

# *Joint Revised Submission CORBA/Firewall Security*

---



*BEA Systems, Inc.*

*Borland International, Inc.*

*Expersoft Corporation*

*FUJITSU LIMITED*

*International Business Machines Corporation*

*IONA Technologies PLC*

*NEC Corporation*

*Netscape Communications Corporation*

*Oracle Corporation*

*Sun Microsystems, Inc.*

*May 18, 1998*

*OMG Document orbos/98-05-04*



Copyright 1998 BEA Systems, Inc.  
Copyright 1998 Borland International, Inc  
Copyright 1998 Expersoft Corporation.  
Copyright 1998 FUJITSU LIMITED.  
Copyright 1998 International Business Machines Corporation.  
Copyright 1998 IONA Technologies PLC.  
Copyright 1998 NEC Corporation  
Copyright 1998 Netscape Communications Corporation.  
Copyright 1998 Oracle Corporation.  
Copyright 1998 Sun Microsystems, Inc.

All rights reserved.

The companies listed above hereby grant to the Object Management Group, Inc. (OMG) and OMG members, permission to copy this document for the purpose of evaluating the technology contained herein during the technology selection process by the appropriate OMG task force. Distribution to anyone not a member of the Object Management Group or for any purpose other than technology evaluation is prohibited.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems— without the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

## *Contents*

---

<b>1. Preface .....</b>	<b>5</b>
1.1 Introduction .....	5
1.2 Submission Contacts .....	5
1.3 Submission Supporters .....	8
1.4 Guide to Submission .....	8
1.5 Items Not Addressed in this Submission .....	8
1.6 Proof of Concept .....	8
<b>2. Response to RFP Requirements .....</b>	<b>9</b>
2.1 RFP Proposal Specific Requirements .....	9
2.1.1 Mandatory Requirements .....	9
2.1.2 Optional Requirements .....	10
2.2 Resolution of RFP Requirements .....	10
2.2.1 Mandatory .....	10
2.2.2 Optional .....	11
<b>3. Design Rationale and Background .....</b>	<b>13</b>
3.1 Rationale Summary .....	13
3.2 Background: Existing Practice .....	14
3.3 Overview of Specification .....	15
3.4 Architectural Overview- GIOP Proxy Firewall .....	16
3.5 3.5 Firewall Profiles- Requirements .....	16
3.6 3.6 The Rest of This Document .....	17
<b>4. ORB Interoperability through Firewalls .....</b>	<b>19</b>
4.1 Firewall Principles .....	19

---

4.2	ORBs and Firewalls . . . . .	21
4.3	Scope of Firewall Support in CORBA . . . . .	21
4.4	Types of Firewall . . . . .	22
4.5	TCP Firewalls . . . . .	22
4.6	SOCKS. . . . .	24
4.7	GIOP Proxy . . . . .	28
4.7.1	Connection styles . . . . .	28
4.7.2	Callbacks . . . . .	29
4.7.3	IOP/SSL considerations . . . . .	32
4.7.4	GIOP Proxy Interface. . . . .	32
4.8	Firewall tag components . . . . .	35
4.9	Firewall POA Policy . . . . .	39
4.10	Outbound firewalls. . . . .	40
4.11	Traversal algorithm . . . . .	40
4.12	Passing credentials . . . . .	43
4.12.1	SSL Certificates . . . . .	44
4.13	IOP/SSL Considerations . . . . .	44
<b>5.</b>	<b>Bi-directional GIOP . . . . .</b>	<b>47</b>
5.1	Bi-Directional GIOP . . . . .	47
5.1.1	Bi-Directional IOP . . . . .	49
5.2	Bi-directional GIOP policy . . . . .	50
<b>6.</b>	<b>Conformance &amp; CORBA Changes. . . . .</b>	<b>53</b>
6.1	Conformance . . . . .	53
6.2	Changes to CORBA 2.2 . . . . .	54
Appendix A	Consolidated IDL . . . . .	55
A.1	Firewall Module . . . . .	55
A.2	Additions to SSLIOP . . . . .	57
A.3	Additions to the IOP Module . . . . .	58

# *Preface*

---

*1*

## *1.1 Introduction*

The Internet has metamorphosized into a global information resource for private and commercial use. Most organizations would like to take advantage of the powers of the World-Wide-Web and still protect their assets. Therefore, network firewalls were introduced to prevent unauthorized access and attacks, by protecting the points of entry into the network. Currently, there is no standard mechanism for a firewall to identify and control the flow of IIOP traffic.

The intent of the submission is to provide a standard approach to control IIOP traffic through network firewalls, thereby allowing outside access to CORBA applications.

## *1.2 Submission Contacts*

The following lists contact information for the submitters of this document. All questions concerning this submission should be directed to:

Ed Cobb  
BEA Systems, Inc.  
385 Moffett Park Drive  
Sunnyvale, CA 94089  
Phone: +1 408 542 4264  
Fax: +1 408 744 0775  
Email: ed.cobb@beasys.com

Jeff Mischkinsky  
Borland International, Inc.

951 Mariner's Island Blvd.  
San Mateo, CA 94404  
Phone: +1 650 312 5158  
Fax: +1 650 286-2475  
Email: jeffm@visigenic.com

Ken Fleming  
Expersoft Corporation  
5825 Oberlin Drive, Suite 300  
San Diego, CA 92121  
Phone: +1 619 824 4185  
Fax: +1 619 824 4110  
Email: kfleming@expersoft.com

Masayoshi Shimamura  
FUJITSU LIMITED  
Nikko Fudousan Building, 2-15-16, Shinyokohama  
Kohoku-ku, Yokohama 222, Japan  
Phone: +81 45 476 4590  
Fax: +81 45 476 4726  
Email: shima@rp.open.cs.fujitsu.co.jp

Anne Aldous  
IBM Corporation  
11400 Burnet Road  
IZIP: 9133  
Austin, TX 78758  
Phone: +1 512 838 2186  
Fax: +1 512 838 0156  
Email: aldous@us.ibm.com

Martin Chapman  
IONA Technologies PLC  
The IONA Building  
8-10 Lower Pembroke Street  
Dublin 2, Ireland  
Phone: +353 1 662 5255

Fax: +353 1 662 5244  
email: mchapman@iona.com

Michael J. Greenberg  
NEC Systems Laboratory, Inc.  
4 Independence Way  
Princeton, NJ 08540  
Phone: +1 609 734 6142  
Fax: +1 609 734 6001  
Email: mjg@syl.nj.nec.com

Rahul Bhargava  
Netscape Communications Corporation  
M/S MV-061  
501 East Middlefield Road  
Mountain View, CA 94043  
Phone: +1 650 937 2600  
Fax: +1 650 528 4129  
Email: rahul@netscape.com

Sastry Malladi  
Oracle Corporation  
500 Oracle Parkway  
Redwood Shores, CA 94065  
USA  
phone: +1 650-506-8063  
fax: +1 650-654-6211  
email:smalladi@us.oracle.com

Jeff Nisewanger  
Sun Microsystems, Inc.  
901 San Antonio Road, MS UCUP02-201  
Palo Alto, CA 94303  
Email:jeff.nisewanger@eng.sun.com

### *1.3 Submission Supporters*

Companies and contacts supporting this submission are:

John Sebes.  
TIS Labs at Network Associates, Inc.  
3965 Freedom Circle  
Santa Clara, CA 95054  
Phone: +1 408 602 5646  
Email: ejs@tis.com

### *1.4 Guide to Submission*

Chapter 2 restates the requirements described in the original RFP, and discusses how these are addressed in this submission.

Chapter 3 provides design rationale and design goals of the submission.

Chapter 4 provides a description and specification of firewall support in CORBA. In particular it presents three types of firewalls that could be used when interconnecting ORBs across a network. These are TCP, SOCKSv5, and GIOP proxy firewalls. The issues related to client and server side usage are presented together with IDL definitions. Particular attention has been made to ensuring that IIOP/SSL connections can be established through firewalls.

Chapter 5 presents a modification to the existing GIOP/IIOP protocols. This modification permits a server to reuse a connection it has with a client, so that the server may issue requests to callback objects on the client side. This is called bi-directional GIOP.

Chapter 6 provides a conformance statement, and outlines the changes required to CORBA V2.2.

Finally, Appendix A consolidates the IDL defined in this submission.

### *1.5 Items Not Addressed in this Submission*

This proposal does not address the use of SECIOP as a transport mechanism for secure invocations. We will address these items at a later time.

### *1.6 Proof of Concept*

This submission is based on several products and prototypes developed by submitters of this revised specification.



### *2.1 RFP Proposal Specific Requirements*

The following two sections, *Mandatory Requirements* and *Optional Requirements*, were cut and pasted from the original RFP document. We have included them in this section for the convenience of the reader.

#### *2.1.1 Mandatory Requirements*

Proposals shall specify how a firewall can process IIOP to allow CORBA objects managed behind the firewall to have operations invoked on them from the outside world. Proposals shall describe how firewall processing of IIOP can be done to enable firewalls to do the following:

- Permit outside access to some inside CORBA-based application services, and also prevent access to IIOP-based services that should not be accessible from the outside.

Process IIOP as an ordinary application protocol. Proposals shall describe firewalls can meet the above requirement by performing both, either, or neither of the following:

- determine what network traffic is expected to be IIOP (e.g. destination hosts, ports);
- differentiate between IIOP traffic that is permitted to enter the enclave, and IIOP traffic that is not permitted;
- Protect inside target object servers from attack by data streams that are not valid IIOP.

Proposals shall describe whether firewall processing of IIOP depends on:

- Examination of IIOP message header data or IIOP message data, and if so which fields.
- Authentication mechanisms (IIOP with SecIOP or SSL or any other form of authentication) and if so what the use or dependency is.

- Co-ordination of configuration data of firewalls and ORBs, e.g., to ensure that object references (IORS) received by outside invokers can be used to make requests that a firewall may allow.

Note also that this RFP does not require any modification of existing CORBA specifications, e.g. IIOP, SecIOP, or SSL/CORBA. Likewise, responses to this RFP should not require any modification without a very compelling justification.

### *2.1.2 Optional Requirements*

In addition, proposals may describe how firewall processing of IIOP can be done to enable firewalls to do the following:

- Control access at the granularity of specific objects and/or methods and/or any other data that may be specific to an individual object invocation request message;
- Provide differentiated access based on authentication via firewall-to-firewall SSL transport of IIOP, so that authenticated requests can be permitted more access than unauthenticated requests;
- Provide differentiated access when SecIOP is implemented with IIOP, so that authenticated requests can be permitted more access than unauthenticated requests;
- Use SecIOP or SSL to provide private IIOP interaction with selected authenticated outsiders;
- Perform IIOP security functions when SSL or SecIOP is used for end-to-end privacy between invoker and target object;
- Meet mandatory requirements for any other inter-ORB protocols, as well as any optional requirements.
- Proposals may provide IDL interfaces for management and configuration of firewalls.

## *2.2 Resolution of RFP Requirements*

### *2.2.1 Mandatory*

This proposal addresses all the RFP mandatory requirements as listed above, with the following exception. In the case when a pass-through connection is established, as described in section 3.7, the proxy has no visibility to the encrypted byte stream. It is not possible to verify that valid IIOP traffic is flowing after the pass-through connections has been established. However, pass-through connections are only allowed under SSL, so the integrity of the client/server connection is maintained.

The RFP required that the submission describe the information examined by the firewall to process the message. The firewall processing depends on examining the IOR. On, requests, however, the principal value of the Service Context is evaluated.

### *2.2.2 Optional*

The submitters decided not to address the optional requirement of access control of specific objects. This is currently addressed in CORBASEC.

SECIOP is not supported at this time. This proposal does address the use of SSL to provide confidential IIOP with selected outsiders, as well as end-to-end confidentiality.



### *3.1 Rationale Summary*

There are several elements of the rationale for this specification, but all of them stem from one overall goal: better accessibility to CORBA application servers when there is a firewall separating a server from a client. In this context, "better" means that client-firewall-server communication can be enabled and controlled more easily for a broader range of circumstances, with significantly lowered administrative burdens. In other words, ORBs and firewalls currently have a limited form of "peaceful co-existence" that provides satisfactory functionality only in some cases.

Thus, the main goal of this specification is to specify the changes to CORBA that are needed for ORBs to function in a slightly different manner, so that CORBA communication can more easily be handled by firewalls. An additional goal of this document is to provide information on how current firewall techniques can be used to control CORBA communication. This information illustrates the benefits of current techniques, and also the limitations. The need to overcome these limitations is the main driver for this specification.

Interoperable CORBA communication is via the GIOP protocol, which on the Internet is implemented by the IIOP protocol (a mapping of GIOP to TCP transport). Because firewalls control IP networking communication, and because ORBs communicate via IIOP, much of this specification is concerned with various aspects of the ways that firewalls handle the IIOP protocol. It is important to note that there is nothing particularly problematic about IIOP as an Internet protocol, in terms of firewall processing. In fact, this specification does not modify IIOP in any way. Rather, this specification adds to CORBA new data elements (for example, in IORs) that provide clients, firewalls, and servers the information needed for flexible, efficient, controlled firewall traversal. This specification also defines CORBA interfaces that may be used by CORBA software to provide information to a firewall.

## 3.2 Background: Existing Practice

Firewalls today can process IIOP in an effective but limited way. To describe CORBA and firewall communication in this document, we use the term enclave to refer to some set of CORBA objects that are protected by a firewall that controls all network communication between those objects and the outside world. When a client within the enclave communicates with one of these objects, the firewall is not involved. When a client from outside the enclave communicates with one of these objects, the firewall is involved, to ensure that communication occurs only if it is explicitly permitted. For example, at a coarse level of control, an enclave may have some CORBA application server hosts that outsiders can communicate with, and others which are reserves for internal use within the enclave.

The limitations of current practice stem from the two basic requirements for cross-firewall, inter-enclave communication. First, a client must be permitted by its own enclave's firewall to initiate communication to the outside server. That is, the firewall's configuration must include the TCP ports that clients will use to form outgoing TCP connections to exchange IIOP messages with an outside server. Second, the server must be permitted by its enclave's firewall to receive incoming connections from outside clients. Of course, there is much more to firewall processing of IIOP than this, but these two points are the foundation of firewall processing.

The essential problem with IIOP and firewalls is that it is not easy in practice to know in advance (and to represent in a firewall configuration) which hosts and ports will be used for inter-enclave CORBA communication. The host/port addressing information is contained in IORs that describe how to communicate with servers, assuming that clients can contact servers directly. In the inter-enclave case, this assumption does not hold. Clients attempt to contact servers directly. If they are lucky, then the intervening firewalls have been specifically configured to allow the host/port connections needed by the client. Otherwise the client is prevented by a firewall from establishing the required communication. Because the client didn't know that it had to go through a firewall, and didn't know where the firewall was, it was unable to contact the server.

Current technology can mitigate the difficulties on the client side, i.e., allowing the client to go through its enclave's firewall to contact the firewall on the server side. If clients use only TCP ports that are well-known to be for IIOP transport (or are otherwise known in advance to the firewall administrator), then standard TCP-level firewall mechanisms can be used to permit outgoing traffic. Section 4.5 provides more information on TCP-level techniques. If the ports are not known in advance, then Socks proxying techniques can be used to direct outgoing connections first to a Socks proxy, and then outward to the Internet. Section 4.6 provides more information on Socks techniques. In both cases, existing ORBs and firewall techniques are used. Section 4.11 provides more information about outbound firewall traversal.

On the server side, the main difficulty is the difficulty of configuring firewalls to listen for IIOP-bearing connections on all the ports, and destined for all the internal hosts, that are mentioned in the host/port addressing information in any IOR for any object in the enclave that outside clients might be allowed to access. To address this difficulty,

this specification describes new ORB functionality whereby IORs contain the information needed to contact an object's firewall directly. As a result, firewalls can be much more simply configured and managed. A firewall can have one host/port address that clients directly use to contact the firewall that protects the object that it wishes to communicate with.

### 3.3 Overview of Specification

This specification describes changes to CORBA that allow conformant ORBS to provide solutions to several problems in existing CORBA/firewall practice.

The first such solution has been described above, in terms of IORs and firewall addressing. Section 4.8 describes the format of the new fields in IORs that provides the addressing information of firewalls. The extensible format allows for multiple kinds of firewall addressing information to be provided for different kinds of firewalls. In addition, three tags are defined for this format, to support TCP firewalls, Socks proxy firewalls, and GIOP proxy firewalls. GIOP proxies are firewall components that have the capability of providing a very important security function: examination of the IIOP traffic, ensuring that traffic that should be IIOP is in fact IIOP. Section 4.10 describes further functionality of a GIOP proxy, including callbacks (see below).

The second area where this specification provides for new solutions is support for IIOP over SSL. SSL is often used to protect IIOP communication in transit over the Internet between a client in one enclave and a server in another. Therefore, the techniques for IIOP firewall traversal must allow for SSL/IIOP traffic. Furthermore, GIOP proxying techniques must also accommodate SSL. In cases where a GIOP proxy does not need to examine IIOP message data, the SSL can be passed-through unhindered and unexamined. Therefore, the GIOP proxying techniques must accommodate firewall traversal where the IIOP message cannot be viewed. Second, there is support for GIOP proxies that examine IIOP messages, and in some cases must be able to examine IIOP messages transported via SSL. Section 4.13 describes firewall profile tags needed to represent both these cases of IIOP transport via SSL, and also describes data formats for carrying client authentication data in service contexts. This data is needed in the case where a GIOP proxy chains two SSL sessions together (one client-proxy, the other proxy-server) in order to examine the client's IIOP messages. In such cases, the service context is used to pass to the server the authentication data (e.g. X.509 certificate) of the client.

The third area in which this specification provides for new solutions is support for callbacks. The essential problem with callbacks is that the target host of callback operation invocations is a workstation rather than a server host. While an enclave's firewall configuration may permit a few selected inside server hosts to be the target of outside IIOP traffic, it would be very unusual for a firewall configuration to allow any inside workstation to be the target of an incoming TCP connection. Therefore, the current usual callback technique- the server calls back to the client's callback object via a new TCP connection- is simply not acceptable for inter-enclave communication. This

specification supports inter-enclave callbacks in two ways. Chapter 5 describes bi-directional GIOP, a technique whereby a single client-initiated connection can be re-used by the target server to carry IIOP traffic for server invocations on client-side callback objects. Bi-directional GIOP is sufficient for cases where the server that is calling back is also the server that the client originally contacted. In other cases, a server may need to contact a callback object even though the client had not contacted the server. These cases are supported by GIOP proxy functionality described in section 4.10.

### *3.4 Architectural Overview- GIOP Proxy Firewall*

Having described the rationale and approach for each of the three main aspects of this specification, it is important to also provide an overview of GIOP Proxies and the requirements for how firewall profiles should allow GIOP proxies to operate. Several issues arose during the formulation of an approach to CORBA and firewalls, and the GIOP proxy is an important part of an approach (although TCP-level and Socks techniques have their place as well) that covers all of the areas of CORBA/firewall functionality. This document was written to synthesize a number of different mechanisms into a coherent, standards-based specification.

A GIOP proxy is a new network communication component that supports inter-enclave CORBA communication by firewall traversal, SSL support, and callback support. A GIOP proxy could be part of a firewall, or could be deployed as a proxy server behind a firewall that performs only simple TCP-level processing of IIOP traffic.

### *3.5 Firewall Profiles- Requirements*

There are five basic issues that must be addressed for full GIOP proxy functionality.

1. Provide a mechanism which can supports both inbound and outbound GIOP proxies.
2. Allow a client to connect to a server object using information stored in that server's IOR; this means that if an IOR is sent to an external service, such as a Trader, a client which gets a copy of the IOR can still connect to the correct inbound proxy, namely the one corresponding to the server object's enclave.
3. Allow the use of multiple incoming and outbound proxies, so that a client which is within several embedded enclaves can contact a server which itself is within several embedded enclaves.
4. Interoperability - support for third-party ORBs and backwards-compatibility must be considered.
5. Simplicity and efficiency of the proxy code should be maintained if possible.



Chapters 4 and 5 provide the details of new CORBA specifications that are needed to meet these requirements. The overall approach can be summed up as follows:

IORs have profile data that directs clients to proxies;

Clients can use an IDL interface to invoke operations on GIOP proxy objects, to provide information about desired proxy behavior (e.g. target object, callbacks, chaining or pass-through mode for IIOP/SSL);

Proxies decide whether each client's request for each object is permitted in the mode requested;

Proxies use these same interfaces to interact with one another, in cases where there are sequence of proxies between an enclave boundary and a server (or client);

Clients (or upstream proxies) can invoke an operation using either IIOP or SSL/IIOP, and provide the information needed for the proxy to relay the operation invocation to the target server (or downstream proxies);

The combination of bi-directional GIOP and GIOP proxy object interfaces allow for various modes of callback traversal of firewalls.

### *3.6 The Rest of This Document*

Sections 4.1 to 4.7 describe the various kinds of firewall approaches (TCP, Socks, GIOP proxy) and the core issues of firewall traversal, SSL, and callbacks. Sections 4.8 and 4.9 amend the CORBA specification by the addition of definitions of new IOR data and POA and associated POA policy. Section 4.10 amends the CORBA specification by adding definitions of the interfaces to GIOP proxy objects (including support for callbacks). Sections 4.11 and 4.12 describe firewall traversal and how the new interfaces, etc., are used. Section 4.13 amends the CORBA specification by adding firewall profiles for IIOP/SSL. Chapter 5 amends the CORBA specification by the addition of definitions for bi-directional GIOP (which can be used to support callbacks across firewalls).



## *ORB Interoperability through Firewalls*

---

4

This chapter discusses ORB interoperability in networks where firewalls are present. It provides an overview of the issues involved, followed by specifications and descriptions of how inter-ORB interoperability through firewalls can be achieved.

### *4.1 Firewall Principles*

In a CORBA environment, firewalls are used to protect objects from clients in other networks or sub-networks. A firewall will either permit access from another network to a particular object or will prevent it. When access through a firewall is permitted this may be at various levels of granularity. For example, access could be permitted to some objects behind the firewall, or access could be restricted to certain operations on particular objects.

An *enclave* is a group of objects protected by a firewall. The firewall protects the enclave's network (or sub-net) by separating it from other enclaves and/or the Internet at large. The separation is the result of the fact that all communication between the enclave and the outside must pass through the enclave firewall (or one of its firewalls, if there are several). Firewalls have two distinct duties: inbound protection and outbound protection. Inbound protections are used to control external access to internal resources. Outbound protections are used to limit the outside resources that be accessed from within the enclave.

Both aspects of firewall functionality are important for CORBA. A firewall's outbound protection functions should allow inside CORBA application clients and objects to initiate communication with objects outside the enclave. A firewall's inbound protection functions should prevent communication between outside clients/objects and inside objects that the outsiders should not be permitted to communicate with. Without a firewall's outbound protection, clients could access any resources. Without a firewall's inbound protection, all of the enclave's resources are unprotected from the outside world.

Figure 4-1 illustrates an enclave with two inbound firewalls, and one outbound firewall. Note that although the firewalls are logically and functionally separate, they may share the same physical hardware, or even share the same address space.

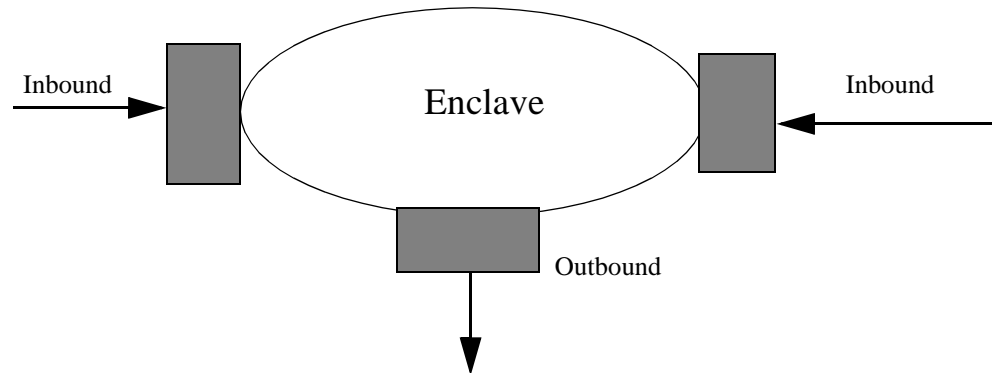


Figure 4-1 An enclave with multiple inbound and outbound firewalls.

Enclaves can be nested, such that an enclave may contain other enclaves in a hierarchical manner. This enables organizations to decentralize firewall access and have different access policies. For example, an engineering department prevents the finance department of the same company from accessing design documents. When enclaves are nested, a sequence of firewalls has to be traversed. A firewall protecting the outer enclave is called either an outermost inbound firewall or an outermost outbound firewall, depending on its type. The outermost inbound firewall represents an entry point into an organization.

Figure 4-2 illustrates a hierarchical nesting of enclaves. The outermost “company” enclave contains two sub-enclaves, “finance” and “R&D”. The “R&D” enclave further contains the “Research” enclave. Firewall “A” is the outermost outbound firewall for the “company” enclave, and firewall “B” is the outermost inbound firewall. Again it is important to note that the distinction between inbound and outbound firewalls is only a logical one, and does not necessarily imply a physical separation or separate address space.

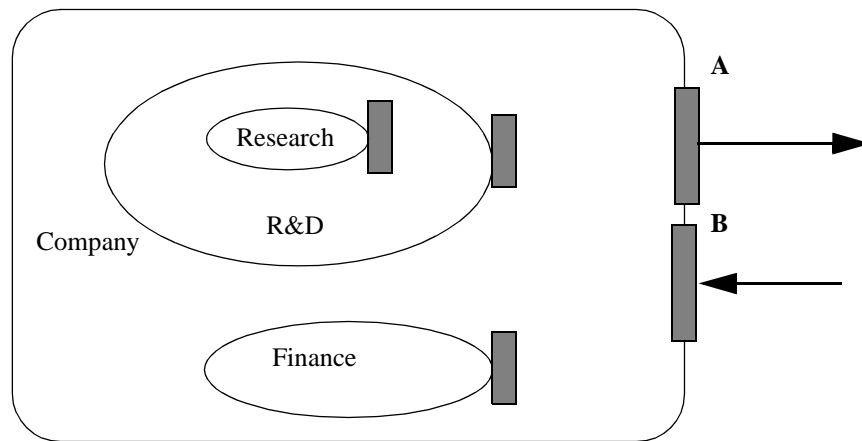


Figure 4-2 A hierarchical set of enclaves.

## 4.2 ORBs and Firewalls

In order to establish a connection to an object in another ORB, two sets of information are required. Firstly the outbound firewalls that need to be traversed must be known, and secondly the inbound firewalls that have to be traversed must be known. This information may be distributed and not known in one place. For example, a client side ORB may only know its first outbound firewall, and that firewall knows the next outbound firewall. This will be particularly true in environments with fixed configurations, such as in intranets and extranets.

Since this specification is trying to cover a wide variety of scenarios (internet, intranet, and extranet), it is not possible to make too many assumptions i.e. it cannot be assumed that a firewall has knowledge to reach another firewall. Indeed the only assumption that can be made is that as a minimum, a client side ORB must know its first outbound firewall, and must know the outermost inbound firewall required to reach an object (this must be known since if there are no outbound firewalls the client ORB has to interact directly with the outermost inbound firewall).

Information about outbound firewalls is configured into the client side ORB. Information about inbound firewalls may be configured into a client side ORB in intranet or extranet configurations, but generally it must be assumed that the client side knows nothing about the server side. The only interoperable means to convey this information is to include inbound firewall information in IORs, within tagged components.

## 4.3 Scope of Firewall Support in CORBA

The prime motivation of this chapter is to describe how it is possible for two ORBs to interoperate when the two ORBs exist in different enclaves, and hence require navigation of firewalls to establish communication.

However since firewalls from different vendors vary considerably, issues related to configuring a firewall (such as to register which host/ports are accessible or which objects) are considered outside of the scope of CORBA, and hence are not defined.

In a similar nature, issues related to configuring an ORB with firewall information is also considered out of scope, since this chapter focuses on interoperability.

## 4.4 *Types of Firewall*

Broadly, there are two types of firewall: transport level and application level.

A transport level firewall allows different resources using different application level protocols to be accessed. Such firewalls neither understand nor care about the type of application protocol being used. Access is based purely on addressing information in the header of transport packets. Hence access decisions are based on where things come from or are going to and not on what is being accessed. Typically access control is performed during connection setup, and if successful any application traffic may pass over the connection. A TCP firewall, for example allows access to FTP, HTTP, or IIOP resources, where access is controlled on which hosts/ports traffic is going between.

Application level firewalls on the other hand are restricted to a particular application level protocol, such as IIOP or HTTP. Access decisions are not only possible based on transport addressing information, but may also be based on specific resources known by the application level protocol. For example, if there are two object that can be accessed through the same host and port, it is possible for the firewall to deny invocations being sent to one object but to allow them for the other. This type of control requires monitoring the traffic after the connection has been established, and hence requires the firewall to understand the application level protocol.

The mechanisms used to interact with a firewall to establish a connection through it is very dependent on the type of firewall it is. Therefore to achieve ORB interoperability it is necessary to define which types of firewall are supported in CORBA. The current specification recognizes three types of firewall types, namely TCP, SOCKSv5, and GIOP Proxy. The specification however is flexible enough in that other firewall types can be added in the future with minimal change.

## 4.5 *TCP Firewalls*

A TCP firewall is a very simple transport level firewall. It performs access control decisions based on address information in TCP headers. For ORB interoperability, TCP firewalls provide the simplest means to protect resources, but at the largest level of granularity i.e. host based control.

A TCP firewall works on a simple address mapping scheme: a connection request received on a certain port of the firewall, results in the firewall establishing a connection to a particular host/port. Once the two connections have been established, application level traffic can be sent from source to destination via the firewall. From a ORB perspective, GIOP messages will travel through the firewall uninterrupted i.e. ORB protocols are inconsequential to a TCP firewall.

The firewall can determine access control information from looking at the source address field in the TCP header, and make a decision as to whether that source host can connect through to the destination. A TCP firewall must have prior knowledge, and conceptually has a configuration table containing tuples of the form:(<inhost, inport>, <outhost, outport>). When a connection request from <inhost, inport> is received, assuming the firewall allows connections from that particular client, a connection is set up to <outhost, outport>.

Since a TCP firewall performs static mappings, this highlights a particular problem. If the outermost outbound firewall is a TCP firewall, and there is also an outermost inbound firewall, the outbound firewall must know this *a priori* since it cannot determine this information from the TCP header packets. Thus TCP firewalls are, in general, not suited to being placed as the outermost outbound proxy, unless a fixed configuration can be assumed, such as may be the case in intranet or extranet environments.

A very simple form of ORB interoperability through TCP firewalls can be achieved without any additions to CORBA. Assuming a server is in an enclave protected by a TCP firewall, the server can be configured to know about this firewall and may substitute the host and port address of the server with the host and port address of the firewall in any IORs issued outside the enclave (how this is done is an implementation issue for the ORB vendor). Hence a client outside the enclave will receive an IOR that contains the address of the firewall and not the server. The client will therefore send GIOP messages to the firewall (which are forwarded to the server) thinking that the object is actually on the firewall. This scheme can be used independently from the other mechanisms described in this chapter, since it is completely transparent to clients. Often TCP firewalls are used in more complex configurations, where it is not feasible to use this scheme. In these cases the mechanisms described in this chapter can be used.

Since traditionally TCP/IP used a port per service, it is now common for TCP services to be identified by the port number used for the server. For example, SMTP mail is delivered on port 25, X11 traffic on port 6000, etc. As a result, most existing firewalls base their low-level access control decisions on the port used, and due to this ORB interoperability through TCP firewalls is impeded as there is no well-known “IIOP port”. We define a recommended “well-known IIOP port” and a “well-known IIOP/SSL port”. Client enclaves with TCP firewalls will then be able to permit access to IIOP servers by enabling access to this port through their firewall.

These ports are not mandatory, and IIOP servers can be set up to offer service through other ports if that is desired. However the ports serve as a basic guideline for server or TCP, SOCKS or GIOP proxy deployment, and allow client enclaves to immediately identify or filter the traffic as IIOP without requiring protocol analysis.

The well-known IIOP port is xx, and the well-known IIOP/SSL port is xx.

---

Issue – OMG needs to assign these numbers (or ranges of numbers?)

---

## 4.6 SOCKS

The SOCKS<sup>1</sup> protocol is an open Internet standard (IETF RFC1928) for performing network proxying at the transport layer. SOCKS creates a proxy which serves as a data channel between a TCP or UDP based client and server. The proxy between the client and server created by SOCKS is transparent to either party, which keeps to a minimum the required modifications to the existing applications when incorporating SOCKS proxy servers. SOCKS supports negotiation of authentication methods and can accommodate various security policies. The most popular application of SOCKS is as a circuit level network firewall, although it is more flexible and generic than a typical network firewall. For example, a SOCKS firewall could understand application layer protocols, including IIOP. A SOCKS firewall vendor may provide a firewall that understand IIOP without any additions to this specification - making a SOCKS firewall IIOP aware is transparent to ORB interoperability.

Figure 4-3 shows the flow of events in the SOCKS protocol. When the client needs to connect to the application server, it sends a message to the SOCKS proxy server to establish a connection between the client and the SOCKS server. The initial message that the client sends to the SOCKS server contains the authentication methods that the client supports. These can include any GSS-API compliant authentication methods, such as User/Password, Kerberos, or SSL. The SOCKS proxy server examines the methods and selects an appropriate one that the SOCKS proxy server supports too. The client and SOCKS proxy server then enter a method-specific sub-negotiation for the purpose of the authentication.

1. The IETF has standardized version 5 of the SOCKS protocol. The standard way to refer to this specific version of the protocol is "SOCKSv5". Within this document, we refer to SOCKSv5 simply as "SOCKS" as a matter of convenience.



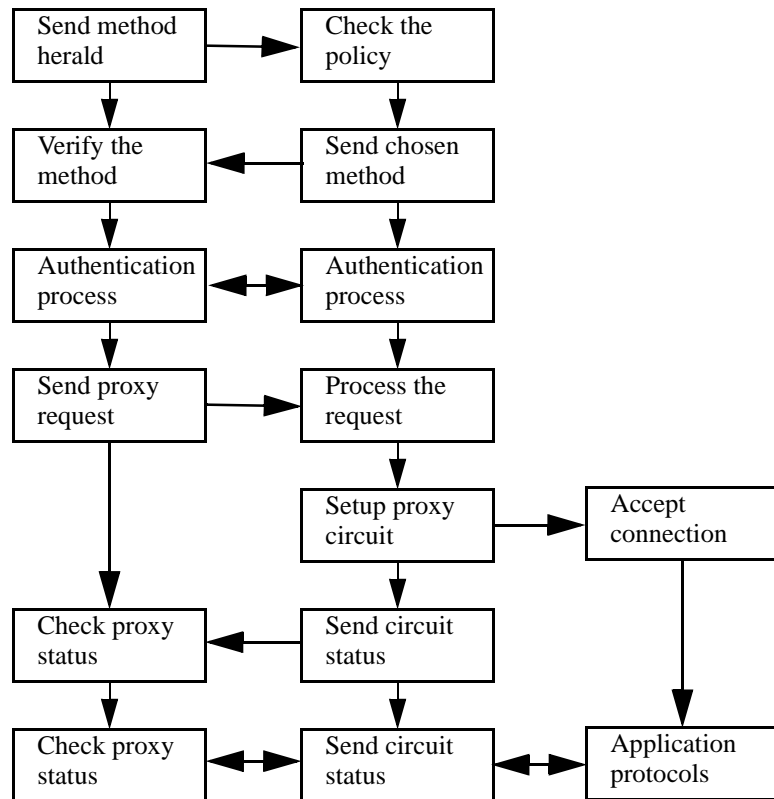


Figure 4-3 The SOCKSv5 Protocol

Upon the successful completion of the authentication process, the client sends the request details to the SOCKS proxy server i.e. the server it wants to connect to. If the negotiated method includes encapsulation for the purposes of integrity and/or confidentiality, the requests must be encapsulated in the method-dependent encapsulation. The SOCKS proxy server then establishes a connection to the application server on behalf of the client, if the application server is accessible from the client according to the configuration data at the SOCKS proxy server. Once the connection from the client to the application server via the SOCKS server is established, the client may now start passing application data to the SOCKS server, which in turn relays the data to the application server. If the selected authentication method supports encapsulation for the purposes of integrity and/or confidentiality, the data encapsulated using the method-dependent encapsulation, from the client to the SOCKS server and from the SOCKS server to the application server.

By default, a SOCKSified TCP or UDP based client communicates to a SOCKS proxy server over port 1080. Note that this port has been reserved by the IETF specifically for this purpose. As such, most existing firewalls enable clients to connect to a SOCKS

proxy server using this port. The following figure depicts a typical scenario of a TCP or UDP based client communicating across a firewall to an application server, using SOCKS.

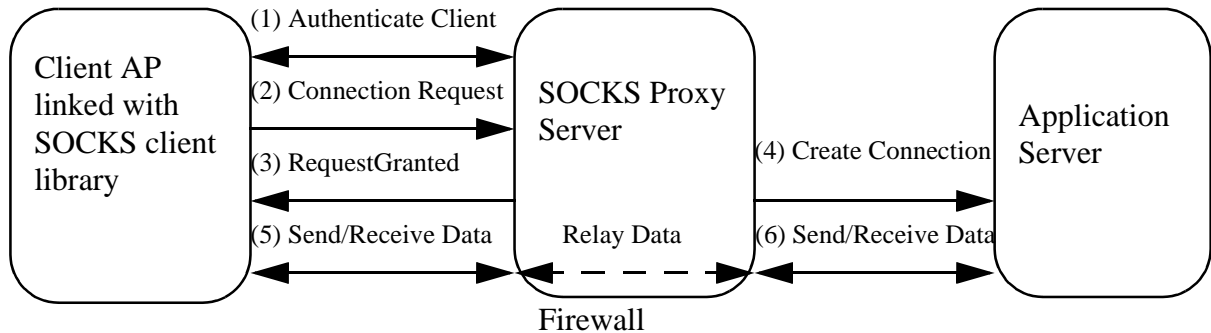


Figure 4-4 SOCKS in a Typical Firewall Traversal Scenario

The figure above depicts the very simple case in which there is exactly one firewall (protecting either outbound traffic from the client, or inbound traffic to the application server). In general, though, there may be any number of firewalls between the client and application server. As such, SOCKS also supports authenticated traversal of multiple proxy servers. As illustrated in the following figure, each connection between the client and a SOCKS server, between two SOCKS servers, and between a SOCKS server and the application server can be authenticated progressively using SOCKS, starting from the client. A virtual private connection can then be created between the client and application server. If SSL is deployed, the client's certificates can be passed through the connections to allow SOCKS servers and the application server to authenticate the client directly.

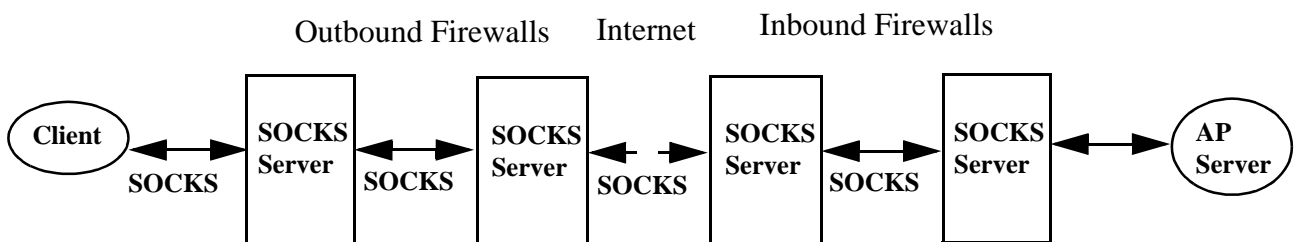


Figure 4-5 Multiple SOCKS servers between client and server applications.

SOCKS provides a flexible and extensible circuit level mechanism for generic proxy server construction. Various authentication/encryption methods, including SSL, can be independently deployed. SOCKS permits transparent traversal through multiple proxy servers. Access control based on IP address information and user information is also supported by SOCKS transparently, with access information put in the configuration files associated with the client and SOCKS proxy servers. Since SOCKS servers relay application data, it is possible to extend a SOCKS proxy with various network traffic screening and filtering capabilities i.e. for it to also act as an application level firewall.

From the perspective of SOCKS, IIOP is simply an example of a TCP-based application protocol. As such, SOCKS is already capable of serving as a proxy mechanism for IIOP, enabling IIOP traffic to traverse firewalls. Thus, to handle the simple case of a CORBA client invoking an operation on a CORBA object across a firewall (a special case of Figure 4-4), the only requirements are that the CORBA client must be linked with a SOCKSified TCP library, and that the firewall must support SOCKS (which most existing firewalls do). Such a scenario is depicted below:

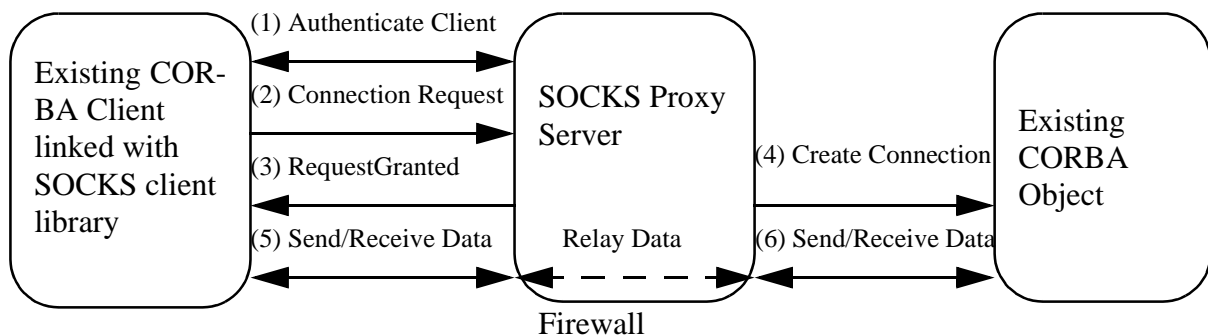


Figure 4-6 Existing CORBA AP Traversing Firewall with SOCKS

A “SOCKSified” TCP library provides an identical API for sending TCP/UDP traffic, and it reimplements these functions to interact with the SOCKS firewall. Therefore to use SOCKS, application code still calls the TCP/UDP API to open up a connection to the destination it requires. No source code changes are required to use SOCKS, it only has to be relented with the “SOCKSified” TCP/UDP library. In order to enable the configuration depicted above, no source code changes to an existing CORBA application, or CORBA 2 compliant ORB, are necessary. The only modifications to the existing environment that are necessary is that the client application must be re-linked with a SOCKSified TCP client library, and the client host must be configured to route SOCKS requests to the appropriate proxy server. The latter is controlled by client-side configuration files.

## 4.7 GIOP Proxy

A GIOP Proxy is an application level firewall that understands GIOP messages and the specific transport level inter-ORB Protocol supported i.e. a TCP GIOP Proxy understands IIOP messages. If more transport mappings of GIOP are standardized, a GIOP proxy supporting that protocol must understand those inter-ORB messages.

A GIOP Proxy firewall, or just GIOP Proxy for short, relays GIOP messages between clients and Objects. It may base access control decisions on information in the GIOP packet. For example, it could block requests to an object with a particular object\_key, or it could block requests for a particular operation on an object.

A GIOP Proxy hosts a GIOP Proxy Object. This is a fully fledged CORBA Object which provides operations for firewall navigation. Note that this does not require a full ORB to be implemented in the firewall, as long as the Object behaves in a way that is consistent with the semantics of a CORBA Object, and it understands the GIOP protocol and a transport mapping (such as IIOP).

To establish a connection to a server, a client first set up a connection to the GIOP Proxy. If the GIOP Proxy is an outbound one, the ORB should be configured with the IOR of the proxy object. If the GIOP Proxy is an inbound one, the server's IOR should contain the IOR of the proxy object on the firewall. After a connection is established, the client interacts with the proxy object to establish a connection to the target server. The interaction(s) required with a proxy may be dependent on the transport mapping. IIOP 1.0 and 1.1 clients interact with a proxy in one way, while IIOP 1.2 clients interact in a different way. This is explained in more detail below. Irrespective of how the client interacted with the proxy, and assuming appropriate permissions, the proxy will establish a connection with the server. Once this is done, the client and server may send GIOP messages to each other, according to the normal GIOP rules.

### 4.7.1 Connection styles

There are two styles of connection through a GIOP Proxy: *normal* and *passthrough*.

A normal connection is where, from a GIOP perspective, the firewall terminates each connection. From a client perspective, the firewall behaves like a server, and from a server perspective the firewall behaves like a client. Whenever a proxy blocks a message it must behave in a manner consistent with GIOP and CORBA semantics. For example, if a request is blocked by the proxy, and the client expects a reply, the proxy must send a reply (probably with a NO\_PERMISSION exception). It is the firewall's job to ensure that both connections maintain orderly GIOP dialogues, such that neither the client nor the server are aware that the proxy is involved.

In a normal connection, a proxy can monitor the GIOP traffic. This gives rise to two security issues. Firstly the client may not trust a GIOP proxy, and hence would not want the proxy to examine the traffic. Secondly, the client and server may be using a particular authentication and/or encryption mechanism that is unknown to the proxy. Both of these cases can be solved by the concept of a *passthrough* connection. A passthrough connection is one where the GIOP Proxy does not terminate the connections (at the GIOP level), it simply forwards on all GIOP messages it receives to

the appropriate party. This recognizes that either the proxy is not capable or is not allowed to examine the traffic. In a pass-through connection, the firewall is not responsible for maintaining the GIOP dialogue on the connection, and it may not issue any GIOP messages of its own (such as replies or close connection). Pass-through connections exhibit similar behavior to a transport level firewall, but on an object level i.e. once the proxy permits access to a particular object any traffic (following the rules of GIOP interactions) may flow uninterrupted through the proxy.

A GIOP Proxy has to support the capabilities of normal and passthrough connections. However, in a particular deployment, a GIOP Proxy may reject requests to establish pass-through connections because of prevailing security policies. It should always be possible, assuming access is permitted, to establish a normal connection through a GIOP Proxy.

### 4.7.2 *Callbacks*

In many cases, it is desirable for a CORBA-based application server to contact a client in order to facilitate asynchronous information flow. Such a pattern involves the client creating an object, and passing the reference to that object to the server as a parameter in a operation. Unfortunately this poses problems when firewalls are present, since it is common that outbound firewalls will not allow inbound connections to be made. In these cases it is possible to treat the callback object in exactly the same way as a fully fledged server and create an IOR for the callback object that contains information about inbound firewalls of the callback object. The server can then establish a connection to the callback object through the servers outbound firewalls, and the callbacks inbound ones.

This mechanism is however not possible for client side ORBs that can't generate IORs with local inbound firewall components for callback objects in the client space. Usually these are dynamically created untrusted objects that either can't or are not allowed to use the local inbound firewall information. For example, a callback object created in a Java applet downloaded via the browser neither has the knowledge about the inbound firewalls nor is allowed to accept the inbound connection by the inbound firewalls. Hence the IOR will not contain the appropriate firewall component information. In the absence of any mechanisms, invocations on such a callback object will be blocked by (at least) the client side firewall.

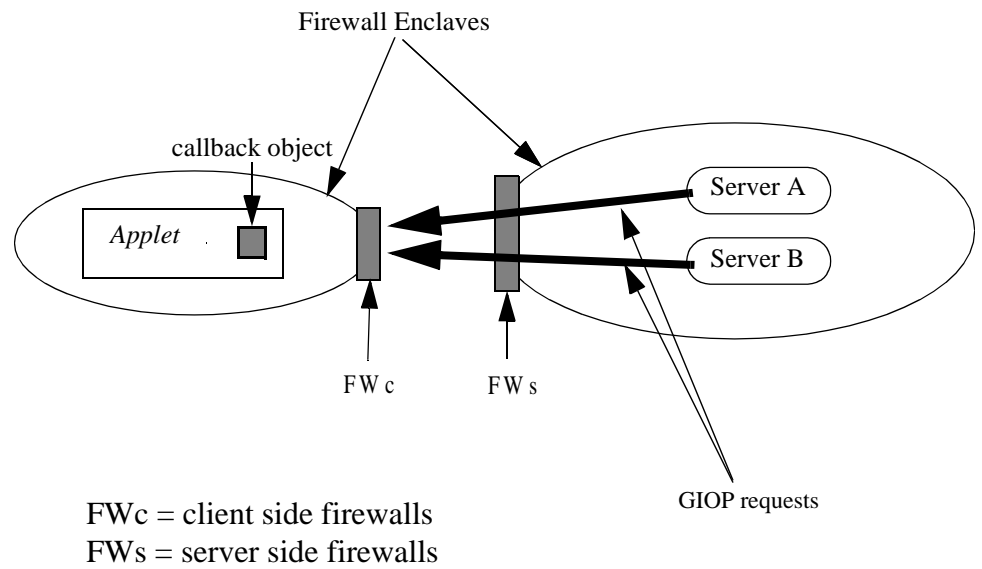


Figure 4-7 Client side firewall blocks requests from servers.

Figure 4-7 illustrates this, by showing that GIOP requests sent by servers will be blocked by the client side firewall.

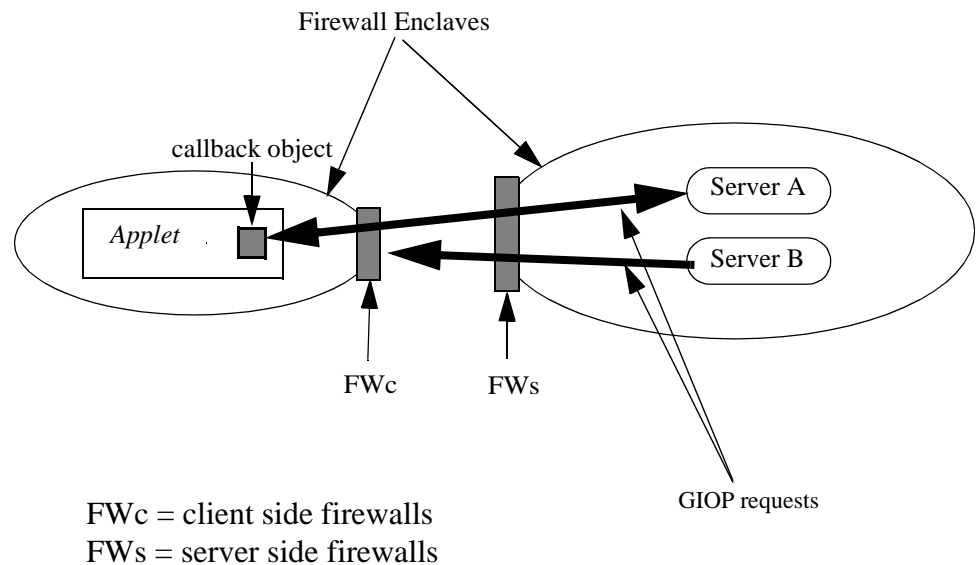


Figure 4-8 Client side firewall accepts GIOP requests using bi-directional GIOP.

Bi-directional GIOP can be used to partially solve the problem by permitting a server to reuse the client's connection to send GIOP request messages. This is only a partial solution since if the server that needs to perform callbacks is on a different host, it must establish a new connection (which is prohibited in the java applet security model, for example). Figure 4-8 illustrates this. Here it is assumed the client has a connection to Server A, and provided both sides permit the use of bi-directional GIOP, GIOP requests from A can be sent on the connection, and will hence pass through the client side firewall. Server B, on the other hand, cannot callback to the client even if the Server B has the object reference of the callback object. Since the IOR that B has will probably not contain firewall components, B cannot even use the normal mechanisms to navigate firewalls.

To provide a more generic solution in addition to the above, GIOP Proxy objects provide an operation that a client may call. The proxy will generate an IOR with appropriate firewall information in it, that can then be exported to the server. The server can establish a connection to the GIOP Proxy, and send traffic on it. The proxy will re-use the connection it already has with the client in a bi-directional mode to send the GIOP messages to the client. The behavior of this is fully defined below.

### 4.7.3 IIOP/SSL considerations

GIOP Proxy firewalls that forward IIOP requests also need to support the use of SSL as a transport mechanism for secure invocations, since ORB interoperability could be based on IIOP/SSL. There also needs to be the same level of access control available to the proxy administrator regarding permitted users and permitted targets.

The desired features are:

- Client and server side authentication for proxified connections.
- Access to client and server X.509 certificates.
- Access control to proxies.

A requirement for SSL support is that the certificate of the client be accessible at each link of the proxy chain, and at the server. Another requirement is that it be possible for each proxy along the chain (or at least each inbound proxy) to impose its access policy on the traffic passed through it. Since SSL was not designed with this kind of proxying in mind, and in fact protects against it as an ostensible "man in the middle" attack, a higher-level solution is defined.

Proxies that can support SSL connections fall into two categories, trusted and untrusted.

Untrusted proxies can forward information from a client in the form of a pass-through connection, i.e. the proxy has no visibility of the encrypted byte stream. This ensures the integrity of the client and server communication but leaves little opportunity for access control. This type of connection restricts the proxy's ability to apply its access control list fully, but it is necessary when either the server or client do not fully trust the proxy.

Trusted proxies can forward connections using a pass-through connection but also can establish separate connections to the server and provide full access control. This allows the implementation of access control either at the server as in the untrusted case or at the proxy at a per operation basis. All trusted proxies belong to a trust group decided by the target servers.

Since all proxies will have access to the IOR of the target object, and the certificate of the client, they can judge whether this client may use a pass-through connection or not. Whether or not a proxy allows or denies permission for a client to use pass-through in any given circumstance is up to the proxy's implementor.

### 4.7.4 GIOP Proxy Interface

All GIOP firewalls must support objects of the following type:



```
// IDL

module Firewall {

    enum ProxyMode { NORMAL, PASSTHRU };

    interface GIOPProxy {
        Object new_target(in Object target, in ProxyMode mode);
        Object new_callback(in Object callback);
    };
};
```

### *new\_target*

The **new\_target** operation informs the firewall that it should prepare itself to receive requests destined for the specified target. The object returned from this operation is the destination on the firewall that a request on the target should be sent to i.e. the **object\_key** in the return object should be used in the GIOP request header.

The mode argument indicates how a client is going to connect to the object. A mode value of **NORMAL** indicates that the proxy will act as an endpoint for GIOP traffic. This allows the proxy to examine the GIOP traffic as it flows through the firewall and potentially apply access control on individual requests. A mode of **PASSTHRU** indicates that the proxy will not be an endpoint of GIOP traffic and is not able to examine the traffic once the connection to the object is established. See Section 4.7.1 for more details.

If the firewall supports GIOP 1.2 and requires a **NORMAL** connection, it is not necessary to invoke the **new\_target** operation. Instead the sender (assuming it knows GIOP 1.2) can place the target IOR in the **targetAddress** field of the **GIOPRequestHeader1\_2**. However if a **PASSTHRU** connection using GIOP 1.2 is required, **new\_target** must be invoked, and the resulting object should be used as the destination of the GIOP 1.2 messages.

Note that in the **NORMAL** case, IIOP over SSL can be used although separate IIOP/SSL connections will be established between the client and the firewall, and the firewall and the target.

The object returned by the **new\_target** operation must contain an object key which allows the proxy to uniquely identify the target. A client is not required to open a new connection to the proxy server, even when the target object(s) are located in different servers.

### *new\_callback*

The **new\_callback** operation is designed, optionally with the bi-directional GIOP, to create an proxy object inside the server firewall domain that is reachable by the different servers to accept callbacks from the different servers.

When the object adapter creates the object reference for the callback object, it may invoke the **new\_callback** operation on the outermost inbound GIOP Proxy on the server side and pass the callback object as the argument. The object returned from this operation - a proxy object on the GIOP Proxy - is the destination on the GIOP Proxy that a request to the callback object should be sent to. Essentially the IOR of this proxy object becomes the IOR of the callback object created by the client.

When the **new\_callback** operation is invoked, the client should usually have the bi-directional service context to inform the GIOP Proxy to reuse the connection to forward requests from GIOP Proxy to the client.

When the server wants to invoke the methods on the callback object, the request messages will be sent to the GIOP Proxy because the IOR of the callback object is the IOR of the proxy object on the GIOP Proxy. The GIOP Proxy should forward the request messages on the connection on which the **new\_callback** for this proxy object was received if the bi-directional IIOP is used.

In the case that there are firewalls between the outer most inbound GIOP Proxy and the servers, there are different ways to deal with depending on the GIOP Proxy implementation and firewall configuration.

1. If the server knows how to reach the outermost GIOP Proxy through the firewalls between them, these firewalls can be treated as outbound firewalls to the server.
2. If the GIOP Proxy knows how the server should reach itself through the firewalls between them, these firewalls can be treated as inbound firewalls to the GIOP Proxy
3. If there are more GIOP Proxies between the outer most inbound GIOP Proxy and the server, the outer most inbound GIOP Proxy can invoke **new\_callback** on the inner inbound GIOP Proxy to create a proxy object which is closer and directly reachable to the server.

Thus this issue is left to the GIOP Proxy implementation and firewall configuration.

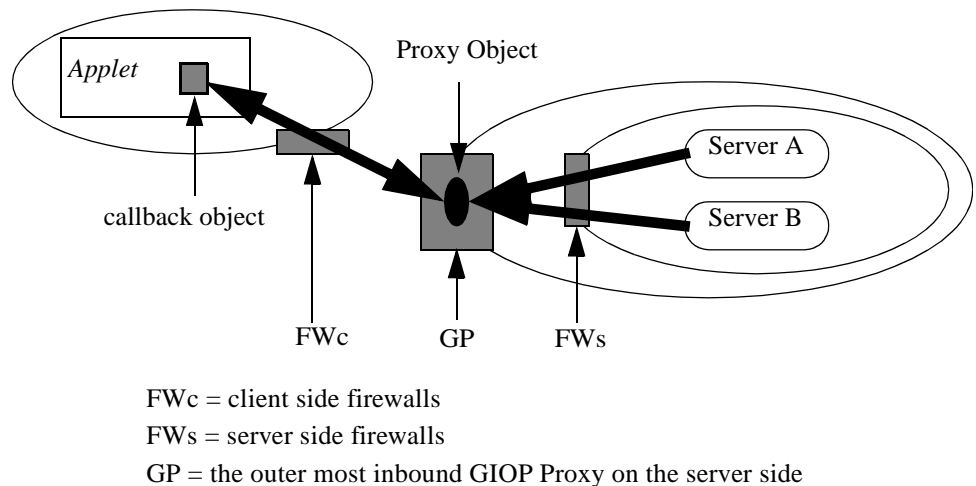


Figure 4-9 GIOP Proxy that permits different server to use callbacks.

Figure 4-9 illustrates use of `new_callback`. When the object adapter in the Applet creates the object reference for the callback object, it should invoke the `new_callback` method on GP with a bi-directional GIOP service context. The `new_callback` method will return a proxy object on the GP. When Server A or Server B invokes the methods on the callback object, the Servers will make connections to GP through FWs since FWs is configured as outbound firewalls to the Servers. The requests the Servers send will reach the Applet's callback object through the proxy object on the GP, reusing the connection the client has with GP.

## 4.8 Firewall tag components

An IOR contains information about the target address of an Object, such as a host/port pair. In order to traverse a firewall, an IOR must contain access information about the inbound firewalls. In a configuration where there are multiple enclaves (firewalls within firewalls) it may be necessary to carry access information for all inbound firewalls, although strictly it is only necessary to convey information on the outermost inbound firewall. To include firewall information in an IOR the following tagged component is defined.

```

module Firewall {

    const IOP::ComponentId TAG_FIREWALL_TRANS = xx; // OMG-
    allocated

    //TAG_FIREWALL_TRANS is a sequence of FirewallMechanism
    sequence <FirewallMechanism> multipleFirewallMechanisms;

    typedef unsigned long FirewallProfileId;

    struct FirewallMechanism {
        FirewallProfileId tag;
        sequence<octet> profile_data;
    };

}; //end module Firewall

```

The **TAG\_FIREWALL\_TRANS** component can appear zero or more times in an IOR profile. It is optionally present and may not be dropped. Each **TAG\_FIREWALL\_TRANS** component represents a single entry point into the target's network. Multiple firewall components indicate that there are multiple entry points into the target's network, any one of which can be used to reach the target.

A **TAG\_FIREWALL\_TRANS** component is encoded as an encapsulated sequence of **FirewallMechanism** structures. The sequence of firewall mechanisms describes the chain of publicly known inbound firewalls that need to be traversed to reach the target object. The order of **FirewallMechanism** in the Firewall Component sequence is important. The sequence dictates the order of traversal necessary to reach the target. The first **FirewallMechanism** in the sequence indicates the furthest publicly known firewall from the target (i.e. an initial entry point) and the last in the sequence represents the closest publicly known firewall to the target object. At least one **FirewallMechanism**, representing an initial entry point, must be present. It is optional, and up to designers and administrators as to whether the full sequence of inbound firewalls are recorded or just the initial entry point firewall.

Each firewall mechanism contains a **FirewallProfileId** and firewall profile data of the structure defined by that type. A firewall profile is defined in terms of the type of firewall supported. Currently three firewall profiles are defined to support SOCKS firewalls, TCP level firewalls, and GIOP proxy firewalls.

The **FirewallProfileId** tag is a numeric identifier used to indicate the type of profile and the encoding of the profile data of that type. These numbers are maintained by the OMG. Each registered firewall profile must have an assigned tag, and must define how the profile data is encoded.

### *Support for IIOP 1.0 IORs*

An IIOP 1.0 IOR cannot contain tagged components in a `TAG_INTERNET_IOP` profile. In cases where firewall information should be carried with the IOR, the IOR should contain a `TAG_INTERNET_IOP` profile describing the target object and a `TAG_MULTIPLE_COMPONENTS` profile that contains firewall components.

### *GIOP Proxy Firewall Tag*

A GIOP proxy firewall tag contains an IOR of a **Firewall::GIOPProxy** object. It is defined as:

```
//IDL
// in module Firewall

const FirewallProfileId FW_MECH_PROXY = 0;

typedef GIOPProxy ProxyFirewallMechanism;
```

The `profile_data` field of a **FW\_MECH\_PROXY** is a CDR encapsulation of a GIOPProxy IOR.

An IOR for a GIOPProxy object must not contain any **TAG\_FIREWALL\_TRANS** components.

### *TCP Firewall Tag*

A TCP firewall tag is defined as:

```
//IDL
// in module Firewall

const FirewallProfileId FW_MECH_TCP = 1;

struct TcpFirewallMechanism {
    string host;
    unsigned short port;
    sequence<IOP::TaggedComponent> components;
};
```

The `profile_data` field of a **FW\_MECH\_TCP** should be encoded as a CDR encapsulation of a **TcpFirewallMechanism** struct.

Currently the only valid component that may be placed in the `components` field in the **TcpFirewallMechanism** struct is an SSL tag - all other tagged components should be ignored.

### *Socks Firewall Tag*

A socks firewall tag is defined as:

```
// IDL
//in module Firewall

const FirewallProfileId FW_MECH_SOCKSV5 = 2;

typedef unsigned short AuthMethodType;

const AuthMethodType NONE = 0;
const AuthMethodType GSSAPI_KRB5 = 1;
const AuthMethodType PASSWORD = 2;
const AuthMethodType CHAP = 3;
const AuthMethodType SSL = 4;
const AuthMethodType CRAM = 5;

struct AuthSchemeKeyDistributor {
    string key_distribution_host;
    unsigned short key_distribution_port;
};

struct AuthenticationScheme {
    AuthMethodType auth_type;
    sequence<octet> auth_kdc;
};

struct SOCKSV5FirewallMechanism {
    string host;
    unsigned short port;
    AuthenticationScheme method;
};
```

The **profile\_data** field of a **FW\_MECH\_SOCKSV5** should be encoded as a CDR encapsulation of a **SOCKSV5FirewallMechanism** struct. The host and port fields should contain the host address and port number of the SOCKS proxy server to which the client should connect using the SOCKS V5 protocol. Typical SOCKS V5 implementations rely on local configuration information to enable the client to select an appropriate authentication method to use when connecting to the proxy server. The **AuthenticationScheme** field is supplied to account for situations in which the client has no configuration information pertaining to the target proxy server. It is defined as a data structure that has two members: one which informs the client which authentication method should be used when connecting to the proxy server, and another, the **auth\_kdc** member, which is provided to contain information about the key distribution authority which should be used when obtaining the necessary authentication and/or encryption keys.

The information contained in the **auth\_kdc** member will be specific to the authentication method being used. Typically, it will either contain an encapsulated **AuthSchemeKeyDistributor** structure containing the hostname and port number for the key distribution authority, or else it will be a null sequence when the authentication method in use requires no key distribution authority. The semantics of the **AuthSchemeKeyDistributor** fields differ per authentication method.

Note that the defined authentication method constants, and their associated values, are the same as those defined in defined in the IETF draft-ietf-atf-socks-pro-v5-02<sup>2</sup>. Also note that information in a SOCKS firewall mechanism is not intended to be used directly by an ORB. It contains information relevant for the socks implementation and should be passed to that.

Since the SOCKS tag is intended for use for SOCKS and not for ORBS (at most the ORB passes the data to a SOCKS library), it is not necessary from an ORB interoperability perspective to define specific encoding for each type of **auth\_kdc** field, although they should be encoded as CDR encapsulations. If it is shown in the future that ORB interoperability is affected by not defining these encodings, they will have to be defined.

## 4.9 Firewall POA Policy

In order to take advantage of the tag component defined above, a server side ORB must contain configuration information about the firewalls in its domain. No interfaces for the setting or retrieving of firewall information in an ORB are defined as this is an implementation issue. However, it is desirable to provide a portable means by which the object implementor can decide whether an object could be accessible through a firewall. The following POA policy is defined for this purpose:

2.The exception to this is SSL, which hasn't yet been officially registered with the IETF. However, the value of 4 has been reserved for SSL, with the expectation that it will be officially submitted to the IETF in the near future.

```

//IDL
// In module Firewall

typedef unsigned short FirewallPolicyValue;
const FirewallPolicyValue EXPORT = 0;
const FirewallPolicyValue NO_EXPORT = 1;

const CORBA::PolicyType FIREWALL_POLICY_TYPE = xx;
                                // to be assigned by OMG

interface FirewallPolicy : CORBA::Policy {
    readonly attribute FirewallPolicyValue value;
};

```

The default value of a FirewallPolicy is **NO\_EXPORT**. When creating a POA with a firewall policy using the **PortableServer::POA::create\_POA**, it is possible that prevailing security policies may prevent any object from being exported beyond the firewall. In these cases the **create\_POA** operation may raise a **PortableServer::POA::InvalidPolicy** exception.

In the absence of a FirewallPolicy being passed in the **create\_POA** operation, a POA will assume a policy value of **NO\_EXPORT**.

To create a FirewallPolicy, the **ORB::create\_policy** operation is used.

## 4.10 Outbound firewalls

An IOR may contain information on the inbound firewalls that need to be traversed to reach the target object. Before an inbound firewall is reached, it may be necessary to traverse outbound firewalls. Information about outbound firewalls is configured into the client side ORB and each outbound firewall. It is only necessary for each client or outbound firewall to know about the next firewall it needs to send requests to i.e. it does not need to know the complete sequence of outbound proxies that need to be traversed. It is out of scope of this specification to define how the client ORB or a firewall is configured with outbound firewall information. It is worth pointing out that a client or an outbound firewall that forwards messages to another outbound firewall, does not use any of the firewall information that may be present in the target IOR. The firewall information in an IOR is only used by the client when there are no outbound firewalls to be traversed, or by the outermost outbound firewall. In either case they use the information in the IOR to determine what the first inbound firewall is.

## 4.11 Traversal algorithm

One of goals for ORB interoperability through firewalls is to allow a mixture of different firewall types to be used between clients and servers. Each firewall type has its own mechanisms for establishing connections through it. In order to traverse a sequence of firewalls of different types it is necessary to understand how they work in combinations. Currently three firewall types are defined (TCP, SOCKSv5, and GIOP Proxy). This section defines the rules necessary to traverse a sequence containing any



combination of these three types of firewalls. Any additional firewall type added must define any rules necessary for it to be used in combination with all the other defined firewall types.

It is assumed that the client is in possession of an IOR that contains firewall profiles. If it is not it can only attempt a regular IIOP request<sup>3</sup>.

A client will determine if it needs to go through a firewall to make a request on the target object. If the client is in the same domain a direct invocation can be made. The client can determine this by examining the host address information in the target IOR.

A client that determines it cannot make a direct invocation needs to traverse firewalls. If the configuration in the client provides information on an outbound firewall that must be traversed, the client will send the request to that firewall. If the client cannot determine an outbound firewall, it looks in the IOR and picks the first **FirewallMechanism** in the **TAG\_FIREWALL\_TRANS** field of any firewall component found in IOR.

Having determined which is the first firewall to traverse, the client will perform different behaviors depending on the type of firewall that needs to be traversed.

### *Traversing a GIOPProxy firewall*

To traverse a GIOPProxy firewall, a client opens up a connection to the object.

If the firewall and client supports GIOP 1.2 and requires a **NORMAL** connection, the client constructs a **GIOP\_REQUEST\_HEADER\_1\_2** that contains the full server IOR (including all firewall components) as the target address information. The request is sent to the object.

If either the firewall or the client do not understand GIOP 1.2, or a **PASSTHRU** connection is required, the client issues a **new\_target** passing over the complete IOR of the target. The operation returns an IOR, which contains information on the host, port and object\_key required to send a request to the firewall.

On receipt of a GIOP request, a GIOP proxy has to determine if the next hop is another firewall or the target. If it is the target, a connection is setup and the request is sent to the object. If the next hop is another firewall, the proxy follows the traversal rules described in this section (i.e. follows the same traversal algorithm as a client does).

### *Traversing a TCP firewall*

A TCP firewall is very simple. To traverse a single TCP firewall the client opens a connection to the host and port of the TCP firewall and sends data on the connection. The TCP firewall will then forward on the traffic to the host/port defined in its configuration tables. However there is a problem if the firewall after a TCP one is a SOCKS firewall. The problem stems from the fact that to traverse a SOCKS firewall

3. Note that a TCP firewall can transparently be used by substituting the host/port information in the IOR with the host and port of the TCP firewall. This does not affect the algorithm defined here. Note also that a GIOP Proxy can also transparently reside behind a TCP firewall using the same mechanism.

the socks protocol needs to be used to establish a connection to it (see Section 4.6). However a TCP firewall cannot perform a SOCKS setup, since it is essentially dumb. Therefore the client will have to do the SOCKS setup, via the TCP firewall. The problem gets more complicated if along the chain there are more SOCKS firewalls after TCP ones. Essentially the client has to progressively build up a connection to the various SOCKS Firewalls on the route.

To traverse a TCP firewall you have to determine if the firewall after the TCP firewall is a SOCKS one or not. If it is this algorithm should be followed:

1. set the host/port of the TCP firewall as the SOCKS server in the clients SOCKS library i.e. configure the local SOCKS library to think that the TCP firewall is the SOCKS proxy server.
2. find the first non-SOCKS firewall or the target after the SOCKS firewall.
3. open the TCP connection to the non-SOCKS firewall or the target
4. if the non-SOCKS firewall is another TCP firewall, the client has to repeat the above process

The above algorithm sets up a “virtual circuit”, from a client to either a GIOP Proxy or the target object. The virtual circuit may traverse a mixture of SOCKS and TCP firewalls.

If the circuit terminates at the target object, the client can use the connection to send required GIOP requests. Otherwise the client traverses the GIOPProxy firewall as described above.

### *Traversing a SOCKS Firewall*

To traverse a SOCKS firewall the client needs to follow the following algorithm:

1. if necessary, configure the SOCKS library with the host/port of the first socks firewall
2. determine the first non-SOCKS TCP endpoint. This will either be a TCP firewall, a GIOP Proxy or the target.
3. if the first TCP endpoint is the target, open a TCP connection to the target host/port and send the GIOP requests. Underneath the covers the sock library will connect to the socks firewall allowing traffic to flow through to the destination.
4. if the first TCP endpoint is a GIOP Proxy, either invoke `new_target` or send a `GIOPRequestHeader1_2` (described above, traversing a GIOP Proxy firewall). Underneath the covers the sock library will connect to the socks firewall allowing traffic to flow through to the GIOPProxy.
5. if the first TCP endpoint is a TCP firewall, it has to be traversed using the algorithm described above for traversing TCP firewalls i.e. you have to set up a virtual circuit to the target or the next GIOP Proxy.

## 4.12 *Passing credentials*

Typically, a secure connection, such as IIOP/SSL, will use a passthrough connection to achieve end-to-end authentication. However passthrough connections may only be accepted if they are permitted by the proxy's security policies.

In the event that the proxy's configuration does not specify that the client is permitted to use a pass-through connection to the object it specified, the proxy returns a **NO\_PERMISSION** exception. This allows the client to fall back to using **new\_target** with a mode of **NORMAL**.

When a client establishes a normal connection to a target via a trusted proxy and uses a secure transport (e.g. IIOP/SSL), in order to achieve end-to-end authentication, the proxy will have to forward the clients certificate/identity to the server. To achieve this, the following service context is defined.

```
// IDL
// in module Firewall

const IOP::ServiceId ForwardedIdentity=xyz; // OMG allocated

typedef unsigned short IdTag;           // OMG allocated
struct Identity {
    IdTag          tag;
    sequence<octet> data;
};

typedef sequence<Identity> IdentityList;
```

Each security mechanism that requires the passing of identities, must have an **IdTag** allocated, and must define for that **IdTag** how the **data** in the **Identity** structure is encoded. **IdTag** values are allocated by the OMG.

The **IdentityList** is a sequence because sometimes other information needs to be passed. For, example, the "role" of the client (the client could have several roles, but the same cert), and/or the extracted Distinguished Name and password, key, etc. if other identity types are defined.

A GIOPProxy inserts a **ForwardedIdentity** service context on the first GIOP request received from a client and forwarded on each new connection between the proxy and target. Additionally the context should be inserted when multiple clients are sharing the same connection from the proxy to the server, and the previous request message was from a different client.

Since the proxy inserts the service context and the proxy is also the point at which requests from different clients are multiplexed on the same server connection, it is only necessary to insert the service context when it would be ambiguous to the server from which client the request originated. This allows the server to cache the most recent

identity and thus reduces the need to pass the identity in every request. Note that it is valid to have the **ForwardedIdentity** service context in every request, although this may incur a performance and message size penalty.

The following **IdTags** are currently defined: **TAG\_ID\_SSL\_CERT**.

**TAG\_ID\_SSL\_CERT** has the value of 0.

#### 4.12.1 *SSL Certificates*

To pass SSL certificates in a **Firewall::ForwardedIdentity** service context, the following is defined:

**// IDL**

**module SSLIOP {**

**const Firewall::IdTag TAG\_ID\_SSL\_CERT = 0; // OMG allocated**

**typedef sequence<octet> ASN\_1\_Cert;**

**typedef sequence<ASN\_1\_Cert> SSL\_Cert;**

**};**

The **data** field of a **TAG\_ID\_SSL\_CERT** should be encoded as a CDR encapsulation of a **SSL\_Cert**.

An **SSL\_Cert** is a sequence (chain) of X.509 certificates, ordered with the sender's certificate first followed by any certificate authority certificates proceeding sequentially upward. An **ASN\_1\_CERT** is encoded using DER.

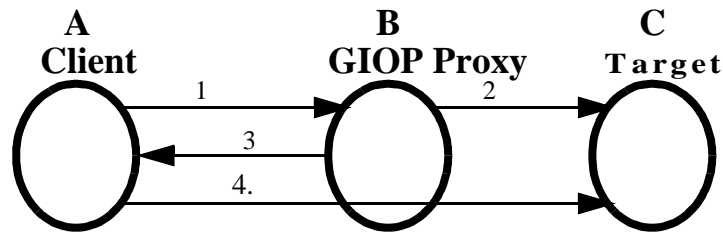
Note: The certificates are not encoded as PKCS #6 extended certificates; they are X.509 certificates as defined by the ITU-T and IETF. Furthermore, the chain of certificates is not a SET of **ExtendedCertificateOrCertificate** as defined by PKCS #7; it is a sequence as defined by the CORBA CDR encapsulation rules.

### 4.13 *IIOP/SSL Considerations*

Establishing IIOP/SSL connections through GIOP Proxies requires some refinements of the mechanisms defined above. These are defined by examples below.

#### *Untrusted Proxies*

The following is a run through of the pass-through connection case for untrusted proxies.



1. A invokes `new_target(C, PASSTHRU)` on B
2. B sets up a TCP connection to C
3. B replies to A
4. A initiates an SSL handshake with C via the proxy.

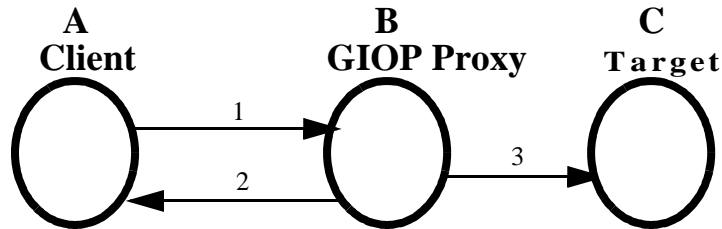
Figure 4-10 Pass-through Connection

Using Figure 4-10, client A connects to the untrusted proxy B using normal IIOP without SSL. The client issues a `new_target` using C as the target objects to connect to and **PASSTHRU** as the mode argument (1). At this point B can perform access control to allow/deny the pass-through connection to C. If a pass-through connection is allowed, B sets up a TCP connection to C (2), and then replies to A (3).

When A receives a successful reply from the `new_target` operation, A starts an SSL negotiation with C to complete the pass-through connection through B (4). B no longer has visibility/control over communication between A and C, although it may drop the pass-through connection at any time.

It is possible to support pass-through through multiple proxies. For example if in the above example there was another proxy B2 between B and C, during processing the `new_target` operation from A, B can try to establish a pass-through connection to C via a call to `new_target` on B2. If this fails, due to `NO_PERMISSION` for example, B should fall back to try to connect through B2 using the **NORMAL** mode.

### Trusted Proxies



1. A invokes `new_target (C, NORMAL)` on B over an IIOP/SSL connection
2. B replies to A
3. B negotiates an SSL connection with C

Figure 4-11 Normal Connection through a Trusted Proxy

Referring to Figure 4-11, if B is trusted, A may decide to establish a **NORMAL** connection through the proxy to the target, C. A invokes `new_target` on the proxy using a mode of **NORMAL** (1) using an IIOP/SSL connection, with authentication of A and/or B taking place. As a result, B can establish an SSL connection to C, with authentication of B and/or C taking place (4). The server C rejects the proxy's attempt to establish an SSL connection if it B does not belong to its trust group. When the two connections are established, client identity or a certificate of the client is made available to the server by **ForwardedIdentity** service context information in GIOP messages.

This chapter contains a proposal for bi-directional GIOP. Section 5.1 describes changes to the GIOP, and Section 5.2 describes a new POA policy to indicate that use of bi-directional GIOP is allowed.

### *5.1 Bi-Directional GIOP*

The specification of GIOP connection management in CORBA V2.0 requires that connections are not symmetrical e.g. only clients (which initialize connections) can send requests and only servers (which accept connections) can receive them.

This restriction gives rise to significant difficulties when operating across firewalls. It is common for firewalls not to allow incoming connections, except to certain well-known and carefully configured hosts, such as dedicated HTTP or FTP servers. For most CORBA-over-the-internet applications it is not practicable to require that all potential client firewalls install GIOP proxies to allow incoming connections, or that any entities receiving callbacks will require prior configuration of the firewall proxy.

An applet, for example, downloaded to a host inside such a firewall will be restricted in that it cannot receive requests from outside the firewall on any object it creates, as no host outside the firewall will be able to connect to the applet through the client's firewall, even though the applet in question would typically only expect callbacks from the server it initially registered with.

In order to circumvent this unnecessary restriction, it is proposed that the asymmetry stipulation above be relaxed in cases where the client and the server agree on it. In these cases, the client (the applet in the above case) would still initiate the connection to the server, but any requests from the server on an objects exported by the client to the server via this connection will be sent back to the client on this same connection.

The mechanism by which the client and server agree on this capability is as follows:

The client creates an object for exporting to a server.

The client exports the IOR as a parameter of an GIOP Request on the server object. If the ORB policy permits bi-directional use of a connection, a Request message should contain an **IOP::ServiceContext** structure in its Request header, which indicates that this GIOP connection is bi-directional. The service context may provide additional information that the server may need to invoke the callback object. To determine whether an ORB may support bi-directional GIOP a new POA policy has been defined ( Section 5.2).

Each mapping of GIOP to a particular transport should define a transport specific bi-directional service context, and have an **IOP::ServiceId** allocated by the OMG. It is recommended that names for this service context follows the pattern *BiDir<protocolname>*, where <protocol name> identifies a mapping of GIOP to a transport protocol, e.g. for IIOP the name is BiDirIIOP. The service context for bi-directional IIOP is defined below ( Section 5.1.1).

The server receives the Request. If it recognizes the service context and supports bi-directional connections, it may send invocations on this object back along the connection.

The server may not wish to support bi-directionality either due to lack of support for it, or because it has been configured that way. In this case, it may fall back to initiating a connection to the object in the usual way.

If a GIOP connection is used bi-directionally, the client should attempt to keep the connection alive as long as is necessary to complete its object's service to the server. If the client initiates a new connection it is not foreseen here that the server can use that connection for requests on the object exported previously.

A server talking to a client on a bi-directional GIOP connection can use any message type traditionally used by clients only, so it can use Request, LocateRequest, CancelRequest, MessageError and Fragment (for GIOP 1.1). Similarly the client can use message types traditionally used only by servers: Reply, LocateReply, MessageError, CloseConnection and Fragment.

CloseConnection messages are a special case however. Either ORB may send a CloseConnection message, but the conditions in section 13.5.1 of the CORBA v2.2 specification are modified in that the ORB sending the CloseConnection must not be awaiting Replies to any Requests, and must not have begun processing any Requests from the other side. If these conditions are satisfied and the ORB sends a CloseConnection, the ORB on the opposite side must assume that any outstanding Requests it has sent were not processed and may be resent on a new connection. The ORB which sends the CloseConnection must not send any messages after the CloseConnection. It may also close the connection, although the caveats regarding protocols which do not implement "orderly shutdown" in section 12.5.1 apply.

Bi-directional GIOP connections modify the behavior of Request IDs. In the GIOP specification, section 12.5.1 (Connection Management), it is noted that "Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection". With bi-directional IIOP, the Request ID unambiguously associates



replies with requests per connection *and per direction*, so the same Request ID can be used for a Request going from client-to-server and for a Request going from server-to-client, simultaneously.

It should be noted that a single-threaded ORB needs to perform event checking on the connection, in case a Request from the other endpoint arrives in the window between it sending its own Request and receiving the corresponding reply; otherwise a client and server could send Requests simultaneously, resulting in deadlock. If the client cannot support event checking, it must not indicate that bi-directionality is supported. If the server cannot support event checking, it must not make callbacks along the same connection even if the connection indicates it is supported.

A server making a callback to a client cannot specify its own bi-directional service context – only the client can announce the connection's bi-directionality.

### 5.1.1 Bi-Directional IIOP

The `IOP::ServiceContext` used to support bi-directional IIOP contains a `BiDirIIOPServiceContext` structure as defined below:

```
// IDL
module IOP {

    IOP::Serviceld    BiDirIIOP = xx; // to be assigned by the OMG

    struct ListenPoint {
        string host;
        unsigned short port;
    };

    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {
        ListenPointList listen_points;
    };
};
```

The data encapsulated in the `BiDirIIOPServiceContext` structure allows the ORB, which intends to open a new connection in order to invoke on an object, to look up its list of active client-initiated connections and examine the structures associated with them, if any. If a `host` and `port` pair in a connection's `listen_points` list matches that which the ORB intends to open a connection to, then the connection can be re-used to make the invocation.

The `host` element of the structure should contain whatever values the client may use in the IORs it creates. The rules for `host` and `port` are identical to the rules for the IIOP IOR ProfileBody\_1\_1 `host` and `port` elements; see section 13.7.2 of the CORBA 2.2

specification. Note that if the server wishes to make a callback connection to the client in the standard way, it must use the values from the client object's IOR, not the values from this **BiDirIIOP** structure; these values are only to be used for bi-directional GIOP support.

The **BiDirIIOP** service context may be sent by a client at any point in a connection's lifetime. The **listen\_points** specified therein must supplement any **listen\_points** already sent on the connection, rather than replacing the existing points. Typically, when the same client has multiple connections to the same server, the **listen\_points** will be identical. However, if they differ they supplement each other i.e. any of the listen points received on any of the connections may be used.

If a client supports a secure connection mechanism, such as SecIOP or IOP/SSL, and sends a **BiDirIIOP** service context over an insecure connection, the **host** and **port** endpoints listed in the **BiDirIIOP** should not contain the details of the secure connection mechanism if insecure callbacks to the client's secure objects would be a violation of the client's security policy.

If a client has not set up any mechanism for traditional-style callbacks using a listening socket, then the **port** entry in its IOR must be set to the outgoing connection's local port (as retrieved using the `getsockname()` sockets API call). The **port** in the **BiDirIIOP** structure must match this value. This will allow multiple clients, all running in restrictive security modes (such as Java applets) on the same host, all of them connecting to one server, to each receive callbacks on their correct connection.

### *IIOP/SSL considerations*

Bi-directional IOP can operate over IOP/SSL, without defining any additions to the IOP/SSL or the bi-directional GIOP mechanisms. However if the client wants to authenticate the server when the client receives a callback this cannot be done unless the client has already authenticated the server. This has to be performed during the initial SSL handshake. It is not possible to do this at any point after the initial handshake without establishing a new SSL connection (which defeats the purpose of the bi-directional connections).

## *5.2 Bi-directional GIOP policy*

In GIOP, there are strict rules on which side of a connection can issue what type of messages (for example clients can not issue GIOP reply messages). However, as documented in above, it is sensible to relax this restriction if the ORB supports this functionality and policies dictate that bi-directional connection are allowed. To indicate a bi-directional policy, the following is defined.

```

//IDL

module IIOP {

    // Bidirectional Policy

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = xx;
                                     // to be assigned by OMG

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };

};

```

A **BidirectionalPolicyValue** of **NORMAL** states that the usual GIOP restrictions of who can send what GIOP messages apply i.e. bi-directional connections are not allowed. A value of **BOTH** indicates that there is a relaxation in what party can issue what GIOP messages i.e. bi-directional connections are supported. The default value of a **BidirectionalPolicy** is **BOTH**.

In the absence of a **BidirectionalPolicy** being passed in the **PortableServer::POA::create\_POA** operation, a POA will assume a policy value of **BOTH**.

A client and a server ORB must each have a **BidirectionalPolicy** with a value of **BOTH** for bi-directional communication to take place.

To create a **BidirectionalPolicy**, the **ORB::create\_policy** operation is used.



### *6.1 Conformance*

There are two different compliance points defined in this specification.

In order to be conformant with the firewall specification (Chapter 4) the following is required:

- A server side ORB must be able to generate IORs that contain TAG\_FIREWALL\_TRANS components.
- A server side ORB must be able to generate TAG\_FIREWALL\_TRANS components that contain FW\_MECH\_PROXY firewall mechanisms (i.e. must be able to support GIOP proxies)
- A server side ORB may generate TAG\_FIREWALL\_TRANS components that contain FW\_MECH\_SOCKSV5 or FW\_MECH\_TCP i.e. support for SOCKS and TCP firewalls is optional.
- A server side ORB must implement FirewallPolicy objects.
- A client side ORB must be able to navigate GIOP proxy firewalls, and optionally, may be able to navigate SOCKSV5 and TCP firewalls.
- The forwarded identity service context must be supported in IIOP/SSL implementations.
- A GIOP Proxy firewall must implement the GIOPProxy interface.

In order to be conformant with the bi-directional GIOP specification (chapter 5), ORBS must support all of the specification. ORBS that support IIOP must implement the bi-directional specification.

## 6.2 *Changes to CORBA 2.2*

This section provides an overview of the changes to CORBA 2.2. as a result of this specification. Detailed editing instructions will be provided to OMG staff if the submission becomes adopted OMG technology.

Chapter 4 of this document, *ORB Interoperability through Firewalls*, with modifications indicated below, should be inserted as a new chapter in the CORBA specification. It should reside immediately after chapter 13, *General Inter-ORB Protocol*. The main change is the introduction of a new firewall module. The parts of Chapter 4 that should be placed elsewhere are:

- the SSL\_Cert forwarded identity definitions - Section 4.12.1 of this document - should be put in an appropriate place in the SSL section of CORBA Security.

Chapter 5 of this document, *Bi-directional GIOP*, should be edited into Chapter 13, General Inter-ORB Protocol.

This Appendix contains the IDL defined in this specification. Note that we hope the OMG will allocate the necessary TAG ids before adoption!

### A.1 Firewall Module

```
// IDL for Firewall Module: file "Firewall.idl"
#include "IOP.idl"

module Firewall {

    const IOP::ComponentId TAG_FIREWALL_TRANS = xx;

    //TAG_FIREWALL_TRANS is a sequence of FirewallMechanism

    typedef unsigned long FirewallProfileId;

    struct FirewallMechanism {
        FirewallProfileId tag;
        sequence<octet> profile_data;
    };

    typedef sequence <FirewallMechanism> multipleFirewallMechanisms;

    // Allocated FirewallProfileId tags.

    const FirewallProfileId FW_MECH_PROXY = 0;
    const FirewallProfileId FW_MECH_TCP = 1;
    const FirewallProfileId FW_MECH_SOCKSV5 = 2;
```

---

```

// definition of the GIOP Proxy interface

enum ProxyMode { NORMAL, PASSTHRU };

interface GIOPProxy {
    Object new_target(in Object target, in ProxyMode mode);
    Object new_callback(in Object callback);
};

// A FW_MECH_PROXY contains:

typedef GIOPProxy ProxyFirewallMechanism;

// A FW_MECH_TCP contains:

struct TcpFirewallMechanism {
    string host;
    unsigned short port;
    sequence<IOP::TaggedComponent> components;
};

// Definitions for SOCKSV5FirewallMechanism

typedef unsigned short AuthMethodType;

const AuthMethodType NONE = 0;
const AuthMethodType GSSAPI_KRB5 = 1;
const AuthMethodType PASSWORD = 2;
const AuthMethodType CHAP = 3;
const AuthMethodType SSL = 4;
const AuthMethodType CRAM = 5;

struct AuthSchemeKeyDistributor {
    string key_distribution_host;
    unsigned short key_distribution_port;
};

struct AuthenticationScheme {
    AuthMethodType auth_type;
    sequence<octet> auth_kdc;
};

// A FW_MECH_SOCKSV5 contains:

struct SOCKSV5FirewallMechanism {
    string host;
    unsigned short port;
    AuthenticationScheme method;
};

```



---

```

// Definitions for the ForwardedIdentity service context

const IOP::ServiceId ForwardedIdentity=xyz;// OMG allocated

// A forwardedIdentity contains a sequence of Identity structs

typedef unsigned short IdTag; // OMG allocated
struct Identity {
    IdTag          tag;
    sequence<octet> data;
};

typedef sequence<Identity> IdentityList;

// Firewall POA Policy

typedef unsigned short FirewallPolicyValue;
const FirewallPolicyValue EXPORT = 0;
const FirewallPolicyValue NO_EXPORT = 1;

const CORBA::PolicyType FIREWALL_POLICY_TYPE = xx;
                                // to be assigned by OMG

interface FirewallPolicy : CORBA::Policy {
    readonly attribute FirewallPolicyValue value;
};

}; //end module Firewall

```

## A.2 Additions to SSLIOP

```

// Additional IDL for SSLIOP module: file "SSLIOP.idl"
#include "Firewall.idl"

module SSLIOP {

    const Firewall::IdTag TAG_ID_SSL_CERT = xx;// OMG allocated

    typedef sequence<octet> ASN_1_Cert;
    typedef sequence<ASN_1_Cert> SSL_Cert;

};

```

---

### A.3 Additions to the IIOP Module

```
// Additional IDL for IIOP module: file "IIOP.idl"
#include "IOP.idl"

module IIOP {

    const IOP::ServiceIdBiDirIIOP = xx; // to be assigned by the OMG

    struct ListenPoint {
        string host;
        unsigned shortport;
    };

    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {
        ListenPointList listen_points;
    };

    // Bidirectional Policy

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = xx;
        // to be assigned by OMG

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```