Evaluation of Three Approaches for CORBA Firewall/NAT Traversal*

Antonio Theophilo Costa, Markus Endler, and Renato Cerqueira

PUC-Rio - Catholic University of Rio de Janeiro
{theophilo, endler, rcerq}@inf.puc-rio.br

Abstract. Applications that use CORBA as communication layer often have some restrictions for multi-domain deployment. This is particularly true when they have to face firewall/NAT traversal. Furthermore, nowadays there isn't a well-accepted unique or standardized solution adopted by all ORBs, compelling applications using this middleware to use proprietary solutions that sometimes do not address the environment restrictions in which they are deployed (e.g. impossibility to open firewall ports). This work presents and compares three solutions for firewall/NAT traversal by CORBA-based distributed applications, each one suitable for a specific situation and exploring its advantages. Examples of such situations are the possibility of opening firewall ports or the possibility of starting a TCP connection to the outside network.

1 Introduction

From its beginning, the CORBA specification [Group, 2004a] [Henning and Vinoski, 1999], [Bolton and Walshe, 2001] was designed aiming to offer to developers a reduction of the complexity of developing distributed object-oriented applications. However, in the meantime the Internet has seen a vertiginous growth in the number of hosts and users, and unfortunately, also a growth of misuse and attacks to networks and the need to protect them. One of these countermeasures has been the extensive use of firewalls [Tanenbaum, 2003] to control in-bound and out-bound traffic to/from a protected network.

So far, firewalls and CORBA applications have coexisted quite well since the latter are usually deployed either in a single administrative domain, or only among domains of partner institutions/companies. However, problems occur when they are required to cross network barriers. The reason is that firewall/NAT crossing conflicts with following two CORBA features: location transparency and peer-based communication model [Group, 2004b].

The first feature allows clients to be unaware of the exact localization of a server object when making a remote invocation to it. These server objects can change their location without breaking existing references to them. Although this feature can be seen as a simplification from the viewpoint of the application developer, for firewall traversal this is a great problem since firewalls usually

* This work is supported by a grant from CNPq, Brazil, proc. # 550094/2005-9.

R. Meersman and Z. Tari (Eds.): CoopIS/DOA/ODBASE 2005, LNCS 3761, pp. 927–944, 2005.

control in-bound traffic through a set of rules controlling which address-port pairs are allowed to be reached. To enable a change of server object location, the firewalls rules would have to be updated, so that the permission to cross the network boundaries would work.

The peer-communication model also conflicts with deployment of firewalls because it incurs into a high number of servers that consequently would require a great number of firewall rules. However, this turns out to be an unacceptable administrative burden, and would also significantly reduce the communication performance across the firewall. With CORBA this problem arises when there are a high number of ORBs deployed in the internal network (objects inside a single ORB often share a pair address/port).

Yet another problem related to inter-domain CORBA applications happens when the internal network uses NAT [1] [Tanenbaum, 2003]. This service creates an isolated IP address space inside the network and prohibits these addresses to be used in the external network. When an internal host has to send messages to the outside network, an element (usually the firewall) maps their local addresses to a few public addresses belonging to the administrative domain of the organization. The problem arises when a CORBA object that resides in a NAT network exports its IOR [Henning and Vinoski, 1999]. In the IOR the NAT address specified at IIOP profile doesn't make sense in the external network since it is not a public address, and cannot be used for routing.

This paper describes our work attempting to address the problem of firewall/NAT traversal by CORBA applications, which was driven by the following implicit requirements of acceptable solutions:

- it must not burden the firewall administration (if possible even do not require any configuration);
- as much as possible, it must be easy to configure and be transparent to the application developer;
- it must not have significant negative impact on the application performance.

The work is inserted in the scope of InteGrade Project [Goldchleger et al., 2003] which is a middleware for grid computing. It uses CORBA as middleware and needs the firewall/NAT traversal capability.

In our work we proposed, implemented and evaluated three possible solutions to this problem, where each of them assumes specific situation (or degree) of firewall configurability. The remainder of the paper is organized as follows: Section 2 presents the three proposals and Section 3 discusses implementation details of each proposal. Section 4 contains our evaluation of the three approaches and presents some results of our tests analyzed. In Section 5 we survey related work in this topic, and compare them with the proposals presented here. Finally, in Section 6 we draw some concluding remarks and future work.

[1] Network Address Translation.

2 Firewall/NAT Traversal Proposals

In networks protected by firewalls/NAT, distributed applications can face different scenarios of access permissions to the outside network. These can range from the plain possibility of opening TCP connections to the outside network to the total prohibition of establishing any kind of connection except HTTP connections through a proxy.

Similarly, network security policies (and the negotiation willingness of network administrators) vary a lot. In some cases, it is relatively easy to get a single port at the firewall opened, but in many cases such concession is extremely difficult.

In the following we describe three approaches for CORBA firewall/NAT traversal, each of them being suited to a different level of network access permission. The idea is that the application deployer should be able to choose the most appropriate alternative and configure her application accordingly using an XML configuration file (described in Section 3.1).

2.1 OMG Approach

The first approach is based on the OMG specification for CORBA firewall traversal [Group, 2004b] presented at Section 5.1. It addresses the situations where the server object is not allowed to receive connections from the outside network, or stated the other way round, the client ORB is disallowed to create connections to any host in a *foreign* protected network.

This approach requires a single port to be opened at the firewall, which is used to drain all the IIOP traffic crossing the network boundaries. Within the protected network, all this in-bound traffic is directed to an application proxy [Group, 2004b], which listens at the opened firewall port and is responsible for forwarding these messages to the intended host. Similarly, all out-bound IIOP traffic is first routed to the application proxy which will use the opened IIOP port to send the messages to its final destination. Hence, a single port needs to be opened at the firewalls, which can be shared by several applications (Figure 1).

When a server object is to be deployed, it must somehow advertise the existence of its application proxy. According to the specification, this is done through a tagged component which is to be added to the IIOP profile of the object's IOR and which contains references to all intermediaries (i.e. proxies) between the external network (e.g. Internet) and the object ORB, including itself. Thus, in the simplest and more common case, it would include only a reference to one application proxy and the ORB. When the client ORB receives this IOR it has to identify this tagged component and create a GIOP NegotiateSession message mentioning all the elements between itself and the server object ORB, including the server object ORB and any other intermediary, excluding itself. This message is then sent sequentially to all of the intermediaries mentioned in the IIOP profile, from the first to the last before the server object (i.e. the server application proxy). If the message succeeds in reaching this last element, it will send a reply to the client ORB announcing that the client is allowed to send normal GIOP messages (Request, Reply, ...).

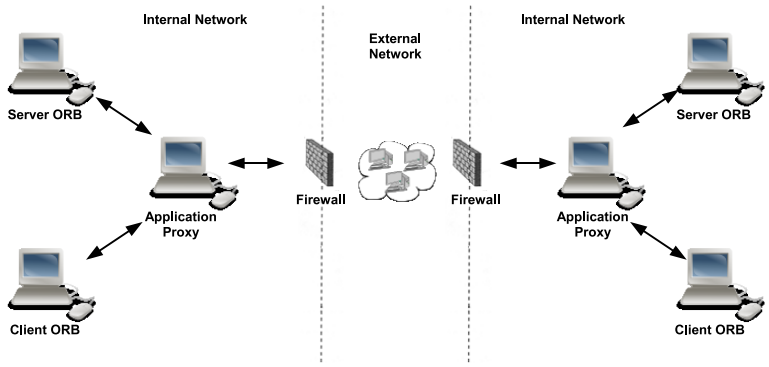

Fig. 1. OMG Approach

In addition, in this approach both the server and client must inform their ORBs of the existence of an eventual application proxy. This is done through a XML configuration file (see Section 3.1) which contains all information needed to build the tagged component and the GIOP NegotiateSession message, by the object ORB and the client ORB, respectively.

This solution has the advantage enabling the interoperability among different ORBs that follow the OMG standard. However, the major drawbacks are the need of a firewall configuration and that the client ORB must be able to properly recognize and handle these new IIOP elements added by the specification.

2.2 TCP Proxy Approach

This approach is intended for server objects that cannot receive connections from elements located outside network, but which may create TCP connections with them.

The main idea is to deploy a proxy (from now called *TCP proxy*, due to the specific transport protocol used) at the outside network and make the object ORB within the protected network connect to it at startup and keep this connection as long as it wishes to remain reachable from the outside network. The object ORB sends a registration message to the TCP proxy for each CORBA object exported, and receives an IOR to be published. This IOR contains a proxy's endpoint, and every client using this IOR contacts the proxy as if it were the object ORB. The proxy then forwards the request to the destination ORB through the TCP connection opened at ORB startup. Through this connection it also receives the replies for the requests, and forwards them to the intended client ORB (see Figure 2). In this figure, the arrows labeled with 1 show the connections opened by object ORBs at startup, while the arrow labeled with 2 represent the connection made by a client ORB after it has received the object's IOR.

Two kinds of connections to the proxy are made by the object ORB. The first one is a short-lived connection created when the ORB needs to register an

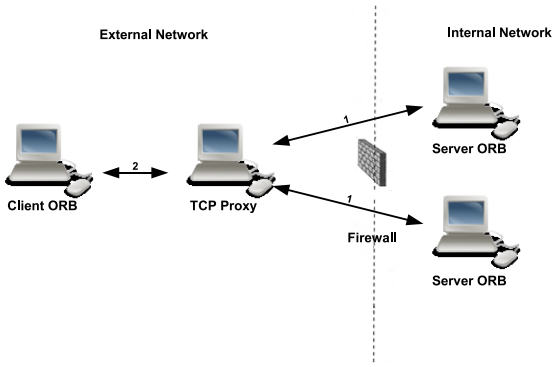

Fig. 2. TCP Proxy Approach

object at the proxy. It is used only for sending the registration message and receiving of the reply containing the IOR to be published.

Hence, during the ORB lifetime many such connections may be made. The second type is a long-lived connection that is made after the first successful object registration. It will be used to receive/send the requests/replies for the objects that have already been or (will be) registered. After the initial handshaking, normal GIOP messages are sent through this single connection shared by all objects. In what follows, we will call them *registration connection* and *data connection*, respectively.

Apart from the fact that a registration must be done at each CORBA object creation, the remaining processing is transparent to the object ORBs: after the the data connection is opened, all request processing is done as if this connection were a normal ORB listen connection with normal GIOP messages passing through it. From the client ORB perspective this is also totally transparent, and no modification whatsoever is required.

An important issue concerns the request IDs used in GIOP Request and Reply messages. According to the CORBA specification [Group, 2004a] this field is used to match requests and replies messages sent over the same connection, and it is the client's responsibility to assign ids correctly in order to avoid ambiguities, e.g. ids for requests which have been canceled, or which have pending replies, must not be reused. Since in this approach, the proxy plays the role of a client of the object ORB and forwards requests, request ids sent by the original clients cannot be considered. This is because these ids will be sent over the same data connection (between the proxy and the object ORB) and may cause ambiguities both at the server ORB and the proxy. Hence, the proxy must generate new request IDs and keep a mapping between the original and the new IDs to replace the new ID with the original ID in reply messages forwarded to the corresponding client ORB.

The main advantage of this approach is that it does not require any firewall/NAT configuration, as long as a connection to the outside network can be made and sustained. Other positive points are the transparency to clients and the ease to modify the object ORB to support this approach (i.e. once the data

connection is open, it is treated as a trivial GIOP listen connection). The main drawback is its lack of scalability at the proxy, as a connection must be kept with each object ORB, but the number of connections are usually limited by the underling operational system.

2.3 HTTP Proxy Approach

The third approach serves server objects in protected networks that can neither get connections from elements at the outside network, nor create TCP connections with them, but where the only allowed connectivity to the external network is by using HTTP protocol.

Similarly to the TCP Proxy (cf. Section 2.2) in this case a proxy (HTTP Proxy[2]) is also deployed on the external network which will be contacted by the client ORBs as if it were the CORBA object owner. Because the object ORBs have the above mentioned access restriction, all its communication with the proxy will use polling to the HTTP Proxy. Thus, in this approach object ORBs send messages encapsulated in a HTTP request and receive the corresponding HTTP reply containing the data requested by the encapsulated messages.

The first thing an object ORB must do is to register one or more CORBA objects at the proxy by sending a HTTP request containing these registration messages and receiving a HTTP reply with the object's IORs to be exported. The IOR created by the proxy will contain its (proxy) endpoint so that the requests can be directed to it. When the proxy receives requests, it will store them and wait for another HTTP request from the object ORB that registered the CORBA object. Once this HTTP request arrives, the stored GIOP Requests will be piggybacked on the HTTP reply sent to the ORB. In addition to object registration request, the object ORB can also send three more types of message: Remove, Reply and Polling. The first is a request to remove an object registry at the proxy; the second is intended to carry a GIOP Reply of a previously received GIOP Request. The third kind of message is used to ask the proxy whether it has some GIOP Requests messages stored for the object. The Polling message thus aims at implementing a periodic and continuous inspection of newly arrived GIOP Requests. The polling periodicity (i.e. time interval) used by the ORB must be set in the configuration file (cf. Section 3.1).

The proxy can send the following two types of messages to the object ORB: RegisterReply and Request, where the former one acknowledges a previously received register message, while the latter contains a GIOP Request received by the proxy from a client ORB.

The HTTP messages can hold any number of encapsulated messages: a HTTP request can have in its body any combination of the four previously described message types, and the HTTP reply may have any combination of RegisterReply and Request messages. This has the advantage of reducing the number of HTTP messages exchanged between the proxy and the object ORB and of reducing the latency of the GIOP request handling.

[2] The name HTTP Proxy is due to the use of HTTP as transport protocol.

As with the TCP Proxy approach, the HTTP Proxy also cannot directly use the request IDs sent by the client ORBs and will generate new request IDs and map them to the original IDs in other to properly forward the replies.

The main advantages of this approach are that it neither requires firewall/NAT configuration (only the "usual" HTTP connectivity), nor any specific client ORB configuration. Of course, its main disadvantage is the HTTP polling. Depending on its periodicity polling either causes waste of network bandwidth, or decreases the applications' responsiveness. Yet another disadvantage is that it requires more adaptations than the TCP Proxy Approach at the object ORB level (see Section 3.4).

3 Implementation

In this section we discuss some issues related to the implementation of the approaches described in Section 2. In our implementation we used a lightweight ORB called OiL [Cerqueira et al., 2005] that is written in the scripting language Lua [Ierusalimschy, 2003] [LUA, 2005]. Like OiL, all the proxies are written in Lua. We start describing the XML configuration file, which is common to all approaches, and then explain the implementation of each approach in some detail.

3.1 Configuration File

In order to deploy a CORBA application that should be able to traverse firewall/NAT, the application developer must write a configuration file in XML. The ORB is informed about the configuration file path through the global variable `FIREWALL_TRAVERSAL_CONF_FILE`.

The XML file has a root element called `firewall-traversal` with a single mandatory attribute called `choice`. This attribute indicates which approach is to be used, and can assume the values `omg` or `proxy`, where the latter represents both the TCP and the HTTP proxy approaches. If the OMG approach is chosen, child elements `inbound-path` and `outbound-path` will indicate if an in-bound or out-bound (or both) path need to be traversed, and their child elements will describe each host of the corresponding path. If `proxy` has been selected, a mandatory child element named `proxy` defines the kind of proxy (TCP or HTTP) and other parameters, such as host address, port and pooling interval (in case of a HTTP proxy), as shown in the following example of a HTTP proxy configuration file.

```
<?xml version='1.0'?>
<!DOCTYPE firewall-traversal PUBLIC "Firewall Traversal
- LAC PUC-Rio" "http://www.lac.inf.puc-rio.br/~theophilo/
firewallTraversal/firewallTraversal.dtd">
<firewall-traversal choice="proxy">
<proxy type="http" address="145.72.24.240"
       port="10012" polling-interval="1"/>
</firewall-traversal>
```

3.2 OMG Approach

To implement the OMG Approach, (cf. Section 2.1), both the server and client ORB required some modifications so that they were able to handle the new data structures defined by OMG. Moreover, an application proxy had to be developed in order to enable access to CORBA objects from the external network.

ORB. As explained in Section 2.1, the traversal information is carried in each IOR through a tagged component in the IIOP profile. Since the tagged component sequence (IOR field where the tagged components are stored) was defined in IIOP version 1.1, the ORB must support it and also be able to handle the elements defined in [Group, 2004b]. Since OiL didn't have this support from the beginning, we had to incorporate this feature in this ORB.

In the server-side ORB the only required modification was the creation of the tagged component in the IIOP profile of the IOR. At each CORBA object creation the ORB checks for any firewall traversal option and whether the OMG Approach has been chosen. If this is the case, it reads some information from the configuration file and uses it to create the tagged component. After that, all the further processing is done as usual, and the server ORB will not be able to distinguish if a request comes directly from a client or through the application proxy.

The client-side ORB has to be modified as follows: before sending a remote invocation message, it first has to check if the object's IOR contains a tagged component for firewall traversal, and if this is the case, a NegotiateSession message has to be sent to the first element between the two. Only after a successful reply, the normal request message can be sent. To avoid the building of the path between the client and the server at each remote invocation, the client-side ORB maintains a cache, saving the time of the IIOP profile and XML file configuration analysis.

Application Proxy. The application proxy was developed with the purpose of supporting multiple concurrent clients and not blocking on any I/O operation, i.e. avoiding the blocking at any accept or receive operation [Stevens, 1998]. The concurrent handling of clients was implemented using Lua coroutines [de Moura et al., 2004] [Ierusalimschy, 2003]. Figure 3 depicts the coroutines within the proxy and their creation order: a main coroutine called *dispatcher* is responsible for choosing the next coroutine to be executed, which also creates and starts a second coroutine called *listenClient* which is responsible for listening to new connections from client ORBs. Whenever a new connection is accepted the *listenClient* coroutine creates and starts a new *treatClient* coroutine, which will be responsible for handling all the communication between two ORBs, after which it will be destroyed.

Both the *listenClient* and the *treatClient* coroutines return control to the *dispatcher* whenever they block at a network I/O operation. It is the *dispatcher*'s responsibility to resume each such coroutine as soon as data arrives at the corresponding connection.

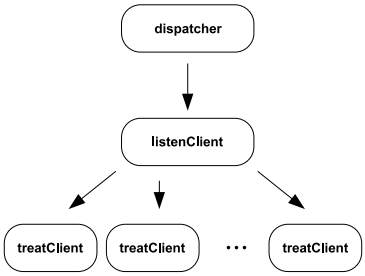

Fig. 3. Application Proxy Coroutines

Problems Encountered. When developing the OMG Approach we encountered some problems due to the omission of some points in the OMG standard. The first problem is related to the GIOP NegotiateSession message. Although the firewall traversal specification mentions its existence, its definition is a reference to a yet unpublished section at [Group, 2004a]. Since the IDL definition of this message could not be found, we created our own simple one, shown below:

```
struct NegotiateSession_1_3 {
  IOP::ServiceContextList service_context;
}
```

The other omission problem relates to the existence or not of the service context entry `FIREWALL_PATH` in the GIOP messages Request and Reply. The existence of this entry (created and used at NegotiateSession message) in these messages would permit the reutilization of already opened connections among proxies.

3.3 TCP Proxy Approach

To implement TCP Proxy Approach (cf. Section 2.2) we had to modify the server ORB and to develop a TCP proxy. As this approach is transparent to the client, no changes in its ORB were necessary.

ORB. The only modification required on the server-side ORB was to make it register newly created CORBA objects at the proxy. At the first time (i.e. first object registration) a data connection is also initialized and inserted in the list of connections used for GIOP message listening. Thereafter, all processing is done as usual. In order to give the application developer access to the object's IOR exported and used at external network, a new method called `_get_ior_exported` was included to the servant returned by the ORB.

Proxy. Like the OMG Application Proxy (Section 3.2), the TCP Proxy was also implemented to handle concurrent requests using Lua coroutines. Figure 4 shows the different types of coroutines and their creation order. Here there is also a main coroutine called *dispatcher* which is responsible to schedule the execution of the other coroutines. Its first action is to create two auxiliary coroutines

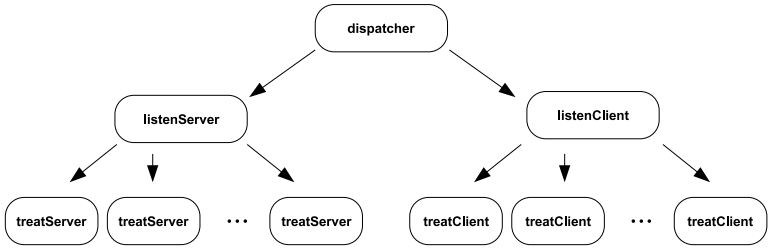

Fig. 4. TCP Proxy Coroutines

called *listenServer* and *listenClient*, responsible for listening to new server and client ORBs connections requests, respectively. Whenever a new connection is established, it creates a new coroutine (*treatServer* and *treatClient* coroutines), responsible for handling the communication with a server or a client ORB, respectively.

The *treatServer* coroutine is in charge of handling either a server ORB request to register an object at the proxy, or to open a data connection through which GIOP messages will pass. In the first case, the coroutine is destroyed after the registration is done. In the latter case, the coroutine remains during the lifetime of the data connection, listening to new GIOP Reply messages and forwarding them to the correct client.

On the other hand, the *treatClient* coroutine just waits for GIOP Request messages, manipulates and forwards them to the correct server ORB. It is destroyed when either the client closes the connection or sends a GIOP CloseConnection message.

3.4 HTTP Proxy Approach

As the TCP Proxy Approach, in this one we have needed to modify the server side ORB and to build the proxy to validate the solution.

ORB. The server side ORB modifications required in this approach are more complex than the ones demanded in the TCP Proxy approach. At each CORBA object creation, a registration must be done using HTTP protocol and the one defined in the Section 2.3. The complexity comes out at the inspection for new requests. In the TCP Proxy approach it suffices to add the data connection opened to a list of connections that would go on a select operation [Stevens, 1998] due to the fact that only GIOP messages are exchanged. Now a different approach must be done because requests may arrive encapsulated in HTTP reply messages.

The original request listening process of OiL is very simple: the server application calls a function called `lo_handleRequest` that will listen for one GIOP Request and reply it. If the application want to treat a infinite number of requests it can put this call on an infinite loop. At ORB level, this function calls a select operation on a list of connections to get a single request. This is the point that must be modified: now the select operation on normal listening connections

must be interleaved with HTTP polling to the HTTP Proxy. The polling interval specified at configuration file (Section 3.1) must be honored and because of this the select operation must have a timeout based on it. Other important point to be noticed is that this HTTP polling can bring more than one request, which compels the ORB to store the additional requests in order to be used in subsequent invocations of the `lo_handleRequest` function.

Proxy. The HTTP Proxy built to validate the solution is very similar to the one developed in the Section 3.3. The architecture presented in Figure 4 is the same of the HTTP Proxy: there is a coroutine responsible for the scheduling of the others (*dispatcher*); two coroutines responsible for server and client listening (*listenServer* and *listenClient*); and others coroutines responsible for the treatment of a single server or client (*treatServer* and *treatClient*).

4 Validation

Once implemented, we evaluated and compared the three approaches by making some performance and scalability tests. For those tests, we run the ORB clients on PCs (Intel Pentium IV 2,80 GHz with 1 GB RAM and running Linux) hosted in two of our Labs (i.e. university sub-networks), and where each client made invocations to a different server object. The server objects were executed on 13 machines (Intel Pentium IV 1,70 GHz with 256 MB RAM and running Linux) of our graduate student lab (LabPos), which is protected by a firewall blocking in-bound connections. The two sets of machines are interconnected by 100 Mbps network with three routers between them.

4.1 Delay

In the first test, we measured the round-trip-delay incurred by each approach when invoking firewall-protected objects. We created five firewall traversal configurations, one for testing both the OMG Approach and the TCP Proxy Approach, two for the HTTP Approach with polling intervals O and 1 second, and finally, one with without a proxy, i.e. by configuring the ORB port manually at the firewall. This fifth configuration, called Direct, was used as a reference measure, to evaluate the delay introduced by each proxy approach. For each configuration, we made approximately 5.000 invocations, and obtained the results shown in Table 1, where columns *Lower*, *Higher* and *Mean* show the lowest, highest and the mean values (in seconds) of the invocations.

As can be seen from the data, the fastest approach was the TCP Proxy Approach, whose delay on average has only 0.0022 seconds (59%) higher, and in some cases even lower delay than the Direct configuration. Compared to the OMG Approach, the main advantage of the TCP Proxy is the smaller number of TCP connections required, and the fewer number of messages exchanged at each invocation. Regarding the number of connections, in the OMG Approach, a connection between the server object ORB and the proxy has to be established

Table 1. Invocation Delay Tests Results

	Measures			
Approach	Lower	Higher	Mean	Std. Deviation
Direct	0.0023	0.0697	0.0037	0.0017
OMG	0.0083	0.3707	0.0351	0.0518
TCP Proxy	0.0036	0.0605	0.0059	0.0011
HTTP Proxy (0s)	0.0073	0.0384	0.0122	0.0024
HTTP Proxy (1s)	0.8762	1.0235	1.0127	0.0139

while in the TCP Proxy Approach such connection has already been opened before at object registration (see Section 2.2). One can say that the TCP Proxy Approach trades scalability (i.e. less number of simultaneous object handled by the proxy due to a limited number of connections allowed) for a better performance. Concerning the number of messages the OMG Approach is also less efficient than the TCP Approach, since the GIOP protocol requires Negotiate-Session messages to be sent and received between the client and the application proxies. One should also notice the larger variation of the delay in the OMG Approach, expressed by the standard deviation.

Even the HTTP Proxy Approach, using the 0s polling interval, has slightly superior performance than OMG Approach, and presents less variation of delay. However, one should not forget that the HTTP Approach has a greater impact on network bandwidth due to the HTTP polling process.

In spite of the TCP Proxy Approach presenting up to 65% higher invocation times, when compared with the Direct configuration, the absolute value is still quite low. Moreover, for larger GIOP messages this overhead tends to become even smaller, for example, if the remote method had more than just an integer parameter and an integer result, as with our tests. Actually, we believe that more tests would be necessary to evaluate how this overhead varies with the kind of method being invoked.

4.2 Scalability

This test suite aimed at measuring the scalability of the approaches, i.e. how the delay caused by the proxies varies with the number of simultaneous pairs client/server interactions. The test scenario consisted of following configuration: the server objects were deployed on the 13 machines (one object for each machine) of our graduate student lab sub-network (LabPos), which is protected by a firewall and connected via one CISCO 7204 router to the network of our other research lab, where the clients were deployed on a 54-node cluster (Intel Pentium IV 1,70 GHz with 256 MB RAM and running Linux). Each of the clients executed on one cluster machine and made remote invocations to one of the servers. We measured the mean round-trip delay for up to 54 simultaneous client invocations (1 to 54 clients accessing 13 servers uniformly distributed) and the results of this test are depicted in the Figure 5.

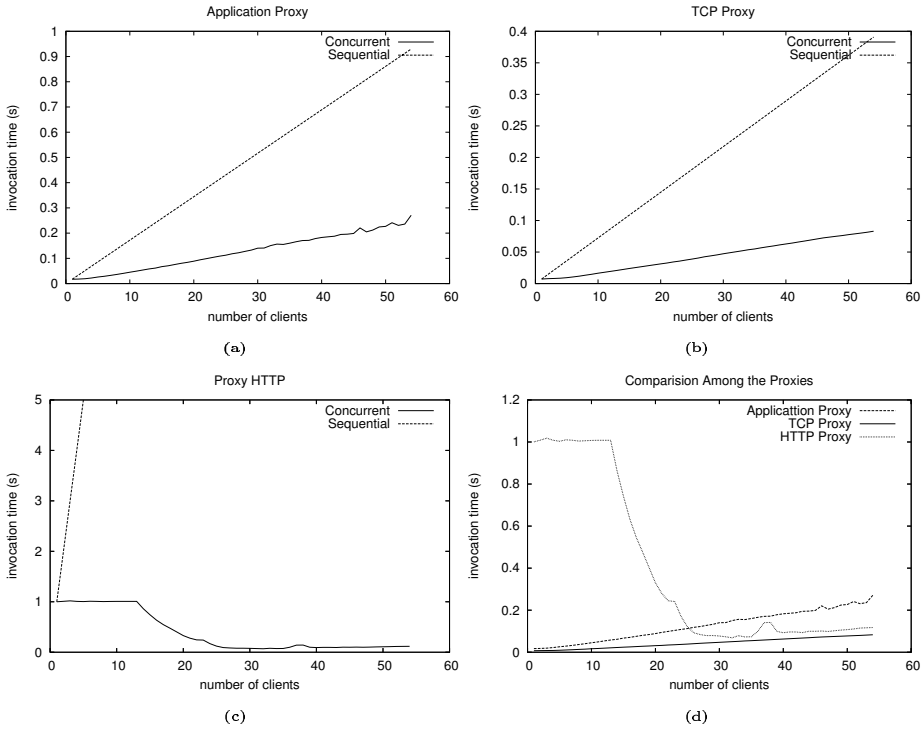

Fig. 5. Scalability Tests Results

The graphics of Figures 5.a, 5.b and 5.c show the delay measurements for the proxies of the OMG, TCP Proxy and HTTP Proxy Approach, respectively (the latter using a polling interval of 1 seconds). Each graphic compares the results obtained with a hypothetical sequential proxy, i.e. a proxy that handles client requests one at a time. As expected, in all approaches the concurrent version shows a better performance than the sequential processing. But it is interesting to notice also the reduction of the invocation delay of the HTTP Proxy with the increase of the number of clients (Figure 5.c). This starts when the client number is 14, and can be explained by the piggyback feature used in HTTP: once there is more than one client per server (at 14 clients) the client requests starts to arrive at the servers not by a polling reply, but as piggybacked on a GIOP Reply sent over HTTP (see Section 2.3). This seems to be the main cause of the delay reduction. The small delay remains until the proxy starts to become saturated with requests, which in our tests happened with approximately 32. Since in our tests we had 13 servers and a variable number of clients, the curve seems to indicate that the lowest delay is obtained when the client/server ratio is between 2 and 2.5. However, also here we believe that more tests should be made in order to obtain a better understanding of the HTTP proxy saturation behavior.

The Figure 5.d compares the delays with the three proxies, showing that the TCP proxy offers smallest increase of delay, albeit the usual server-side connectivity limitations. However, the small difference between the delay increase of the HTTP and the TCP approaches for large number of clients suggests that the HTTP approach is a reasonable alternative when the server ORBs are not allowed to open out-bound TCP connections. Moreover, the results suggest that the HTTP Proxy is better already than the OMG Approach when the client/server ratio is 1.9 or more.

5 Related Work

This section briefly describes related work on firewall traversal for CORBA applications, some of which inspired the present work.

5.1 OMG CORBA Firewall/NAT Traversal Specification

In 2004 the OMG published the CORBA Firewall Traversal Specification [Group, 2004b], which was used as the basis for our *OMG Approach* (Section 2.1).

The standard's basic idea is to extend the GIOP/IIOP protocol with data structures that enable server objects to provide information to clients and proxies on how to open connections to reach them. According to the spec, server objects that want to be reachable by external clients have to put a firewall traversal component in their IOR's tagged components sequence [Group, 2004a]. This is a data structure that contains information about the endpoints of all hosts between the server ORB (inclusive) and the external network (e.g. Internet) which are be addressed in order to make the traversal. When a client gets such an IOR it has to identify the firewall traversal component and build a GIOP NegotiateSession message, which has a service context entry also holding the information about all the hosts on the path between the client and server ORB. The client then sends this message to the first element in the path, which then gets forwarded by the hosts to the next ones on the path, until it reaches the last proxy before the server ORB. If it arrives there, this last proxy positively replies the NegotiateSession message to the previous proxy, which is then forwarded back and all the way along reverse path. When this reply arrives at the client the GIOP Requests/Reply messages can be exchanged normally.

This specification also provides others features, such as the support for secure transport protocols, which so far we have not handled in our work.

5.2 JXTA

The JXTA Project [Brookshier et al., 2002] [Gradecki, 2002] [JXTA, 2005] is a open source worldwide project created by Sun Microsystems intended to offer an infrastructure for peer-to-peer application development. It consists of a set of XML-based protocols that provide system and programming language

independence. The JXTA peers form an ad-hoc network and information is exchanged using some peers (Rendezvous and Relay peers [Brookshier et al., 2002]) as providers and routers of the network.

JXTA claims to offer firewall traversal to applications that use it as the communication layer. The solution offered is a set of public peers (Relay Peers) accessible through HTTP protocol. After the protected peer registers itself at one Relay Peer, the JXTA advertisement and routing engine will route the messages addressed to the protected peer to the Relay Peer at which it registered. This Relay Peer stores the messages until the protected peer contacts it through HTTP, using the fact that out-bound connections using this protocol are generally allowed by firewalls. The protected peers have to periodically make such inquiry (called HTTP polling) which is used both to check for messages at the Relay Peer and to send messages to the JXTA network.

At first, we took into consideration the idea of replacing the ORB communication layer by JXTA in order to traverse firewalls. However, our preliminary tests showed that the delay and high frequency of failures caused by the use of JXTA network were unacceptable. In face of this, we decided to borrow its idea of HTTP polling and provide it as an alternative for the scenarios in which no firewall configuration is possible at all, and TCP connections cannot be established without using the HTTP protocol (see Section 2.3).

5.3 JacORB

JacORB [JacORB, 2005] [Brose, 1997] is a full-featured Java implementation ORB. It provides firewall traversal through a service called *Appligator*, which is a GIOP proxy supposed to be deployed inside the protected network and to have a firewall port opened to it. It is similar to the OMG Approach described in our work, but the JacORB online manual doesn't make clear if the external client has to be configured in order to become aware of Appligator's existence, or if the latter modifies the server object's IOR in order not to require the client configuration. If the client has to be configured, then the OMG Approach is a better option regarding to interoperability. Moreover, JacORB's Appligator does not provide a solution for the situation where the firewall configuration is not possible.

NAT support is offered through an application called `fixior` that patches the Appligator's IOR, inserting the firewall endpoint.

5.4 ICE

The ICE - Internet Communication Engine [Henning, 2004] is a communication middleware similar in concept with CORBA, but not CORBA-compliant. It provides a solution for firewall/NAT traversal through a service called Glacier [Henning and Spruiell, 2005] that may also be used as the firewall of a network. Similarly to our work, ICE doesn't require any application code modification, but only a configuration file. The main idea is to configure both the client and the server to use this service, which will work as a broker between both, acting

as a client to the server, and vice-versa. It also requires firewall configuration, and the client has to be aware of the Glacier's existence.

5.5 Xtradyne

Xtradyne is a company that has a commercial product called I-DBC (IIOP Boundary Controller) [Technologies, 2005], that offers firewall/NAT traversal to applications. In fact, it is a firewall, and its solution is to patch the IOR with an endpoint opened for IIOP traffic so that client invocations are re-directed to the firewall, which in turn contacts the server. Since there is few technical information available, it is not clear how its protocol between the protected application objects and the firewall works, in order to map the outside requests to the intended recipients. An interesting feature of this firewall is its ability to identify IORs sent as CORBA parameters and to modify them accordingly in both directions.

This solution also requires firewall configuration, and does not require client awareness of the firewall's presence.

6 Conclusions and Future Work

This work has presented the design, implementation and evaluation of three solutions for CORBA-based application firewall/NAT traversal. Through several tests we have demonstrated their viability and also discussed the suitability of each approach for specific degrees of firewall/NAT permeability.

In spite of being less efficient than the TCP Proxy Approach, and in some scenarios also worse than the HTTP Approach, the solution based on the OMG standard demonstrated to be a practicable approach, specially when the client is not allowed to make an out-bound connection. An interesting future work on this approach is to support secure transport protocols, as already defined by the OMG.

The TCP Approach turned out to be the most efficient solution developed, and also the easiest to implement since it required the smallest number of server ORB modifications. Its premise - that out-bound connections are allowed - is quite common, and as no firewall configuration is required, this approach represents a very good alternative. For the cases where high performance is a strong requirement, this solution is definitely the best. However, its main drawback is a possibly limited number of simultaneous connections at the proxy. A future work on this approach could be to modify the protocol between the server ORB and the proxy, allowing the latter to terminate unused data connections and enabling the server ORB to recover from this action when desired.

The HTTP Approach gave the most interesting result. It was developed aiming to work in the most restrictive scenario, i.e. where just out-bound connections through HTTP are allowed. Its good performance, due to the extensive use of the HTTP piggybacking feature, has proven this solution to be more efficient than expected, and in some situations even better than the approach based in the

OMG specification. An interesting future work in this direction would be to make it compatible with the presence of Web proxy caches (e.g. Squid [Squid, 2005]) and enable clients in a protected network to use the proxy to send/receive GIOP Request/Reply messages.

As a final remark, it should be clear that more tests need to be done with different message sizes, method parameter types and network configurations, so that we are able to pinpoint the specific implications of each approach on the invocation delay, the network overhead, and the scalability. However, we believe that in any case a customizable, multi-solution, rather a *one-fits-all* approach should be pursued.

References

[Bolton and Walshe, 2001] Bolton, F. and Walshe, E. (2001). *Pure CORBA: A Code-Intensive Premium Reference*. Sams.

[Brookshier et al., 2002] Brookshier, D., Govoni, D., Krishnan, N., and Soto, J. C. (2002). *JXTA: Java P2P Programming*. Sams.

[Brose, 1997] Brose, G. (1997). Jacorb: Implementation and design of a java orb. In *Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, Cottbus, Germany.

[Cerqueira et al., 2005] Cerqueira, R., Maia, R., Nogara, L., and Mello, R. (2005). Oil - (orb in lua). http://luaforge.net/projects/o-two/ (Last Visited in 06/06/2005).

[de Moura et al., 2004] de Moura, A. L., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925.

[Goldchleger et al., 2003] Goldchleger, A., Kon, F., Goldman, A., and Finger, M. (2003). Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. In *ACM/IFIP/USENIX Middleware'2003 Workshop on Middleware for the Grid*, Rio de Janeiro, Brazil.

[Gradecki, 2002] Gradecki, J. D. (2002). *Mastering Jxta: Building Java Peer-to-Peer Applications*. John Wiley & Sons, Inc.

[Group, 2004a] Group, O. M. (2004a). Common object request broker architecture: Core specification. http://www.omg.org/docs/formal/04-03-01.pdf - Version 3.0.3.

[Group, 2004b] Group, O. M. (2004b). Corba firewall traversal specification. http://www.omg.org/docs/ptc/04-03-01.pdf - Final Adopted Specification.

[Henning, 2004] Henning, M. (2004). A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75.

[Henning and Spruiell, 2005] Henning, M. and Spruiell, M. (2005). Distributed programming with ice. http://www.zeroc.com/download/Ice/2.1/Ice-2.1.1.pdf (Last Visited in 06/06/2005).

[Henning and Vinoski, 1999] Henning, M. and Vinoski, S. (1999). *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc.

[Ierusalimschy, 2003] Ierusalimschy, R. (2003). *Programming in Lua*. Ingram (US) and Bertram Books (UK).

[JacORB, 2005] JacORB (2005). Jacorb: The free java implementation of the omg's corba standard. http://www.jacorb.org (Last Visited in 06/06/2005).

[JXTA, 2005] JXTA (2005). jxta.org. http://www.jxta.org (Last Visited in 06/06/2005).

[LUA, 2005] LUA (2005). The programming language lua. http://www.lua.org (Last Visited in 06/06/2005).

[Squid, 2005] Squid (2005). Squid web proxy cache. http://www.squid-cache.org/ (Last Visited in 06/06/2005).

[Stevens, 1998] Stevens, W. R. (1998). *Unix Networking Programming*, volume 1. Prentice-Hall, 2 edition.

[Tanenbaum, 2003] Tanenbaum, A. S. (2003). *Computer Networks*. Prentice Hall, 4th edition.

[Technologies, 2005] Technologies, X. S. I. (2005). Iiop domain boundary controller - the corba an ejb firewall. http://www.xtradyne.com/documents/whitepapers/Xtradyne-I-DBC-WhitePaper.pdf (Last Visited in 06/06/2005).