. Some common masks are:
255.255.255.255 all bits - default,
255.255.255.0 class C bits,
255.255.0.0 class B bits,
255.0.0.0 class A bits.
Note that this option should not be relied upon to provide security since the client IP address can be faked.

`...log`

If this keyword appears in a rule which successfully matches, the message header details will be written to the system log. If you had `log requests` set in your configuration file this would be redundant since the header would be logged anyway.

`...msgtype` *type*

Match on message types. The type can be one of the following: `Request`, `Reply`, `CancelRequest`, `LocateRequest`, `LocateReply`, `CloseConnection` or `MessageError`. See section 3.4 on page 24 for more details on IIOP message types.

`...object` *symbol*

Match if the object being accessed is identified by *symbol* (this rule only applies to incoming Requests). The symbol must have already been declared in an earlier `object` or `persistent-object` entry.

`...object-host` *hostname*

Match if the object being accessed is located on the host identified by *hostname*.

`...op` *operation*

Match if the operation in a Request is the same as the operation string `operation` (this rule only applies to incoming requests).

`...principal` *p*

Match if the `principal` sending a request message is the same as the readable-hex-representation byte string *p* (see section 3.2 on page 21 for more information on this format). Note that principals are very easy to forge so this parameter does *not* provide any security.

`...servicecontexts` *sclist*

Match if the an incoming request uses one or more IOP Service Contexts and if one or more of the Service Context IDs used is listed in the *sclist* parameter. IOP Service Contexts are a mechanism which, according to the IIOP spec, allows "service-specific context information" to be passed along with requests and replies. In keeping with the firewall philosophy of "anything not expressly permitted is denied", it is suggested that these are filtered out until a future stage when they become necessary[1] at which point each can be enabled on a specific basis.

The format of the *sclist* parameter is as follows: Each Context ID is represented as a positive integer (these integer IDs are assigned by the Object Management Group to uniquely denote a particular type of Service Context). Multiple Context IDs can be listed, separated by ',' (comma) characters. A range of Context Ids can be matched by listing the start ID, a '-' (dash) character, and the end ID. The string `all` is used to match one or more Context Ids, and the string `max` is used to denote the upper bound of the Context ID range. Here are some examples:

```
deny servicecontexts 1-3,5,7,9-20
deny servicecontexts 0-5,7-max # all except 6
deny servicecontexts all # one or more Service Contexts
```

Note that if a rule allowing specific service contexts is followed by a wildcard deny rule, the effect is non-intuitive. A request containing both permitted and denied service contexts would be forwarded, as it would hit the `allow` rule first. Caution is advised here.

`...unlisted-object`

Match if the object being accessed is an unlisted object, that is, it has been specified as a target by the client-side ORB.

1. A service contexts is used, for example, by the CORBA Transaction Service.

Index

R

S

T

U

V

W