
DE-FG03-01ER83311

Principal Investigator Svetlana G. Shasharina

April 5, 2002
Table of Contents

Introduction ................................................................................................................................3
Task 1: CORBA 2.3 ..................................................................................................................3
Task 2: Certificate and authorization services .................................................................4
  Certificate services ...........................................................................................................4
  Akenti authorization and administration ......................................................................5
Task 3: Secure communication protocols ........................................................................7
Task 4: Portable interceptors ............................................................................................7
Task 5: Globus GSI .............................................................................................................10
Task 6: Multiple applications ...........................................................................................10
Task 7: CVS .......................................................................................................................11
Conclusions .......................................................................................................................12
Introduction

In Phase I we concentrated on investigating how to bring various security mechanisms into the CoDeveloper collaboration environment. In order to utilize existing CORBA security mechanisms, we started with upgrading our system to CORBA 2.3 standards (Task 1). This brought into our arsenal such technologies as Bidirectional IIOP, Portable Interceptors, and IIOP over SSL. Use of bidirectional IIOP and experimentations with various port policies allowed us to formulate our approach on how to resolve firewall issues (Task 1) and work through Web browsers.

Next we investigated technologies for generating and managing security certificates. This was done by building and testing the OpenSSL and OpenLDAP products (Task 2). In order to implement higher-granularity authorization, we built and used Akenti to prototype a system with a protected resource allowing three different types of use: a stakeholder and two groups of users (Task 2).

For Task 3 we implemented a secure CORBA connection within CoDeveloper by using FreeSSL plug-in for ORBacus, which brought IIOP over SSL, thus providing mutual client-server authentication. Using Portable Interceptors, we developed a prototype client-server system with arbitrary authorization performed at each method invocation (Task 4). To carry out Task 5, we installed Globus toolkit version 2 and familiarized ourselves with the Globus Security Infrastructure.

Finally, in order to demonstrate flexibility of the CoDeveloper architecture and make the application more useful in scientific applications, we have prototyped technical solutions for bringing multiple applications into one collaboration session (Task 6) and implemented concurrent version control, CVS, (Task 7). We give more details on each task in what follows.

Task 1: CORBA 2.3

This task was to port the CoDeveloper to the CORBA 2.3 protocol. Use of CORBA 2.3 would insure that CoDeveloper uses the most OMG-compliant ORBs, has means to implement communications across firewalls and bring in various security mechanisms. Upgrading the code to ORBacus 4.0.5 achieved this goal. This required some changes described below.

The biggest change in the CORBA 2.3 specs (compared to CORBA 2.1 used in the project previously) was the introduction of the POA (Portable Object Adapter). Specifications of the previous object adapter (BOA, Basic Object Adapter) allowed for creation of incompatible server side implementations, and the problem was fixed by better-formulated POA specs. The POA provides such fundamental services as object creation, servant registration, and request dispatching. Use of the POA required changing the client and the server main programs.

Another new feature in our code was use of Portable Interceptors (PIs), which allowed us to experiment with bringing authorization into our framework. Although PIs were first described in CORBA 2.3.1 specification, this first draft was so under-specified that it could not lead to any reasonable implementations. There was a new specification\(^1\) for PIs that was used by most of ORBs implementing them (including ORBacus 4.0.5). Portable interceptors are optional...
mechanisms that allow ORB to intercept the request flow and perform additional actions in the interception points. The use of Portable Interceptors is described in detail in the Task 4 section.

Finally, we extensively used bidirectional IIOP also introduced by CORBA 2.3. Bidirectional IIOP was implemented to address the firewall issues, and we briefly describe our solution in the remainder of the section. The CoDeveloper application is a distributed system of a server and clients. Some of these entities might be behind a firewall. Thus, we had to consider four different situations: a client and a server are on the same side of the same firewall; a client is behind a firewall, a server is not; a server is behind a firewall, a client is not; and client and a server are behind different firewalls. The first situation is trivial and needs to be addressed only because the other three might require complex changes of the server and client code, which might affect the work of the system in the simplest settings. In other words, we needed to provide a solution that will transparently work in arbitrary settings.

The situation is complicated by the fact that our client is intended to work through Web browsers and is implemented as an applet. Applets, which are downloaded to clients behind a firewall, not permitting incoming connections, cannot normally use IIOP to receive callbacks because the client must accept a connection from the server.

Currently there are no interoperable CORBA solutions to this problem. To solve it in our case, we brought bidirectional IIOP into the server and client implementation. Bidirectional IIOP is an enhancement to standard IIOP (standard in GIOP 1.2), which provides an approach alternative to a callback. With bidirectional IIOP, callbacks are sent back over the single, initially established connection, so that applets do not have to receive a new connection. This fixed the problem for applets behind the firewall.

A client needs to be able to talk to a server at its IP and port number, while the server needs to reply to the client’s IP and port number, and firewalls might prevent this from happening. Also, the server might be assigned a dynamic port number, unless it is set to use one number at a startup time. Typical firewalls leave some ports open. For example, the HTTP port (80) is always open. The telnet and ftp ports are open sometimes too. That is why, in addition to using bidirectional IIOP, we concluded that the client and server (each) should keep one port for the CoDeveloper communications. This conclusion was tested in various settings.

**Task 2: Certificate and authorization services**

This task was to implement secure certificate and authorization services along with administrative tools.

**Certificate services**

In order to understand the fundamental certificate structure and storage services, we approached this task by first building and testing the OpenSSL and OpenLDAP products individually on our development Linux server. OpenSSL was used to generate self-signed key pairs to act as our local Certificate Authority. With an administrator allowed access to the signing pass phrase, users could initiate a certificate request via standard openssl commands and mail this to the local CA. The administrator would then sign requests with the CA pass phrase.
and return the signed certificate back to the user. The user would then be responsible for importing this certificate into their web browser. Copies of the user certificates were also held in a local directory structure.

OpenLDAP was then built and installed as a true daemon, running at the default tcp port number 389 on our internal server. This allowed us to create a centralized database of LDAP entries, which could include the signed certificates earlier produced from OpenSSL. We created an LDBM database definition based on our domain name, "dc=txcorp,dc=com" and used this as our initial namingContexts (o=txcorp.com). LDAP entries were then added manually via LDIF files, via the ldapadd command executed by the LDAP Manager, which contained appropriate objectclass assignments and values for our test examples.

Once basic "user entries" were verified with the ldapsearch command, LDIF entry files were then modified to include the contents of original OpenSSL certificates, and subsequently added to the LDAP database. It was becoming apparent that with any sizeable amount of user certificate traffic that invokes manual "command line" methods of maintaining such certificates and databases could become a timely task. Not to mention if off-site replication services or integration points from multiple CAs or LDAPs were required.

For our development authentication test bed, we chose Netscape's Certificate Management (NCM) server for its combination of CA services and LDAP retrieval methods. In addition to automating the interactions between CA and LDAP, this product provides a Web based GUI for managing all servers related to certificate management and storage, as well as allowing end users to request and import certificates from a centralized internal Web site. Simplifying the process of certificate request, transport, signing, storage and eventual import to a users Web browser was efficient progress.

During our configuration of NCM we chose a single LDAP database instance based on "txcorp.com" with additional namingContexts of "NetscapeRoot" and "netscapeCertificateServer", to contain prior examples, Netscape generated administration passwords, and new Netscape certificate requests and signings.

All tests via secure and insecure NCM Web servers proved to be streamlined processes, reliable for all requirements of certificate management and storage/retrieval. Akenti services and certificates, as described below, would be built upon this NCM authentication base.

Akenti authorization and administration

The task of building and configuring Akenti was completed via source code compilation, utilizing Java v1.18 and a variety of Java security modules installed (JSSE, JCE, JNDI). Newer releases of Akenti are compatible with Java 1.3 and will be tested at a later date.

We chose to assign and protect access to a directory structure on our server, calling this our Akenti "TestResource". To this resource we will have three different users, each at different security levels, which would be defined via Akenti Policy, Attribute and UseCondition certificates. The first task of creating a primary Stakeholder was begun with a new user ID issuing a request for a certificate via the local web page of the Netscape Certificate Server. The
CA was then responsible for retrieving the request via NCM, and signing the new certificate. The end user (Stakeholder) is able to check status of the pending request, and once completed retrieve the signed certificate for importing into their web browser for future authentication purposes.

Two different user IDs were then also certified and signed via the same Netscape Certificate Server process and thus stored in the local LDAP database. We will refer to these users as "Local User" and "Remote User", each of whom will be allowed different access levels to the TestResource. Akenti relies on a combination of "servers" and certificate files when assigning attributes and access policies to resources. We chose to follow the Akenti template files included with the source distribution to understand the relationships and sequential processes involved when allocating authorization to a named resource.

The primary configuration file (Akenti.conf) requires a number of modifications based on local directory structures and file names. We chose a RootResourceName of "TestResource" with a full path to a local directory on our server. ServerHostName was obviously also changed to local hostname, with Default PolicyFileName and ResAttrName filenames left at default values to adhere to Akenti standards. With a Stakeholder defined in rootPolicy.xml, this authenticated user now has the ability to assign attributes for TestResource and our examples, Local User & Remote User. We chose to modify a template for attributelocation.xml, and create an attribute name of "Location" with two possible values: "Local" and "Remote". Only Local User was assigned the attribute value of "Local", whereas Remote User was obviously valued as "Remote". Each entry was signed by the same "issuer", Stakeholder. We defined a default set of actions, which were globally applied to TestResource: read, write and execute.

Each of these actions are constrained to an Attribute/Value pair, as configured in the UseCondition certificate, or uc.xml template. This file contains the IssuerDN (Stakeholder) and CADN for signing purposes as well as conditions applied to a resource name. In our case, we implemented two different Condition Constraints for TestResource. Any user who's AttrName=Location value matched "Local" was given rights for all actions: read, write, and execute. A second constraint of the AttrName value matching "Remote" was only given rights to the read "action". The list of trusted Certificate Authorities used to verify each users certificate is also included in uc.xml, this must be a subset of the CAs listed in the overall Policy Certificate.

Converting these XML templates to signed Akenti certificates requires executing the utility CertGen, the resulting files: .htauthority (from rootPolicy.xml) is placed at the root level of our directory tree, UseCondition certificates (from uc.xml files) and Attribute certificates (fromattributelocation.xml) are placed in the "TestResource/cert" directory. The final step in verifying Policy, UseCondition and Attribute Akenti certificates involved a modified shell script which calls the Akenti tool "verifyAkentiCert", validating relationships between new Akenti certificates and corresponding PEM files. The .htauthority (PolicyCertificates) file and Attribute certificates are verified with the Akenti self-signed PEM. While two verifyAkentiCert tests are run for a UseCondition certificate, once against the Akenti PEM, and a second time with the issues (Stakeholders) PEM.
Task 3: Secure communication protocols

The task was to implement secure communication protocols within our application. Our approach to implementing a secure CORBA connection within CoDeveloper was to use the FreeSSL plug-in developed by Object Oriented Concepts, the same company that produced the ORB used in CoDeveloper. The FreeSSL plug-in operates by running the IIOP protocol over an encrypted Secure Sockets Layer (SSL) channel. The SSL protocol, developed by Netscape Communications Corporation, provides communications privacy over a network designed to prevent eavesdropping, tampering, and message forgery. The FreeSSL plug-in enables secure communications via SSLv3 using the ORBacus ORB in both Java and C++.

For use with C++ CORBA code, FreeSSL has C++ libraries that build upon the OpenSSL toolkit from The OpenSSL Project. Whereas, for use with Java CORBA code, FreeSSL has Java archives build upon two tools, IAIK-JCE and iSaSiLk, developed by The Institute for Applied Information Processing and Communications (IAIK) group from the Graz University of Technology in Austria. The first tool, the IAIK Java Cryptography Extension (IAIK-JCE), is a set of APIs and implementations of cryptographic functions, including symmetric, asymmetric, stream, and block encryption methods. It supplements the default security functionality of the Java Development Kit, which itself includes digital signatures and message digests. The second tool, iSaSiLk, is a Java implementation of the SSLv3 protocol, supporting all the common cipher suites except Fortezza. iSaSiLk operates on top of the IAIK-JCE Java Cryptography Extension APIs, but will also work on any alternative JCE implementation supported by a proper provider for supplying the required cryptographic algorithms.

We installed OpenSSL, IAIK-JCE, iSaSiLk, and FreeSSL and constructed a C++ server / Java client test code. It used the FreeSSL and base packages along with hand generated certificates to allow the client to connect via SSL to the server and then subsequently receive callbacks from the server. This was sufficient to demonstrate that the plug-in could be used within CoDeveloper.

Task 4: Portable interceptors

The task was to prototype Portable Request Interceptors (PIs) in CoDeveloper with the idea to bring in authorization into CoDeveloper. We made this task more general by formulating a general PI solution for bringing an arbitrary authorization into an arbitrary client server system. In what follows we discuss this solution.

Client and server side PIs provide interception points into the ORB within the request/reply sequence, through which ORB services can query request/reply data and transfer service context between client and server. In this way they provide a means of structuring an ORB’s interaction with extra-ORB services. The request interceptors intercept the flow of request/reply sequence in well-defined points on the client’s as well as server’s side. The points are such as sending/receiving, polling receive reply/exception, send reply/exception (see Fig. 6).

Our assumption is that authorization can be encapsulated in two local objects: Authority and Credential. The objects are defined by their OMG-IDL interfaces, with the Authority interface defining the authorization method:
// Authority.idl
#include <Credential.idl>

module securityProtocol {
    interface Authority {
        void authorize(in Credential);
    };
};

The implementation of the authorize function depends on the desired security protocol. For example, if we want to use Akenti protocol, this function should be implemented to take the client’s Attribute certificate from the Credential, check the attributes against the user policy certificates, and attest that this client has a right to execute on the server. The Authority object is created on the server side, while the Credential object comes from the client. In fact, since we want each client to produce a unique Credential, it should come from the client’s main program, not from the client side interceptor implementation. The server side is different, since there is one policy per server. That is we created the Authority object in the constructor of the server side interceptor and made Authority a data member of this class.

To prevent the client access to any server objects, one should make the Authority and Credential meet in any of interception points that come before a servant invocation. Since the Authority object originates on the server, the only point where the authorization can take place will be the receive_service_requests point.

In order to make the Credential object accessible in the server interceptor, we had to use PortableInterceptor::Current (PICurrent in what follows). PiCurrent is a table of slots that different services can use to transfer their data to request or reply service contexts. In our example, in the client main, we first create a new Credential, convert it to a CORBA::Any variable and put this variable into a slot with a particular slot number.

![Diagram](https://via.placeholder.com/150)

**Fig. 5.** The concept of request interceptors is to intercept the flow in interception points (5 points on the client side and on the server side).
After that we extract this variable in the `send_request` point of the client interceptor and concert it into a new service context, which makes it available in the server interception points. Finally, in the `receive_request_service_contexts` point we perform the authorization:

```cpp
void TxServerInterceptorImpl::receive_request_service_contexts
(PortableInterceptor::ServerRequestInfo* ri)
throw(CORBA::SystemException, PortableInterceptor::ForwardRequest) {
    // Get Credential as a service context and convert it to correct type
    IOP::ServiceContext_var context =
        ri->get_request_service_context(serviceContextId);
    size_t l = context->context_data.length();
    CORBA::OctetSeq octs(l);
    memcpy(octs.get_buffer(), context->context_data.get_buffer(), l);
    CORBA::Any_var any0 = codec->decode_value(octs, _tc_Credential)
        ri->set_slot(slot_id, *any0);
    const Credential* cred;
    *any0 >>= cred;
    // Authenticate using auth (data member of Authority type)
    auth->authorize(cred);
    // Proceed or throw exception, so that the invocation is not completed
}
```

This example allowed us to formulate the following strategy. To implement authorization using interceptors, we need to implement a library with the following elements:

a. **Authority** – an object that is collocated with a CoDeveloper server, and **Credential** – an abstraction of a credential. We will require instantiation of Authority object for the lifetime of each CoDeveloper server. This object will implement authorization as
defined by the chosen server security policy.

b. A client function, which creates a Credential object upon the client startup and puts into a slot of PICurrent, so it becomes accessible to client interceptors.

c. Functions needed to make these objects communicate as described above: transforming Credential object into CORBA::Any, from CORBA::Any to an Octet sequences and back. This is done using a IOP::Codec object, which is instantiated in the ORBInitialier and passed to the interceptors at their construction.

This library will insure that users of CoDeveloper will not be concerned with implementing security, and the use of CoDeveloper will be protocol transparent. Although this solution referred to Akenti authorization, it is absolutely general and can be used for implementing arbitrary authorization and authentication protocols.

**Task 5: Globus GSI**

The task was to examine the use of the Globus GSI toolkit for secure communications. As a first step, we installed Globus Toolkit™ version 2 on our system. This required creating a local (Tech-X) certificate manager and generating user certificates.

The Globus Toolkit uses the Grid Security Infrastructure (GSI) for enabling secure authorization and communication over an open network. GSI provides a number of useful services for grids, including secure communication (authorized and confidential) between elements of a computational grid, supporting security across organizational boundaries, and "single sign-on" for users of the grid, including delegation of credentials for computations that involve multiple resources and/or sites. GSI is based on public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol. Extensions to these standards have been added for single sign-on and delegation.

In our evaluation, we concluded that the most valuable for CoDeveloper capability of GSI is its support for "single sign-on" and delegation of credentials. This will allow users of CoDeveloper to combine computational, visualization and data resources from different sites and use them transparently.

**Task 6: Multiple applications**

Implementing multiple applications involved modifying several parts of the client, the protocol, and the server. The client had to be modified to learn from the server the names of the applications served by the session. The client GUI had to be modified to allow the user to specify which application is to receive a certain command. For execution of client-edited files, the application is specified through the file extension. This then required one change to the OMG/IDL interface, namely that the execute method must also send over the name of the application to execute the command.

On the server end, the first change needed was to be able to parse in from the input files the multiple applications to be served along with the corresponding file extensions. For each of the applications, it is necessary to create a communication pipe. We then also had to create a mapping from the name to the pipe and from the file extension to the pipe. Next, for each
executable command, we must now use the map from the application name to the pipe to determine which pipe to send the data to and then send the data. For file execution commands, the map from file extension to pipe is used. The remainder of the coding within the server is pretty much unchanged, as the select method allows us to obtain data from any of the return pipe’s, and at that point the middleware python application has converted the results from the application into a universal message, so that the server can then inform the client that the new image or text output is available and so display it.

Task 7: CVS

The task was to integrate the CVS code repository system into CoDeveloper. We modified the server code so that, at the time when a client connects to a session, a CVS repository gets created in a predefined directory of the server. On the client side, a special CVS session menu (shown in Fig. 7) was added to the GUI that allows users to send files to the CVS repository on the server and get updated versions back. The menu choices are made as intuitive and as close to CVS as possible. For example, command Add is supposed to send file test1 to the server, save it there in a designated directory and execute a CVS command, which adds this file to the CVS repository:

```bash
cd ../techechohtml/workdir/sveta_CpFileImage
cvs -d 'pwd'/../../techechoRoot add test1
```

After the client is done with editing the text, she commits the file to the repository. This action takes the changes in the text, sends them to the server and commits the file. Other clients of the same session can update test1 file after that by using “Update” command. This brings the text window on their screens displaying the latest version of the text file. We made this commands to act as close to CVS as possible. Thus, if the file has been committed by anybody else, CoDeveloper does allow the user to commit, but instead it displays a message recommending to update this file first. Thus, several geographically separated users can work on the same document simultaneously: adding or deleting text, committing changes to the common repository and updating local versions as needed.
The mechanics behind the scene is not that trivial. Since these actions involve transmitting data between the client and the server, the corresponding functions are prototyped in OMG-IDL interfaces. As an example, let's consider the “Commit” command. The IDL interface for the server (Distributor.idl) declares the following function:

```idl
void commitTextFile (in string clientname, in string filename,
in string text, in long length) raises (UnknownName);
```

where the first argument is the name of the client who adds the file, the second argument is the file name, the third argument is the text in the file and the fourth is the length of the text. The server implements this method in C++:

```c++
void DistributorImpl::commitTextFile
(const char* clientname, const char* filename,
 const char* text, CORBA_long length) {
    // Body of the function which calls system command equivalent to:
    // cvs -d 'pwd'/../../CoDevRoot add filename
}
```

On the client side, after the client commits, the `commit()` function is called. It reads the name of the client, the name of the top file, the string representing the text in the file and its length, and then passes this information to the instance of the `ChatDialog` class.

The implemented CVS system is useful in that it conserves bandwidth, and it allows a measure of privacy to each collaborator. Each collaborator could further interact with an application, such as a visualization tool, that could generate text or images that would then be shared with all collaborators through the CoDeveloper system.

**Conclusions**

All objectives of Phase I were achieved, and all the tasks were fully completed. These accomplishments demonstrated that use of CORBA-based client-server architecture is suitable for running shared remote applications and capable of implementing high-grain security using existing CORBA mechanisms and implementations. We also have shown that our system is...
flexible to accommodate multiple applications in one session and provide means for concurrent version control.

In addition, we formulated some directions for implementing security in Phase II. We will use premier CORBA implementations supporting IIOP via SSL to provide authentication. In order to utilize the prototyped (with Akenti) system with multi-level user authorization, we will bring it into our general technical solution using portable interceptors and merge it with the CoDeveloper framework.

3 http://www.ooc.nf.ca/fssl/.
4 www.openssl.org.