CORBA Firewall Traversal Specification

This OMG document (ptc/2003-01-13) replaces the draft adopted specification. It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by March 3, 2003.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on May 12, 2003. You can find the latest version of a document from the Catalog of OMG Specifications at this URL: http://www.omg.org/technology/documents/spec_catalog.htm

OMG Final Adopted Specification
January 2003

CORBA Firewall Traversal Specification

Final Adopted Specification
ptc/03-01-13
January 2003
Based on: orbos/01-08-02 and errata orbos/01-10-06

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

The OMG documentation is organized as follows:

OMG Modeling Specifications

Includes the UML, MOF, XMI, and CWM specifications.

OMG Middleware Specifications

Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted parts of this specification:

- Borland Software Corporation
- IONA Technologies PLC
- Network Associates, Inc.
- Sun Microsystems, Inc.
- Xtradyne Technologies AG

CORBA Firewall Traversal 25

Note – From the OMG Editor: Eventually this specification will become part of CORBA Core. The chapter number is subject to change and is being used as a place holder for now.

Contents

This chapter contains the following sections.

25.1 Design Rationale and Background

25.1.1 Overview

The overall goal of this specification is to provide better accessibility to CORBA application servers when there is a firewall separating a client from a server. In this context, "better" means that client-firewall-server communication can be enabled and controlled more easily for a broader range of circumstances. Currently, ORBs and firewalls have a limited form of "peaceful co-existence" that provides satisfactory functionality only in some cases.

There are two reasons why CORBA poses a unique problem for firewalls: server location transparency and the peer-based communication model. One of the primary benefits of CORBA is that a client does not need to know the exact location of an object to invoke on it. Therefore the server can be relocated or the object can be provided by a new server without changing any client-side information. This location transparency introduces a problem when a firewall is used to protect the server, because normally a firewall provides protection by only allowing inbound connections to a small number of well-known addresses. Therefore, the server location must be known by the firewall *a priori*.

The CORBA peer-based communication model also causes a problem when communicating through firewalls. Traditional internet applications use a client-server model where the number of servers is relatively small, and the servers are under the administrative control of a single organization. In CORBA, any host could act as a client or a server, so the number of "servers" is relatively high and the "servers" are under the administrative control of multiple organizations. The problem with a large number of potential servers is two-fold. First, Network Address Translation (NAT)[1] is deployed by many organizations to extend the available address space and to hide the topology of the internal network from the outside. In the traditional client-server model, the servers use actual routable addresses (non-NAT) for the publicly available servers. But if a large number of hosts could potentially be servers, then this solution eliminates the effectiveness of NAT. Second, firewalls can easily be configured with a policy to control traffic to a small number of servers and deny access to all other hosts. If the number of servers becomes large, then the firewall policy becomes both unmanageable and decreasingly effective.

This specification identifies 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 the limitations. The need to overcome these limitations is the impetus for this specification.

Interoperable CORBA communication occurs via the GIOP protocol, which on the Internet is implemented by the IIOP protocol. Because firewalls control IP networking communication, and because ORBs communicate via IIOP, this specification is concerned with various aspects of how 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 new data elements to CORBA (for example, in IORs) that provide clients, firewalls, and servers the information needed for flexible, efficient, controlled firewall traversal. In fact, if CORBA servers

1. NAT allows an organization to use a private address space for internal use, and the firewall translates those addresses into globally unique addresses when clients communicate over the Internet. This allows many clients within an organization to share a few unique addresses.

use a single IP address that is routable by all of the clients, i.e., the address of the server is unambiguous within the clients' enclaves, then no additions need to be made to the CORBA specification for basic invocations on CORBA servers.

25.1.2 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 it will prevent it. Access through a firewall may be permitted at various levels of granularity. For example, access could be permitted to some objects behind the firewall based on network address, 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 subnet) 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's 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 can be accessed from within the enclave.

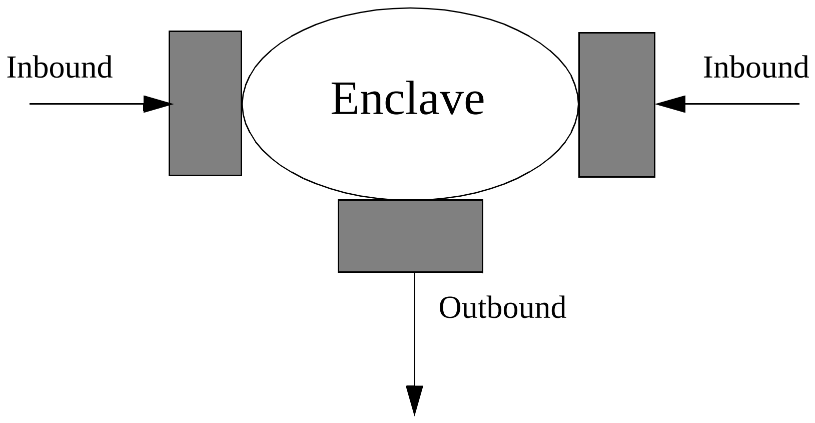


Figure 25-1 An enclave with multiple inbound and outbound firewalls

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 25-1 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.

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 the direction of the invocation. The outermost inbound firewall represents an entry point into an organization. Figure 25-2 illustrates a hierarchical nesting of enclaves. The outermost “Company XYZ” enclave contains two sub-enclaves, “Finance” and “R&D.” The “R&D” enclave further contains the “Research” enclave.

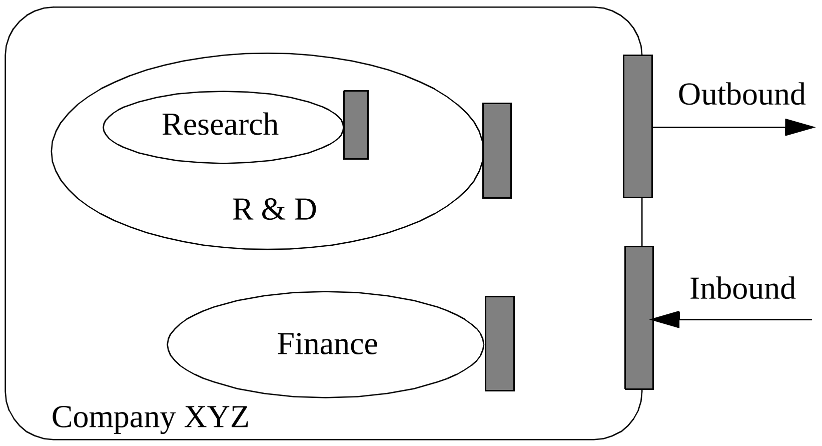


Figure 25-2 A hierarchical set of enclaves

25.1.3 *Types of Firewall*

Broadly, there are two types of firewall: transport level and application level. A transport level firewall allows resources to be accessed via any application level protocols. Such firewalls do not understand the type of application protocol being used, rather access is based purely on addressing information in the header of transport packets. Hence, access decisions are based on the source and destination of a message and not on the resource being accessed. Typically access control is performed during connection setup, and if the connection setup is successful, any application traffic may pass over the connection. A TCP firewall, for example allows access to FTP, HTTP, or IIOP resources, where access control is based on which hosts and ports traffic is travelling between.

Application level firewalls on the other hand are restricted to a particular application level protocol, such as IIOP or HTTP. As a result, access decisions can be based on both transport addressing information and on specific resources known to the application level protocol. For example, if there are two objects that can be accessed via 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.

25.1.3.1 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 coarsest 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 an 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 by 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 of the source to destination mappings, 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>.

A 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 of 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 TCP/IP services typically use a port per service, it is 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. ORB interoperability through TCP firewalls is currently impeded as there is no well-known IIOP port, therefore we define a recommended well-known IIOP port and a well-known IIOP/TLS 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 and firewall deployment, and allow client enclaves to immediately identify or filter the traffic as IIOP without requiring protocol analysis.

The well-known IIOP port is 683, and the well-known IIOP/TLS port is 684.

25.1.3.2 Application Proxy

An application proxy is an application level firewall that understands GIOP messages and the specific transport level inter-ORB Protocol supported e.g. IIOP. An application proxy firewall, or just application 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.

To establish a connection to a server, a client first sets up a connection to the proxy. If the proxy is an outbound one, the ORB is configured with the address of the proxy. If the proxy is an inbound one, the server's IOR should contain the address of the proxy service 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. Irrespective of how the client interacts 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.

25.1.4 Rationale

There are a number of variables in establishing IIOP connections through firewalls. These variables include firewall topologies (placement of servers within networks and subnetworks), firewall types (transport, application), use of NAT, and the desire for secure communication. The underlying goal of this specification is to provide a means for ORBs to establish a connection from client to server in a uniform manner, regardless of these variables. Furthermore, a firewall generally maintains its security by only executing a small amount of well-tested code. For that reason, this specification makes it possible for a firewall to make access decisions in a deterministic manner without complex application-level interactions (e.g., invocations on CORBA objects) that may require an ORB implementation on the firewall.

25.2 IIOP Communication Via Firewalls

25.2.1 Firewall Traversal Overview

A server indicates that the objects it serves reside behind a firewall by placing a **TAG_FIREWALL_PATH** tagged component in the IIOP profile of the IOR for that object. The information in that tagged component enables the client to make an invocation through the firewall that will eventually reach the server. This information includes an ordered list of host addresses from the outermost inbound firewall to the target server.

In order to set up a connection to the target server, the client must first send a connection setup message to the target. This connection setup message is a **NegotiateSession** message discussed in Section 25.2.5, "Connection Setup Message," on page 25-12 and it contains a **FIREWALL_PATH** service context entry. The **FIREWALL_PATH** service context contains the information provided in the **TAG_FIREWALL_PATH** component of the IOR. This information allows firewalls to open up the correct connections along the path to the server. Once the entire virtual connection has been established, the target returns a **FIREWAL_PATH_RESP** service context in a return **NegotiateSession** message, and the client and server can communicate using GIOP. The IOR entries, service context entries, and NegotiateSession message will be outlined in more detail in the following sections.

25.2.2 FWSpec Structure

Each host along the path from the client to the target server can be identified as a collection of endpoints. An endpoint is an address by which a client can access the services provided by that host. That collection of endpoints is defined in an **FWSpec** structure, outlined below.

```
module CSIIOP {
    // A TAG_IIOP_SEC_TRANS component contains a struct
    // IIOP_SEC_TRANS that gives addressing information for IIOP services
    // on a host
    const IOP:ComponentId TAG_IIOP_SEC_TRANS = 43;
    struct IIOP_SEC_TRANS {
        TransportAddressList addresses;
    };
};

module Firewall {

    struct FWSpec {
        boolean is_intelligent;
        IOP::TaqggedComponentSeq endpoints;
    };
    typedef sequence<FWSpec> FWPath;

    // A TAG_PASSTHRU_TRANS component contains a struct
    // CSIIOP::IIOP_SEC_TRANS indicating the addressing information for
    // PASSTHRU services
    const IOP:ComponentId TAG_PASSTHRU_TRANS = 41;

};
```

is_intelligent

Indicates whether or not this host is capable of processing the connection setup message. The **is_intelligent** attribute must be `true` for application proxy firewalls and servers, and it must be **false** for transport level firewalls. The one exception to this rule is when an application proxy is behaving like a transport level firewall; i.e., there is a static mapping from an incoming address/port pair to an outgoing address/port pair and the firewall does not examine the connection setup message.

endpoints

The addresses and ports that this host can be contacted on. This field can contain any tagged components that specify address, port, and optional “type of service” information. At this point, this field can contain any combination of **TAG_TLS_SEC_TRANS**, **TAG_SECIOP_SEC_TRANS**, or **TAG_CSI_SEC_MECH_LIST** components, or one of two added components **TAG_PASSTHRU_TRANS** and **TAG_IIOP_SEC_TRANS**. These components describe the services that are provided by the firewall or server. Table 25-1 illustrates what each component indicates in a **TAG_FIREWALL_PATH** component.

Table 25-1 Use of Tagged Components in the FWSpec Structure

Component Type	**Service Provided by host**
TAG_IIOP_SEC_TRANS	The protocol used will be plain IIOP that will be inspected by the firewall as it passes through.
TAG_TLS_SEC_TRANS	The firewall or server will act as a TLS endpoint for the client. The firewall will inspect the IIOP traffic as it passes through.
TAG_SECIOP_SEC_TRANS	The firewall or server will act as a SecIOP endpoint. The firewall will inspect the IIOP traffic as it passes through.
TAG_PASSTHRU_TRANS	The firewall will act as a tunnel, allowing the client to directly connect to the server without firewall intervention. This is useful if the client and server need to establish a transport security association. Only a firewall FWSpec can contain this component.
TAG_CSI_SEC_MECH_LIST	The firewall or server understands CSIv2 and can handle the CSIv2 protocol. The actual transport used is specified in the TAG_CSI_SEC_MECH_LIST and can be one of TAG_IIOP_SEC_TRANS, TAG_TLS_SEC_TRANS, or TAG_SECIOP_SEC_TRANS. In all cases, the firewall or server will act as the secure transport endpoint and the firewall will inspect the protocol as it passes through.

The new **TAG_IIOP_SEC_TRANS** component simply provides a way of specifying addresses and ports for plaintext IIOP service. The new **TAG_PASSTHRU_TRANS** component can only be used for **FWSpec** entries describing firewalls, and it describes a service endpoint whereby a firewall will not inspect the data going through the firewall. This service allows clients and servers to connect to each other directly using a secure transport, so that the security association is setup from client to server, not client to firewall and firewall to server.

25.2.3 Firewall Tagged Component

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

```
module Firewall {

    // The component with ID TAG_FIREWALL_PATH contains an
    //  FWPath
    const IOP::ComponentId TAG_FIREWALL_PATH = 42;//OMG Allocated
};
```

The IOR Component with ID **TAG_FIREWALL_PATH** contains an **FWPath** type, which is a sequence of **FWSpec**s. This **FWPath** contains the **FWSpec**s for all of the firewalls along the path to the server, including the **FWSpec** of the server. These **FWSpec**s must be in order from the outermost inbound firewall to the server. The **TAG_FIREWALL_PATH** IOR component can only be placed in IORs with GIOP version 1.3 or higher because the firewall traversal algorithm relies on a GIOP message introduced in GIOP 1.3.

Since transport-level firewalls use a static mapping from external host/port pairs to internal host/port pairs, it is possible in some firewall configurations to eliminate the **FWSpec**s of transport-level firewalls from the **FWPath**. For example, take the case where a server is located behind a transport-level firewall. In this case the **FWPath** could contain one entry, the **FWSpec** of the server, which contains the address and port of the firewall. However, this is only possible in the case where there are no clients in the same enclave as the server. If clients did share the enclave with the server, they would not have sufficient information to contact the server. Therefore, this optimization should only be performed during the configuration of the firewall information on the server, and it is dependent upon the specific application and the firewall configuration.

The **TAG_FIREWALL_PATH** component may appear multiple times within a profile. Each individual component specifies a separate path to the target server. For example, a network may have multiple access points through multiple firewalls. If this is the case, then multiple firewall components could be specified to allow clients to access the server through either firewall.

25.2.4 Firewall Service Context

The **FIREWALL_PATH** service context must only be sent in the connection setup request. The service context contains the ordered list of firewalls that the client chose in order to reach the server. Each application level firewall will be able to parse this service context to determine what the next hop in the path to the server should be. The **FIREWALL_PATH** service context is defined as follows.

```
module Firewall {

    // The FIREWALL_PATH service context contains a FirewallPathContext
    //  structure
    const IOP::ServiceId FIREWALL_PATH = 20;   // OMG Allocated

    struct FirewallPathContext {
        long          host_index;
        FWPath        path;
    };
};
```

The **FIREWALL_PATH** service context contains a **FirewallPathContext** structure. The **host_index** field is an index indicating the current point in the firewall path. The outermost inbound firewall has index zero, and the index increases by one for each successive **FWSpec** with the target server having the highest index. The **path** attribute contains a sequence of **FWSpec**, and is obtained from the **FWPath** in the **TAG_FIREWALL_PATH** component of the IOR. The **FIREWALL_PATH** service context may only be sent in a **NegotiateSession** message and only during the connection setup period - see Section 25.2.5, “Connection Setup Message,” on page 25-12.

When a client ORB needs to open a connection to an object with a **TAG_FIREWALL_PATH** component in the **TAG_INTERNET_IOP** profile, the ORB extracts the **FWPath** entry from the IOR and places it in the **FIREWALL_PATH** service context. However, the **FWPath** in the IOR may contain multiple **IOP::TaggedComponents** for each **FWSpec**. When sending the connection setup message, the client ORB must choose, based upon that client’s policy, a single component from each **FWSpec** in the IOR to be placed in the corresponding **FWSpec** structure in the **path** member of the **FirewallPathContext** service context of the message. If the **path** incorrectly contains more than one endpoint for any **FWSpec**, a firewall or target server must use the first endpoint in the **endpoints** sequence when making connection decisions.

There are two reasons why it is important that an **FWSpec** in the **FirewallPathcontext**’s **path** member only contain a single **IOP::TaggedComponent**. First, a firewall or server must be able to determine whether the preceding firewall or client intends to connect using a secure transport, create a **PASSTHRU** connection, or neither. If multiple endpoints are available, the firewall or server may not be able to determine the preceding host’s intention. Second, using a single endpoint allows the firewall to make a deterministic decision about what endpoint to connect to on the subsequent host in the **path**. How the client ORB determines which endpoint to choose is described in Section 25.2.6, "ORB Policies," on page 25-13.

The **host_index** attribute is used by application firewalls and servers to locate their entry in the **path** in order to determine the transport connection type and address of the next host in the path. The **host_index** attribute always indicates the next "intelligent" host on the path to the server, as indicated by the **is_intelligent** element of the **FWSpec**. Thus, when an application firewall or server receives the **FIREWALL_PATH** service context, the **FWSpec** indicated by **host_index** indicates that firewall's or server's **FWSpec**.

The client ORB can use any strategy to choose which firewall to send the connection setup message to. For example, a client ORB might first attempt to invoke directly on the server, and if that fails, attempt to successively open a connection to the firewalls in **path** until a successful connection is established. Another strategy might be to always first open a connection to the outermost inbound firewall. No particular strategy will work for all network configurations. The policy for making this decision is discussed in Section 25.2.6.2, "Path Insertion Policy," on page 25-15.

After the client ORB has determined which firewall to send the connection setup message to, it must set the **host_index** field to be equal to the index of the **FWSpec** that is the **FWSpec** of the next "intelligent" host on the path to the server. If the **FWSpec** of the firewall that the client ORB chose to connect to has **is_intelligent** set to **true**, then **host_index** is the index of that **FWSpec**. Otherwise the ORB must find the first **FWSpec** in the **path** following the chosen firewall that has **is_intelligent** set to **true**, and **host_index** will be the index of that **FWSpec**. Note that the **FWSpec** of the target server will always have **is_intelligent** set to true, so there will always be a valid value for **host_index**. Similarly, as each application proxy processes the connection setup request message, it must increment the **host_index** to indicate the next **FWSpec** in the path that has **is_intelligent** set to **true**. This process ensures that the **host_index** always points to an "intelligent" host.

It should be noted that a client may maliciously create endpoints for a given **FWSpec** when building the **FIREWALL_PATH** service context. For that reason, firewall implementations should verify that the firewall policy allows a connection to the endpoint specified in the service context before opening a connection. This suggests that firewalls might provide a means of configuring policy such that certain protocol types (PASSTHRU,TLS, etc.) are allowed to only specific servers.

Once the connection has been established, the last intelligent firewall in the **path** sends a **FIREWALL_PATH_RESP** service context in another **NegotiateSession** message (see Section 25.2.8.2, "Inbound Firewall Traversal," on page 25-17). The contents of the **FIREWALL_PATH_RESP** service context are described below.

```
module Firewall {
    // The FIREWALL_PATH_RESP service context contains a
    //   FirewallPathRespContext
    const IOP::ServiceId FIREWALL_PATH_RESP = 21;   // OMG Allocated

    typedef unsigned short FWReplyStatusType;
    const FWReplyStatusType NO_EXCEPTION = 0;
    const FWReplyStatusType SYSTEM_EXCEPTION = 1;
    typedef sequence<octet> FWReplyBody;
    };
    FirewallPathRespContext {
        FWReplyStatusType    status;
        FWReplyBody          body;
    };
```

In the normal, non-exception case, the **FirewallPathRespContext** status field is **NO_EXCEPTION** and the body field is empty. If a firewall is unable to setup a connection, that firewall constructs an appropriate system exception for the failure, sets the status field to **SYSTEM_EXCEPTION**, and returns that exception in the body field of the **FirewallPathRespContext**, CDR marshaled as a sequence of octets.

25.2.5 Connection Setup Message

Before a client and server can begin communicating, the client needs to send a connection setup message to the server that contains enough information for the firewalls along the path to the server to open the correct connections. To provide this functionality, a new GIOP message type is needed. The definition for this message is shown below.

```
module GIOP {
    typedef octet MsgType;
    const MsgType Request = 0;
    const MsgType Reply = 1;
    const MsgType CancelRequest = 2;
    const MsgType LocateRequest = 3;
    const MsgType LocateReply =4;
    const MsgType CloseConnection = 5;
    const MsgType MessageError = 6;
    const MsgType Fragment = 7; // new in 1.1
    const MsgType NegotiateSession = 8; //new in 1.3

    struct NegotiateSessionHeader {
        ::IOP::ServiceContextList contexts;
    };
};
```

The **NegotiateSession** message is encoded as a GIOP header followed by a **NegotiateSession** header. The **NegotiateSession** message has no body.

The **NegotiateSession** message can be sent by either the client or the server, regardless of whether bi-directional GIOP is in use.

The **NegotiateSession** message contains a set of service contexts as defined for Request and Reply messages. However, the service contexts that are to be sent in a **NegotiateSession** message have certain restrictions. Namely, those service contexts can be restricted such that they can only be sent at certain times in the connection lifetime.

Those periods are:

- connection setup period
- session setup period
- session established period

The connection setup period occurs while a client is attempting to setup a logical connection to the server. This period only occurs when the GIOP connection consists of a sequence of transport-level connections (when firewalls or bridges are in use). Only service contexts that are used for connection setup may be sent during this period.

The session setup period occurs before any GIOP **Request** or **LocateRequest** messages have been sent on a connection. During this period, the client and server can send any number of **NegotiateSession** messages. After all session negotiation has taken place, the client sends the first GIOP **Request** or **LocateRequest**.

After the client has sent the first GIOP **Request** or **LocateRequest** message, the connection remains in the session established period for the duration of the connection. Service context entries that are defined to be sent during this period must take into account that a connection may have been negotiated as bi-directional, and therefore no guarantees can be made regarding the presence or lack of outstanding requests.

Session information negotiated using the **NegotiateSession** message is only valid for the duration of the connection on which the **NegotiateSession** message is sent. No session information is maintained across connections. Furthermore, there exists only one session per connection. Therefore, service contexts defined for the **NegotiateSession** message must indicate whether or not that context can be sent multiple times, and if so, whether the information contained in subsequent contexts is additive or whether it replaces information sent in previous contexts.

25.2.6 ORB Policies

25.2.6.1 Path Selection Policy

When building the **FIREWALL_PATH** service context, the client ORB must be able to select endpoints from each of the **FWSpec**s in the **FWPath** element of the **TAG_FIREWALL_PATH** component. As mentioned earlier, each **FWSpec** in the **path** element of the **FirewallPathContext** service context must have only one endpoint in order for the firewalls and server to determine what type of connection is being established. The client ORB selects endpoints based on the **PathSelectionPolicy**.

The **PathSelectionPolicy** is an ORB level policy. This policy may be overridden at the ORB level, or it can be overridden for specific objects that the ORB intends to invoke upon. The **PathSelectionPolicy** contains a set of **FeatureDirectives** for both the gateway and the target server. The gateway refers to the outermost-inbound server-side application firewall. These feature directives instruct the client ORB on how to choose a path to the target server based on features like target authentication and confidentiality. The client ORB must choose an endpoint for each firewall and server in the **TAG_FIREWALL_COMPONENT** such that the **PathSelectionPolicy** is satisfied.

The client application can specify a policy for target_authentication, confidentiality, and integrity for both the gateway and the server. Table 25-2 describes what the **FeatureDirective** values mean for each of these variables. Not all combinations of values are valid. A policy that contains two contradictory policy values is an invalid policy. The client ORB satisfies this policy by choosing endpoints from the **FWSpec** of each firewall and server that meet the policy requirements.

```
module Firewall {
    // Feature Directive
    // A Feature Directive is a general directive used in policy that
    // stipulates the use of a particular feature. Such examples include,
    // confidentiality, integrity, authentication, etc.
    typedef long FeatureDirective;

    // The FD_DoNotUse FeatureDirective means definitely do not to use
    // the feature.
    const FeatureDirective FD_DoNotUse = -2;

    // The FD_DoNotUseIfPossible FeatureDirective means do not to use
    // the feature if it is possible.
    const FeatureDirective FD_DoNotUseIfPossible = -1;

    // The FD_UseDefault FeatureDirective means to use or not to use
    // the feature depending on defaults.
    const FeatureDirective FD_UseDefault =  0;

    // The FD_DoNotUseIfPossible FeatureDirective means do not to use
    // the feature if it is possible.
    const FeatureDirective FD_UseIfPossible =  1;

    // The FD_DoNotUse FeatureDirective means definitely use
    // the feature.
    const FeatureDirective FD_Use =  2;

    struct FeatureDirectiveSet {
        FeatureDirective target_authentication;
        FeatureDirective confidentiality;
        FeatureDirective integrity;
    };

    const ::CORBA::PolicyType PATH_SELECTION_POLICY_TYPE = 61;
    local interface PathSelectionPolicy : CORBA::Policy {
        readonly attribute FeatureDirectiveSet target_server;
        readonly attribute FeatureDirectiveSet gateway;
    };
};
```

Table 25-2 Explanation of FeadtureDirectives for PathSelectionPolicy

Policy Variable	**Explanation**
target_authentication	Whether or not to authenticate this host. For example, if the gateway **FeatureSet** contains a **FeatureDirective** of **FD_UseIfPossible**, the client ORB should attempt to find a path that would cause the secure transport endpoint to be the gateway, enabling the client ORB to authenticate the gateway
confidentiality	Whether or not the client requires confidentiality to this host. For example, if the target_server **FeatureSet** contains a **FeatureDirective** of **FD_Use**, then the client ORB must find a path that is encrypted all the way to the server. However, trusted intermediate firewalls could act as secure transport endpoints along the path, depending on the value of **target_authentication** for both **target_server** and gateway.
integrity	Whether or not the client requires integrity to this host. This field has the same policy semantics as the confidentiality field.

25.2.6.2 Path Insertion Policy

The server specifies a list of **FWSpec**s in the **TAG_FIREWALL_PATH** component of the IOR. Depending on network topologies and relative locations of clients and servers, different clients may take different paths to reach the server. For instance, a client located in the same enclave as the server might connect directly to the server whereas a client on the internet will connect to the outermost inbound server-side firewall. Therefore, a client must have a policy about which FWSpec in the list of **FWSpec**s to attempt to connect to first. The **PathInsertionPolicy** is used for this purpose. The **PathInsertionPolicy** is a client-side policy that is defined as and ORB-level policy that can be overridden for specific objects.

```
module Firewall {
    typedef short PathInsertionPolicyValue;
    const PathInsertionPolicyValue OUTSIDE_IN = 0;
    const PathInsertionPolicyValue INSIDE_OUT = 1;
    const PathInsertionPolicyValue NO_FIREWALL = 2;

    // Allocated by OMG
    const CORBA::PolicyType PATH_INSERTION_POLICY_TYPE = 62;

    interface PathInsertionPolicy : CORBA::Policy {
        readonly attribute PathInsertionPolicyValue value;
    };
};
```

A policy value of **OUTSIDE_IN** indicates that if the client detects a **TAG_FIREWALL_PATH** component, the client should build the **FIREWALL_PATH** service context beginning with the **FWSpec** of the outermost-inbound firewall (this first **FWSpec**). If the client fails to connect to the server using that **FIREWALL_PATH**, the client should attempt to build a new **FIREWALL_PATH** context beginning with the next **FWSpec** in the list. The client should attempt to contact the server using sequentially increasing **FWSpec**s from the **FWSpec** list until a successful connection is established or all **FWSpec**s have been tried.

A policy value of **INSIDE_OUT** indicates that if the client detects a **TAG_FIREWALL_PATH** component, the client should build the **FIREWALL_PATH** service context beginning with just the **FWSpec** of the server (the last **FWSpec**). If the client fails to connect to the server using that **FIREWALL_PATH**, the client should attempt to build a new **FIREWALL_PATH** context beginning with the previous **FWSpec** in the list (this includes the **FWSpec** of the server). The client should attempt to contact the server using sequentially decreasing **FWSpec**s from the **FWSpec** list until a successful connection is established or all **FWSpec**s have been tried.

A policy value of **NO_FIREWALL** indicates the the client should ignore the **TAG_FIREWALL_PATH** component and only attempt to directly contact the server using the normal address information in the IIOP component.

25.2.7 Firewall and ORB Configuration

In order for a server to place information about the firewall path into an object's IOR, the server must know about the topology of the network. How that information is supplied to the server ORB is implementation dependent. Several solutions might include using a static configuration file or dynamically discovering the firewall topology from a configuration agent. Similarly, the firewall policy regarding which hosts are accessible from outside the enclave is implementation dependent.

25.2.8 Firewall Traversal Algorithm

A server ORB can determine if an object is to be accessed through a firewall via configuration information. When a server ORB determines that an object must be accessed through a firewall, the server ORB places a **TAG_FIREWALL_PATH** component into the IIOP profile of the IOR that contains the firewall path information as described earlier. In addition, the ORB must place some address and port number into the IIOP profile itself as described in section 15.7.2 of the CORBA 2.4.1 specification. The address provided in the IIOP profile shall be the address (or preferably DNS name) and port of the **FWSpec** of the server.

The client ORB determines that an object is accessed through a firewall by the presence of the **TAG_FIREWALL_PATH** component in the IIOP profile. The client ORB then prepares to send the connection setup message. First the client must prepare a **FIREWALL_PATH** service context, extracting the **FWPath** information from the IOR as described earlier. Recall that the connection setup message will only be sent if there is an "intelligent" firewall on the path to the server. This includes any outbound firewall proxies as described in the next section. Next the client must traverse any outbound firewalls as described in the next section.

25.2.8.1 Outbound Firewall Traversal

Outbound firewall traversal is typically simpler than inbound traversal due to less restrictive policies for outbound connections. In some cases, it may not be necessary to take any additional steps for outbound firewall traversal than to just open a TCP connection to the outermost inbound firewall on the server side. However, there are cases in which a firewall security policy will not allow arbitrary outbound connections, so there must be a means to handle those situations.

The approach used for outbound IIOP connections is the same as the approach for other Internet protocols like HTTP or FTP. Namely, the client must be configured with an outbound IIOP proxy to which it can send its connection requests. A client ORB must determine whether or not the client-side proxy is needed when making a connection, and if so, the client will open a connection to the proxy rather than the outermost inbound server-side firewall. How a client determines whether or not an outbound proxy is needed is an implementation issue, and other Internet protocol implementations can be used as a model for this implementation. Likewise, how the client ORB is configured with outbound proxy information is an implementation issue.

The client ORB shall be configured with information equivalent to a FWSpec for the outbound proxy. When choosing a path to the server, the outbound proxy should be considered in addition to the inbound server-side firewalls. This means that the **PathSelectionPolicy** must also be satisfied in choosing a path through an outbound proxy, and that the **FWSpec** for the outbound proxy must be placed as the first **FWSpec** in the **FIREWALL_PATH** context.

If an outbound proxy is within a nested set of enclaves, that proxy could also be configured with an outbound proxy. Since the proxy can determine the target of the connection setup request using the **FIREWALL_PATH** context, the proxy makes the same decision as the client did in the previous step; i.e., determine if the outermost inbound firewall can be contacted directly. If not, forward the connection setup request to the next outbound proxy. When an outbound proxy selects an **FWSpec** for the next outbound proxy, that proxy must choose a PASSTHRU endpoint for secure transports or an IIOP or PASSTHRU endpoint for the plain IIOP transport. The reason for this is that a client is typically only configured with one outbound proxy, and so it is unable to choose a path through all of the outbound firewalls. Because successive proxies don’t know the **PathSelectionPolicy** of the client, they are unable to determine the correct endpoint type. The choice of a PASSTHRU endpoint will not affect any choices that the client made regarding whether the client intended to connect to the gateway or to the server. If the client was configured with a sequence of outbound proxies, the client can place the list of outbound proxies in the **FIREWALL_PATH** context before the **FWSpec**s of the inbound firewalls, in which case the use of IIOP or PASSTHRU endpoint types is not necessary because the client ORB predetermines the outbound path.

25.2.8.2 Inbound Firewall Traversal

As each inbound application proxy firewall receives the **NegotiateSession** message with the **FIREWALL_PATH** context, it must first locate its entry in the firewall path using the **host_index** field from the service context. The firewall then takes note of the connection type and examines the **FWSpec** of the next host in the path to the server. The firewall can then make a determination as to whether or not the requested connection is allowed. If the connection is not allowed, the firewall must return a **NO_PERMISSION** exception in the **FIREWALL_PATH_RESP** service context. Otherwise, the firewall opens a connection to the next host in the **path**.

Before forwarding the connection setup message, the firewall must increment the **host_index** value. The **host_index** must indicate the next intelligent host in the **path**, indicated by the **is_intelligent** field in the **FWSpec** having a value of **true**. In addition, the firewall must determine if it is it the last "intelligent" firewall in the path to the server. If so, then the connection setup message should not be forwarded, rather the firewall should return a **NO_EXCEPTION** reply in the **FIREWALL_PATH_RESP** service context. If not, then the firewall forwards the connection setup request to the next host in the **path**.

Eventually the entire logical connection from the client to the server is set up, or an exception has been returned, and all of the GIOP connections have been closed. If the connection was successfully established, then if the client or any of the firewalls need to begin an TLS handshake, the handshake process takes place using the TCP connections that had previously been established. After the TLS handshakes are completed, the firewall traversal processes is complete.

25.2.9 Callback Invocations

Though this specification provides a solution to the routing and location transparency issues for a variety of network topologies including those employing NAT, policy issues still exist that make it impossible for firewalls to be completely transparent to CORBA applications. Specifically, callbacks present a unique problem. Firewalls are usually configured to allow limited access to a limited set of well-known server addresses. Most hosts cannot be accessed from outside the enclave. CORBA clients may sometimes also act as servers, requiring that invocations from hosts outside the enclave be allowed through the firewall. There are several possibilities for dealing with this problem.

First, if a CORBA client is not behind a firewall, then the server can make callback invocations without restrictions other than the server-side outbound connection policy. This is the simplest case. A related case is if the client is protected by a firewall, but the firewall policy allows inbound connections to that client. In this case, the client and server roles are reversed for the callback invocations, and the mechanisms previously outlined in this specification can be used for firewall traversal.

Second, if a CORBA client is behind a firewall then the server can make callback invocations on the client using bi-directional GIOP (see the accompanying document orbos/2001-08-03). Bi-directional GIOP allows a client to receive GIOP messages normally intended for a server and vice-versa. In this case, the server simply uses the channel opened by the client for callback invocations, and there is no need for additional algorithms.

Third, a CORBA client may be behind a firewall and a third-party might desire to make an invocation on an object managed by the client. Third-party in this case indicates that the client has not initiated a connection to that host, instead, that host received an IOR for a client object from some other host. Firewall policy might not permit inbound connections to that client, and a bi-directional GIOP connection is not possible because the client does not have a connection open to the third-party host. In this case it is not possible for a callback invocation to occur. Keeping this in mind, application developers must write their applications such that a client is given a reference to the third-party host instead. Then the client can contact that host, and callbacks can occur via bi-directional GIOP. This is the normal application development model for Internet applications, but not necessarily for CORBA applications. CORBA applications will need to be carefully designed in order to avoid third-party callbacks through firewalls.

25.2.10 Implications of Secure Transports

25.2.10.1 Identity Delegation

There are a number of additional problems with CORBA firewall traversal when secure transport connections are desired. First, secure transports establish a security association between the hosts that act as endpoints for the connection. Some secure transports, in particular TLS, do not provide any means for delegation of authorization or identity. It may be desirable to have a firewall act as an endpoint for a secure connection in order to inspect the protocol for errors or to provide access control to the enclave. However, if a firewall is an endpoint for a secure connection, the identity provided to the client or server will be that of the firewall, and not of the client or server as would be desired.

In order to work around this problem, a new CSIv2 **IdentityToken** type, **ITTCompoundToken**, is introduced. The purpose of this token is for the firewall to be able to provide the server with the information that the client provided to the firewall. The definition of that token is:

```
module CSI {
    typedef sequence<IdentityToken> IdentityTokenList;
    struct CompoundIdentityToken {
        IdentityToken         asserted_identity;
        IdentityToken         authenticated_transport_identity;
        IdentityToken         authenticator_identity
        IdentityTokenList     authentication_trail;
    };

    const IdentityTokenType ITTCompoundToken = 16;

    // This structure goes in the 'id' value of IdentityToken.  An
    // IdentityExtensionToken with the_type=ITTCompoundToken contains a
    // CompoundIdentityToken
    struct IdentityExtensionToken {
        IdentityTokenType the_type;
```

```
            IdentityExtension id;
        };
};
```

The **CompoundIdentityToken** enables the firewall to pass on to the server the authentication information that it collected. The fields of the **CompoundIdentityToken** are defined as follows:

asserted_identity the identity that the client presented in the identity_token field of the CSIv2 EstablishContext message that was intercepted by the firewall.

authenticated_transport_identity the identity that was authenticated by the firewall through the secure transport.

authenticator_identity the identity that the client presented in the **client_authentication_token** field of the CSIv2 **EstablishContext** message that was intercepted by the firewall. The firewall only provides a value in this field other than **ITTAbsent** if the firewall authenticated the client using the **client_authentication_token**.

authentication_trail this is a field that provides a place for a firewall or a chain of firewalls to place their own identities to give the server additional information about what firewalls the invocation has passed through.

The fields in the **CompoundIdentityToken** shall not contain an **IdentityTokenType** of **ITTCompoundToken**.

If the receiver of a **CompoundIdentityToken** trusts the sender of the token, then the receiver can use that information in making a trust decision. For instance, if a server receives a **CompoundIdentityToken** from a trusted firewall, the resulting invocation principal would be the identity in the **asserted_identity** field, given that the server trusts the identity provided in the **authenticated_transport_identity** field or the **authenticator_identity** field to make invocations as the **asserted_identity**. The invocation principal and trust determinations shall be the same as presented in the CSIv2 specification, as if the server had authenticated the identities in the **authenticated_transport_identity** and/or **authenticator_identity** fields. But the server shall only accept a **CompoundIdentityToken** if the server's security policy allows the firewall identity to present a **CompoundIdentityToken**.

If a firewall detects a **CompoundIdentityToken** in the **EstablishContext** message, but the firewall policy does not allow the client to use a **CompoundIdentityToken**, the firewall shall generate a **NO_PERMISSION** exception to send to the client. The firewall shall not forward that message to the server. If the server detects a **CompoundIdentityToken** in an **EstablishContext** message that did not come from a trusted source, the server shall generate a **NO_PERMISSION** exception to send to the client. The server shall not proceed to execute the object invocation. It is important that **CompoundIdentityToken**s are only accepted from authenticated sources that are trusted to use them because the server has no proof that the sender actually authenticated the client information in the **CompoundIdentityToken**.

The **CompoundIdentityToken** can be used in several ways. First, when a client sends **EstablishContext** messages to the server, but the secure transport connection is with the firewall, the firewall can introduce the **CompoundIdentityToken** into the **EstablishContext** message.

If the **EstablishContext** message from the client contains an **identity_token**, then the **asserted_identity** field of the **CompoundIdentityToken** shall be set to that value. Otherwise the value of **asserted_identity** shall be **ITTAbsent**.

If the **EstablishContext** message from the client contains an **authentication_token**, the firewall has several options. If the server does not support the type of token in **authentication_token**, the firewall must either authenticate the client's token and place the value of the authenticated identity in the **authenticator_identity** field and remove the **authentication_token** from the **EstablishContext** message, or the firewall must generate a **NO_PERMISSION** exception to return to the client. If the server does support the type of token in the **authentication_token**, then the firewall can either authenticate the client's token and place the value of the authenticated identity in the **authenticator_identity** field, or leave the **authentication_token** in the **EstablishContext** message for the server. If the firewall does not support that type of **authentication_token**, the firewall could leave the authentication token in the message for the server to verify. Whatever decision the firewall makes in this case must be consistent with the server's requirements for an **authentication_token**. If the firewall does not set the value of the **authenticator_identity** field, then the value of that field shall be **ITTAbsent**.

If the firewall authenticated the client's identity through a secure transport, the firewall shall place the value of that identity in the **authenticated_transport_identity** field. Otherwise the value of the **authenticated_transport_identity** field shall be **ITTAbsent**.

Finally, the firewall may optionally place its own identity information into the **authentication_trail** field. The presence of a firewall's identity in the authentication_trail field is dependant upon the configuration of that firewall. Once the **CompoundIdentityToken** has been constructed, the firewall replaces the **identity_token** value in the **EstablishContext** message with the **CompoundIdentityToken** and forwards the message on to the server. However, if the firewall's authentication of either the client's transport identity or **authentication_token** identity fails, the firewall shall not forward the message to the server, but rather return a **NO_PERMISSION** exception.

A second way in which the **CompoundIdentityToken** can be used is if the firewall receives an **EstablishContext** message that already contains a **CompoundIdentityToken**. If the firewall trusts the sender of the **CompoundIdentityToken**, the firewall may optionally add its identity to the **authentication_trail** field and forward the message on to the server. If the firewall does not trust the sender to provide a **CompoundIdentityToken**, then the firewall shall not forward the message to the server. Instead the firewall shall send a **NO_PERMISSION** exception to the client.

A third way in which the **CompoundIdentityToken** can be used is if the firewall receives an invocation from a client over an authenticated connection, but the client did not provide an **EstablishContext** message for that invocation. In that case, if the server supports CSIv2 **CompoundIdentityToken**s, the firewall may optionally establish a CSIv2 security association with the server by adding an **EstablishContext** message to the GIOP Request message. The **identity_token** in the **EstablishContext** message shall be a **CompoundIdentityToken**, constructed as outlined earlier in this section, with the client's authenticated transport identity in the **authenticated_transport_identity** field. The firewall shall strip the subsequent **CompleteEstablishContext** or **ContextError** messages from the replies before forwarding them to the client.

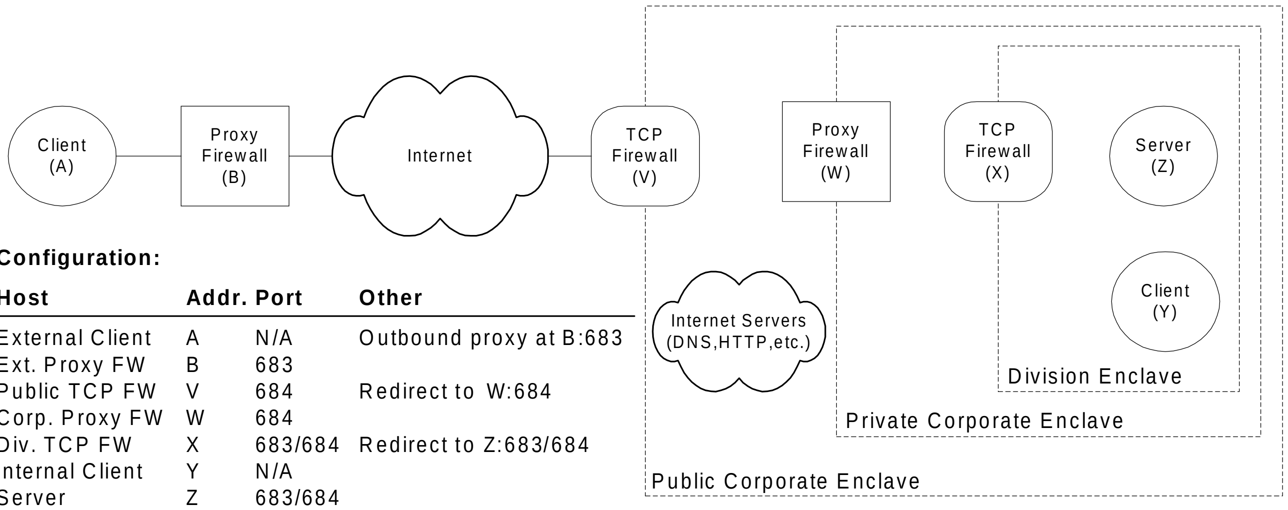
25.2.10.2 Connection Setup

In order to eliminate differences in setting up secure and insecure connections, the connection setup message is sent using IIOP without any protection of a secure transport. This allows proxies to setup PASSTHRU connections because they will be able to determine the desired target of the invocation. Once the logical connection from client to server has been established (the connection setup request and reply have been seen by all of the firewalls), any necessary transport-level security context establishment takes place over the pre-existing TCP/IP connections.

The implication this has on application firewalls is that an application proxy can always expect to see plaintext GIOP on incoming connections. It can then later set up any security contexts on that connection, if necessary. However, if a server also receives the connection setup message, the server could see plaintext GIOP messages from a firewall or secure transport messages from clients within the server's enclave. Therefore the server will be unable to determine whether to process incoming connections as GIOP or as a secure transport. For that reason, the connection setup message must not ever be forwarded to a server. That is, the last "intelligent" host in the firewall path must send the correct reply to the connection setup request. This means that if there are no application proxy firewalls, the client ORB must still open a TCP connection to the server, but the client ORB will not send a **NegotiateSession** message containing firewall path information. Further, every application firewall must check if it is the last intelligent device on the path to the server, and if so, attempt to establish a TCP/IP connection to the next host in the **path**, and then return a response to the connection setup request. Note that in considering whether or not to create or forward the connection setup message, a client must also consider any outbound proxies as described in Section 25.2.8.1, "Outbound Firewall Traversal," on page 25-17. An outbound proxy is also considered an intelligent device for this determination. Likewise, any outbound proxies will also have to consider other outbound proxies in the path to the server when making this determination.

25.3 Firewall Traversal Use Case

The following example serves to illustrate most of the important features of this specification. In particular, the example shows how the traversal algorithm applies to a specific firewall configuration. The example is shown in Figure 25-3.



Configuration:

Host	Addr.	Port	Other
External Client	A	N/A	Outbound proxy at B:683
Ext. Proxy FW	B	683	
Public TCP FW	V	684	Redirect to W:684
Corp. Proxy FW	W	684	
Div. TCP FW	X	683/684	Redirect to Z:683/684
Internal Client	Y	N/A	
Server	Z	683/684	

Figure 25-3 Firewall configuration for use case

For this scenario, the IOR of an object on the server would have an IIOP profile shown in Figure 25-4. This IOR contains a **TAG_TLS_SEC_TRANS** component that could be used by the internal client to invoke on the server. In addition, the IOR contains a **TAG_FIREWALL_PATH** component that allows external clients to traverse the firewalls in order to invoke on the server.

```
1:  Profile ID:                 0 (TAG_INTERNET_IOP)
2:  Version:                    1.3
3:  Host:                       Z
4:  Port:                       683
5:  Object Key:                 "my_object"
6:  # of Components:            2
7:  Component 0:
8:      Component ID:           20 (TAG_TLS_SEC_TRANS)
9:      Target Supports:        0x7E
10:     Target Requires:        0x7E
11:     Port:                   684
12: Component 1:
13:     Component ID:           xx (TAG_FIREWALL_PATH)
14:     # of FWSpecs:           3
15:     FWSpec 0:                       //Refers to Corp. Proxy firewall
16:         Intelligent:        True    // using the public TCP firewall's
17:         # of Endpoints:     2      // address
18:         Endpoint 0:
19:             ComponentId:    (TAG_CSI_SEC_MECH_LIST)
20:             target_requires:   Confidentiality, Integrity,
21:                     EstablishTrustinTarget, EstablishTrustInClient,
22:                     IdentityAssertion
23:             transport_mech:  (TAG_TLS_SEC_TRANS)
24:                 target_supports: same as line 20
```

```
25:                 target_requires: same as line 20
26:                 addresses:     V:684
27:             as_context_mech  empty
28:             sas_context_mechempty
29:         Endpoint 1:
30:             ComponentId:      (TAG_PASSTHRU_TRANS)
31:             addresses:        V:684
32:     FWSpec 1:                         //Refers to the division TCP
33:         Intelligent:          False   // firewall.  Options for both TLS
34:         # of Endpoints:       2
35:         Endpoint 0:
36:             ComponentId:      (TAG_IIOP_SEC_TRANS)
37:             addresses:        X:683
38:         Endpoint1:
39:             ComponentId:      (TAG_PASSTHRU_TRANS)
40:             addresses:        X:684
41:     FWSpec 2:                         //Refers to the server endpoints.
42:         Intelligent:          True    // One each for IIOP/TLS and IIOP
43:         # of Endpoints 2:
44:         Endpoint 0:
45:             ComponentId:      (TAG_IIOP_SEC_TRANS)
46:             addresses:        Z:683
47:         Endpoint 1:
48:             ComponentId:      (TAG_CSI_SEC_MECH_LIST)
49:             transport_mech:   (TAG_TLS_SEC_TRANS)
50:                 target_requires:    same as line 20
51:                 target_supports:    same as line 20
51:                 addresses:          Z:684
52:             as_context_mech:        empty
53:             sas_context_mech:       empty
```

Figure 25-4 IIOP profile of an object on the server containing firewall information

This firewall component contains three **FWSpec**s but there are actually four hosts, three firewalls and the server, that should be represented in this firewall component. The reason for this is that **FWSpec** 0 (lines 15-31) makes use of an optional optimization. Since the public TCP firewall is configured to automatically redirect connections to the proxy firewall, **FWSpec** 0 actually represents the proxy firewall but the address in the **FWSpec** is the address of the public TCP firewall (line26). This optimization is possible in this case because there are no clients in the public corporate enclave that will make invocations on the server. If there were such clients, this particular firewall component would not have enough information for those clients to make an invocation on the server. Notice also that only authenticated TLS connections or PASSTHRU connections are allowed to the first firewall. This was configured by the administrator so that only secure connections are allowed from the internet.

FWSpec 1 (lines 32-40) represents the division TCP firewall. Since this node is a TCP firewall, the isIntelligent field is set to FALSE (line 33). Notice that in this **FWSpec** there are endpoints available for both IIOP and IIOP/TLS. The administrator configured this firewall so that internal corporate users could access the server without using TLS. Instead of placing the IIOP endpoints for **FWSpec** 1 and **FWSpec** 2 in the same firewall component as the TLS endpoints, the administrator could have put the IIOP endpoints in a separate firewall component altogether. This choice depends on the specific application. In this example, one use case will make use of both the TLS and non-TLS endpoints, so the administrator chose to place all of the endpoints in a single firewall component.

FWSpec 2 (lines 41-53) represents the server. A server will always have the **is_intelligent** field set to TRUE (line 42). This **FWSpec** also contains both an IIOP and an TLS/IIOP endpoint. Unlike in **FWSpec** 0, **FWSpec**s 1 and 2 actually contain the address of the division TCP firewall and the server respectively. The reason an optimization was not used in this case is that there is a client in the server's enclave that will also make invocations on the server. If the previously mentioned optimization were performed, then the client may not have enough information to invoke on the server. In this case, since the client is in the same enclave as the server, the client could also use the information contained in the IIOP profile itself since the server **FWSpec** information must be the information provided in the IIOP profile. However, this example illustrates how the **FWSpec** optimization may cause the server to be unavailable to some clients.

The following use cases demonstrate the algorithm for choosing endpoints and opening a connection to the server.

25.3.1 Secure Connection to Gateway

Once the external client has received the IOR it must choose the endpoint from each **FWSpec** that it will use to contact the server, based on its **PathSelectionPolicy**. For this use case, the client has a **PathSelectionPolicy** of:

target_server: target_authentication=FD_DoNotUseIfPossible, confidentiality=FD_Default, integrity=FD_Default

gateway: target_authentication=FD_Use, confidentiality=FD_Use, integrity=FD_Use

This policy indicates that the client must establish an secure connection to the gateway in order to authenticate the gateway and create a secure channel to the gateway. So based on that policy, the client ORB selects appropriate endpoints from the **TAG_FIREWALL_PATH** component and builds a **FIREWALL_PATH** service context containing **FWSpec**s with those endpoints. That service context is shown in Figure 25-5.

```
1:  Service ID:                     xx (FIREWALL_PATH)
2:  Host Index:                     0
3:  # of FWSpecs:                   4
4:  FWSpec 0:                       // for the outbound proxy
5:      Intelligent:                True
6:      # of Endpoints:             1
7:      Endpoint 0:
8:          ComponentId:            (TAG_PASSTHRU_TRANS)
9:          addresses:              B:683
10: FWSpec 1:
11:     Intelligent:                True
```

```
12:      # of Endpoints:             1
13:          Endpoint 0:
14:              ComponentId:        (TAG_CSI_SEC_MECH_LIST)
15:              target_requires:    Confidentiality, Integrity,
16:                          EstablishTrustinTarget, EstablishTrustInClient,
17:                          IdentityAssertion
18:              transport_mech:     (TAG_TLS_SEC_TRANS)
19:                  target_supports: same as line 20
20:                  target_requires: same as line 20
21:                  addresses:      V:684
22:              as_context_mech     empty
23:              sas_context_mech    empty
24: FWSpec 2:
25:      Intelligent:                False
26:          # of Endpoints:         1
27:          Endpoint 0:
28:              ComponentId:        (TAG_IIOP_SEC_TRANS)
29:              addresses:          X:683
30: FWSpec 3:
31:      Intelligent:                True
32:          # of Endpoints 1:
33:          Endpoint 0:
34:              ComponentId:        (TAG_IIOP_SEC_TRANS)
35:              addresses:          Z:683
```

Figure 25-5 FIREWALL_PATH service context for secure gateway policy

In this case the ORB chose to make an TLS connection to the application proxy (line 12) and make non-TLS connections to the server (lines 22 and 28). It would also have been acceptable under the given policy to make an additional TLS connection from the application proxy to the server, rather than to connect without TLS.

The client ORB is configured with an outbound proxy, so it first opens a TCP connection to the proxy (B:683). The client ORB then sends the NegotiateSession message with the **FIREWALL_PATH** service context. When the proxy receives this request message, it notes that this is an outbound connection, and determines the type of connection desired and the destination address using the information in the service context. In this case, the connection type is PASSTHRU and the destination address is V:684. The proxy then opens a TCP connection to V:684, updates the **host_index** field, and forwards the original **NegotiateSession** message.

Recall that V:684 is actually the external address of the outermost TCP firewall. This connection is redirected to W:684. The proxy at that address receives the message and begins to parse it in the same manner that the client-side proxy did. The proxy determines that this inbound connection must be terminated as a TLS connection. It also determines that the next hop is X:683, and that this connection has a type of IIOP meaning that no TLS will be used. The proxy must also update the **host_index** field. Since the next hop is not an intelligent device, the proxy increments the **host_index** field by two to indicate the next **FWSpec** that has isIntelligent set to TRUE. The proxy then opens a TCP connection to X:683. But since the proxy is the last intelligent device other than the server, the proxy does not forward the connection setup request. Instead it must send a **NegotiateSession** message with a **NO_EXCEPTION FIREWALL_PATH_RESP** back to the client, indicating that the connection was successfully established.

The reply message is subsequently forwarded back through the firewalls to the client. At this point the connection has been established and the TLS handshakes must occur. In this case, the client begins an TLS handshake, and the server-side proxy firewall terminates the TLS connection. Once the security association between those two hosts has been established, normal GIOP communication can occur. All of the firewalls forward the messages, and the server-side firewalls that have access to the plaintext messages can examine the messages for correctness or perform access control on the requests if desired.

25.3.2 End-to-End Secure Connection

This use case is very similar to the previous case except that the client ORB has a different **PathSelectionPolicy**. This example only includes the changes from the previous example and does not give a full description of the traversal algorithm. The **PathSelectionPolicy** is:

target_server: target_authentication=FD_Use, confidentiality=FD_Use, integrity=FD_Use

gateway: target_authentication=FD_DoNotUse, confidentiality=FD_Use, integrity=FD_Use

The service context entry selected by the client is shown in Figure 25-6.

In order to satisfy the **PathSelectionPolicy**, the client ORB chose a PASSTHRU endpoint for the proxy firewall (line 14), a PASSTHRU endpoint for the division TCP firewall (line 20), and a TLS connection to the server (line 49). Therefore, all of the intermediate hosts will only forward the GIOP messages, and the server will terminate the TLS connection with the client.

The firewall traversal algorithm is identical to the previous example except that the gateway firewall does not terminate the TLS connection because the endpoint type specified in the service context is PASSTHRU instead of TLS. Instead the server terminates the TLS connection.

These are just a couple examples of the many different configurations of firewalls that can be supported. However the process for establishing a connection is very similar in all cases.

```
1:  Service ID:                     xx (FIREWALL_PATH)
2:  Host Index:                     0
3:  # of FWSpecs:                   4
4:  FWSpec 0:                       // for the outbound proxy
5:      Intelligent:                True
6:      # of Endpoints:             1
7:      Endpoint 0:
8:          ComponentId:            (TAG_PASSTHRU_TRANS)
```

```
9:              addresses:              B:683
10: FWSpec 1:
11:         Intelligent:                True
12:         # of Endpoints:             1
13:         Endpoint 0:
14:             ComponentId:            (TAG_PASSTHRU_TRANS)
15:             addresses:              V:684
16: FWSpec2 2:
17:         Intelligent:                False
18:         # of Endpoitns:             1
19:         Endpoint 1:
20:             ComponentId:            (TAG_PASSTHRU_TRANS)
21:             addresses:              X:684
22: FWSpec 3:
23:         Intelligent:                True
24:         # of Endpoints:             1
25:         ComponentId:                (TAG_CSI_SEC_MECH_LIST)
49:             transport_mech:         (TAG_TLS_SEC_TRANS)
50:                 target_requires:    same as line 20
51:                 target_supports:    same as line 20
51:                 addresses:          Z:684
52:             as_context_mech:        empty
53:             sas_context_mech:       empty
```

Figure 25-6 FIREWALL_PATH service context for END_TO_END secure policy

25.4 Conformance and CORBA Changes

An ORB implementation that is compliant with this specification must implement the data structures and algorithms presented in sections 25.2.1 to 25.2.8. A firewall implementation that is compliant with this specification must implement the data structures and algorithms presented in Section 25.2.1 - 25.2.2, 25.2.4 - 25.2.5, 25.2.7 - 25.2.9. Though an implementation may not support secure transports, it must be able to interpret the FIREWALL_PATH service context, regardless of the connection type, and act accordingly. Section 25.2.10, "Implications of Secure Transports," on page 25-19 is optional, as is implementation of bi-directional GIOP.

This document supersedes the previously adopted CORBA firewall specification. In addition, OMG document orbos/2001-08-03 specifying changes to bi-directional GIOP supersedes the adopted specification for bi-directional GIOP. These specifications are not backwards-compatible with the previous specifications and they are intended to make it possible to create a functional protocol for the interoperation of ORBs and firewalls.

Appendix A Consolidated IDL

A.1 Firewall Module

```
module Firewall {

    struct FWSpec {
        boolean is_intelligent;
        IOP::TaqggedComponentSeq endpoints;
    };
    typedef sequence<FWSpec> FWPath;

    // A TAG_PASSTHRU_TRANS component contains a struct
    // CSIIOP::IIOP_SEC_TRANS indicating the addressing information for
    // PASSTHRU services
    const IOP:ComponentId TAG_PASSTHRU_TRANS = 41;// OMG Allocated

    // The component with ID TAG_FIREWALL_PATH contains a
    //  FWPath
    const IOP::ComponentId TAG_FIREWALL_PATH = 42;//OMG Allocated

    // The FIREWALL_PATH service context contains a FirewallPathContext
    //  structure
    const IOP::ServiceId FIREWALL_PATH = 20;   // OMG Allocated

    struct FirewallPathContext {
        long          host_index;
        FWPath        path;
    };
    // The FIREWALL_PATH_RESP service context contains a
    //   FirewallPathRespContext
    const IOP::ServiceId FIREWALL_PATH_RESP = 21;   // OMG Allocated

    typedef unsigned short FWReplyStatusType;
    const FWReplyStatusType NO_EXCEPTION = 0;
    const FWReplyStatusType SYSTEM_EXCEPTION = 1;
    typedef sequence<octet> FWReplyBody;
    };
    FirewallPathRespContext {
        FWReplyStatusType    status;
        FWReplyBody          body;
    };
    // Feature Directive
    // A Feature Directive is a general directive used in policy that
    // stipulates the use of a particular feature. Such examples include,
    // confidentiality, integrity, authentication, etc.
    typedef long FeatureDirective;

    // The FD_DoNotUse FeatureDirective means definitely do not to use
```

```
        // the feature.
        const FeatureDirective FD_DoNotUse = -2;

        // The FD_DoNotUseIfPossible FeatureDirective means do not to use
        // the feature if it is possible.
        const FeatureDirective FD_DoNotUseIfPossible = -1;

        // The FD_UseDefault FeatureDirective means to use or not to use
        // the feature depending on defaults.
        const FeatureDirective FD_UseDefault =  0;

        // The FD_DoNotUseIfPossible FeatureDirective means do not to use
        // the feature if it is possible.
        const FeatureDirective FD_UseIfPossible =  1;

        // The FD_DoNotUse FeatureDirective means definitely use
        // the feature.
        const FeatureDirective FD_Use =  2;

        struct FeatureDirectiveSet {
            FeatureDirective target_authentication;
            FeatureDirective confidentiality;
            FeatureDirective integrity;
        };

        // Allocated by OMG
        const ::CORBA::PolicyTypePATH_SELECTION_POLICY_TYPE = 61;
        local interface PathSelectionPolicy : CORBA::Policy {
            readonly attribute FeatureDirectiveSet target_server;
            readonly attribute FeatureDirectiveSet gateway;
        };

        typedef short PathInsertionPolicyValue;
        const PathInsertionPolicyValue OUTSIDE_IN = 0;
        const PathInsertionPolicyValue INSIDE_OUT = 1;
        const PathInsertionPolicyValue NO_FIREWALL = 2;

        // Allocated by OMG
        const CORBA::PolicyType PATH_INSERTION_POLICY_TYPE = 62;

        interface PathInsertionPolicy : CORBA::Policy {
            readonly attribute PathInsertionPolicyValue value;
        };

    };
```

A.2 Changes to GIOP

```
    module GIOP {
        typedef octet MsgType;
    const MsgType Request = 0;
    const MsgType Reply = 1;
    const MsgType CancelRequest = 2;
    const MsgType LocateRequest = 3;
    const MsgType LocateReply =4;
    const MsgType CloseConnection = 5;
    const MsgType MessageError = 6;
    const MsgType Fragment = 7; // new in 1.1
    const MsgType NegotiateSession = 8;

    struct NegotiateSessionHeader {
        ::IOP::ServiceContextList contexts;
    };
};
```

A.3 Changes to CSI

```
module CSIIOP {
    // A TAG_IIOP_SEC_TRANS component contains a struct
    // IIOP_SEC_TRANS that gives addressing information for IIOP services
    // on a host
    const IOP:ComponentId TAG_IIOP_SEC_TRANS = 43; // OMG allocated
    struct IIOP_SEC_TRANS {
        TransportAddressList addresses;
    };

    typedef sequence<IdentityToken> IdentityTokenList;
    struct CompoundIdentityToken {
        IdentityToken     asserted_identity;
        IdentityToken     authenticated_transport_identity;
        IdentityToken     authenticator_identity
        IdentityTokenList authentication_trail;
    };

    const IdentityTokenType ITTCompoundToken = 16;// OMG allocated

    // This structure goes in the ‘id’ value of IdentityToken.  An
    // IdentityExtensionToken with the_type=ITTCompoundToken contains a
    // CompoundIdentityToken
    struct IdentityExtensionToken {
        IdentityTokenType the_type;
        IdentityExtension id;
    };
};
```