Chapter 6

Building a Scalable Mediator-based Query System

6.1 Objectives

In the previous chapters, we have introduced our mechanisms for supporting IIOP calls in two CORBA enclaves separated by firewalls, for supporting CORBA callbacks, and for supporting the communication among different distributed environments. In order to let you have a more detailed understanding of our proposed mechanisms, and also to show you how our mechanisms contribute in integrating different distributed systems, we would like to show you how we implement a practical example, a mediator-based query system. It demonstrates how we use our mechanisms to bypass firewalls, to use callback features, and to expand across heterogeneous systems, in order to build a scalable information systems for system integration process.

In this Internet age, people put lots of information on the Internet for others to retrieve. Though there are plentiful information ready for us, we may not be able to query for the contents we need. First, the volume of information is expanding dramatically. Even within an organization, multiple databases are usually employed to store their data. Hence, techniques for searching across a number of distributed data sources are important. Second, information may be provided by various organizations, which means we may need to search across many different sites to obtain the richer information.

In order to solve the first problem, we have established a web-based query system using mediators to search in distributed databases. The mediator is the middleware that forwards user queries to various database engines, and when the database engines searched out the results, it integrates them and returns them back to the users. We will give an introduction to mediators, and describe our system design and implementation in section 6.2. We use CORBA for our infrastructure implementation such that mediators can make queries to various data sources, or even other mediators, within the CORBA enclave.

Although the second problem, that is, making queries to other sites, can be solved by an extension to our mediator system, we need to tackle some technical problems first.

The first problem is the firewall issue. For a local system, we usually have a firewall to protect the computers inside from outside attacks. As we are using CORBA and IIOP cannot pass through firewalls for communication, we try to use Java Servlets, XML and HTTP to simulate object method calls and parameter transmissions in CORBA. By doing so, we can make our system to be more scalable in the Internet with firewalls. This will be discussed in 6.3.

We then demonstrate how we enhance our query system by using the callback feature. We extend our mechanism to use XML and Servlets to perform some interesting features with callback. Section 6.4 will cover this part.

The second problem is that when we need to combine information among heterogeneous distributed environments, we do not have a generic method to do so. Here, we use XML and Servlets again to connect our CORBA-based system with DCOM-based system and JavaRMI-based system. This part will be covered in section 6.5

By doing all this, finally, we develop a simple and generic way to achieve a more scalable query system against firewalls and heterogeneous distributed environments.

6.2 Introduction to Our Mediator-based Query System

6.2.1 What is mediator?

The mediator is the middleware between the clients and database servers, which can solve some deficiencies of traditional client/server systems [33, 34]. The tight relation between client and server may lead to the following problems: First, a server may be dedicated to some clients only; also, clients may need to search a number of servers to obtain what they need, while those servers may be heterogeneous. Mediator is one of the architectures that can meet the need to make data widely available over a distributed environment. Mediators forward client queries to appropriate data sources, and then integrate the answers from different sources, and forward the integrated answer back to the clients. Figure 6.1 is an example.

There are several advantages of using a mediator system:

- Conceptually, all distributed data sources are integrated into a single component even though the data sources are heterogeneous. Hence, clients need not know about the location or other specific information of the data sources.

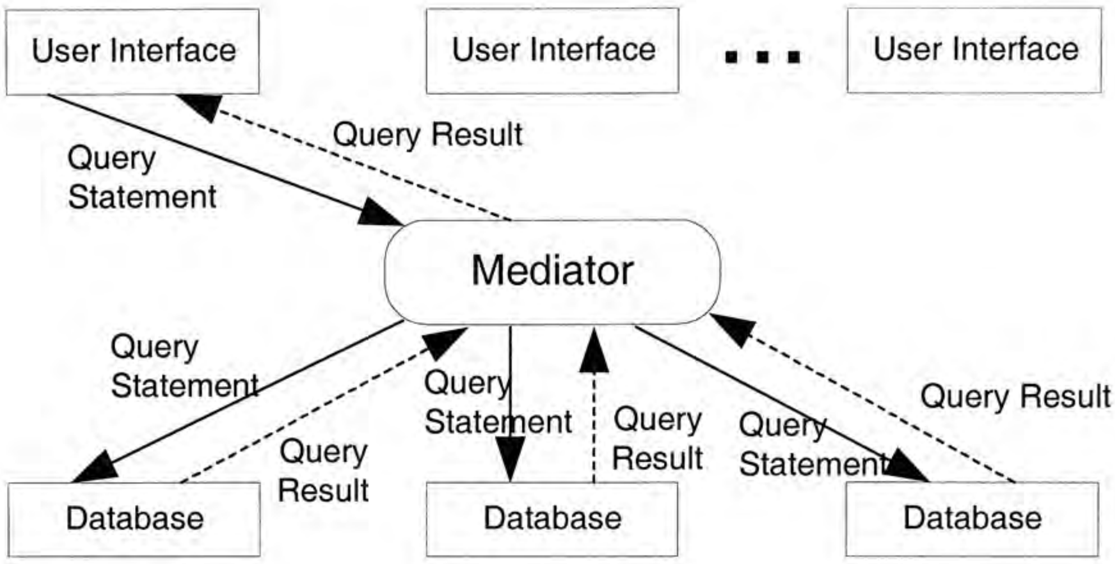

Figure 6.1: Diagram of the mediator concept

- Client programs need not care about the changing of data source locations, and the addition, deletion, or even failure of some data sources.
- The mediators can help the users to choose the most appropriate data sources, based on their queries submitted, to enrich the quality of information retrieval.

6.2.2 The Architecture of our Mediator Query System

Here, we describe the basic architecture of our mediator query system. Our mediator query system is mainly consisted of two components: Query Mediators and Database Query Engines. The design of the architecture of our query system is shown in Figure 6.2. Similar to other mediator systems, the database engines are waiting for the requests from the mediator components. Also the mediator components are waiting for requests from the user interface and upon reception will send these queries to the database engines.

Furthermore, mediators can also send queries to other mediators, which

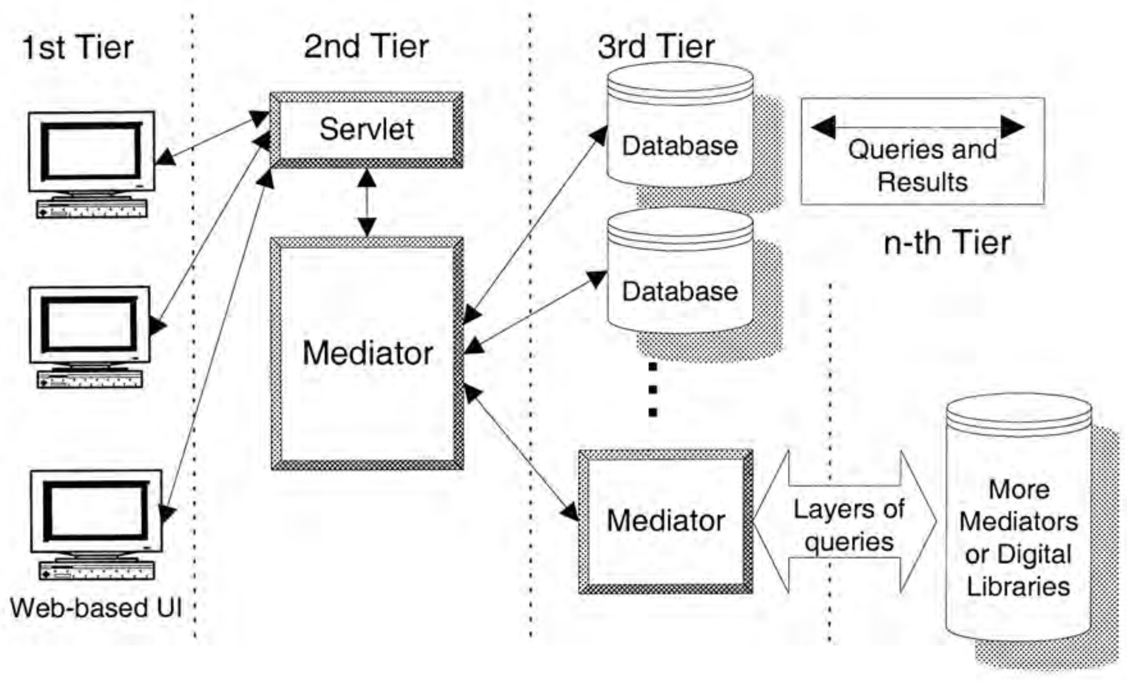

Figure 6.2: The architecture of our query system

may further forward queries again to other database engines or mediators. This mechanism forms an n-tier distributed system. As mediator components have to make queries to both database engines and other mediators, we would like those database engines and mediators to have the same generic interface.

In our system, we also use XML for the internal data representation and storage because it works well in a heterogeneous environment. Hence, we use XML-QL [35, 36] as the query language in the whole system, which is a query language dedicated for XML data developed by AT&T. We use news data obtained from local newspapers in our experiments. They are all converted to XML format.

In the practical application of using mediator architecture in a distributed environment, we need to handle some special cases. One is the infinite looping problem: as a mediator may make queries to another mediator, the queries may be transferred from one mediator to another. Eventually, there may be a case that the mediators have formed a cyclic path and the first mediator is being queried by itself. We need some methods to detect infinite looping. One possible approach is to give each query a unique ID, and all mediators keep track of all IDs of those queries that are already submitted but no replies have been received yet. In case there is an upcoming query with the same ID as any one entry in its record, we can tell that an infinite looping has occurred.

The second problem is to avoid having a long waiting time for users, which may be caused by: the connection between some objects may have been broken, or the number of layers that the queries need to traverse may be too many. For the broken connection problem, we simply use a time-out parameter to specify the maximum amount of time that we are willing to wait. For the too many layers of query traversal problem, we simply use a maximum layer parameter to specify the maximum number of layers that we want to go.

6.2.3 The IDL Design of the Mediator System

We are using CORBA for our system infrastructure. To design the interfaces of different components, we use IDL. CORBA IDL is an interface definition language structures for all concepts of the CORBA object model independent of programming languages. Both Query Mediators and Query Database Engines are implementing the same interface in order to make these two objects the same in the view of the users. In our IDL, we only define a common interface called `QueryEngine`. (See Figure 6.3)

We are supposed to provide to the `QueryEngine` a query statement, and it will return to us the answer in String format, which is a XML expression. We have defined only one simple method in `QueryEngine`, i.e. `query()`, which has a XML-QL statement as its argument, and returns a XML string as the result. This can be used in both Database Query Engines and Query Mediators, such that programmer can notice no difference between making a query on them.

```
module QueryEngineApp
{
  struct SysPara
  {
    long qid;
    long timeout;
    short maxlayer;
  };

  interface QueryEngine
  {
    string query(in SysPara para, in string QueryStatement);
  };
};
```

Figure 6.3: The IDL design of our system

Though they only share the same interface, the implementation of `query()` method would be different.

6.2.4 Components in the Query Mediator System

We rely on CORBA technology for building the system infrastructure because CORBA provides a very good infrastructure for designing and implementing applications in a distributed environment. In order to integrate our system into the web environment, we also use Java Servlet technology. Java is used for our implementation, because of its portability. As we have mentioned before, both the Query Database Engine class and the Query Mediator class are implementing the `QueryEngine` interface. We have named these two classes as `QueryDB` and `QueryMed` respectively.

A `QueryDB` object is directly connected to the data source. A caller can call the method `query()`, and this method will take the query statements (XML-QL statements in our implementation) as the argument and search for the XML document specified, then it will return the result to the caller in a stream of

XML string. We have the `QueryDBServer` object as the server for creating a `QueryDB` object, and registering it to the CORBA name service. The server is also ready to set up multiple threads to support multiple requests on a `QueryDB` object at a time. This server should be started at command prompt.

`QueryMed` object is the Query Mediator which forwards query statements to other mediators or database engines. Its implementation is more complicated than `QueryDB`. Other than the `QueryEngine` interface, `QueryMed` also implements another interface, `QueryMediator`, shown in Figure 6.4. Methods of this `QueryMediator` interface cannot be called by other distributed objects, but can only be called by Query Mediator Server objects, which contain the `QueryMed` objects and located at the same host with them.

```
public interface QueryMediator
{
    public QueryEngineApp.QueryEngine[] qelist();
    public void qelist(QueryEngineApp.QueryEngine[] arg);
    public void append_result(String res);
}
```

Figure 6.4: `QueryMediator`, another interface that `QueryMed` Class implemented

In a `QueryMed` object, the attribute `qelist` would store all the `QueryDB` objects and `QueryMed` objects which it will further search for. And `query()` will start a thread for each `QueryDB` or `QueryMed` object and the thread will take the XML-QL query statement as argument and pass it to its corresponding object in qelist by calling their `query()` method. Then, when all these objects have returned the XML result back to the threads, they will call the `append_result()` method of the parent `QueryMed` object, `query()` will further organize and integrate the results into a single XML file stream and then return it to the caller.

`QueryMedServer` object is similar to `QueryDBServer` object, which will create a `QueryMed` object to handle queries. It will also bind the list of query engines (`QueryDB` and `QueryMed` objects) from CORBA services and can set up multiple threads to support multiple requests at the same time.

Both the database and mediator need to use a configuration file to configure the objects before start up. The configuration file would contain the following attributes: CORBA name server location, CORBA name server port, Object name used for registering in CORBA name server, log file name, and for `QueryMed` object, it also needs the list of `QueryMed` and `QueryDB` objects for distributing the queries.

With `SysPara` object as the parameter of `query()`, we can detect infinite loops and avoid long waiting. The `qid` in `SysPara` is a unique number to identify a query. This number consists of the system time when the user generates the query, the IP address of the user's machine, plus a four-byte random number. As described before, when a mediator needs to call other mediators or database engines, it has to pass this parameter to them by using the newly modified `query()` method interface. The mediator itself will keep track of all IDs of those queries that are already submitted but no replies yet. In case there is an upcoming query with the same ID as any one entry in its record, we can tell an infinite loop has occurred. When an infinite loop is detected, that query mediator will simply do nothing and return an empty string to the caller.

`maxlayer` states the maximum layer that the query can travel onwards. When that value is passed from one mediator to another mediator or database engine, the value will decrease by one. The query will stop being forwarded when the `maxlayer` value becomes zero. `timeout` states the maximum time in milliseconds that a mediator or database engine can wait. When that value is passed from one mediator to another mediator or database engine, the value will be decreased by the estimated processing time of that mediator itself. The estimated time is calculated by the statistic of previous connections and queries. The query will stop being forwarded when that value becomes zero.

6.3 Helping the Mediator System to Expand Across the Firewalls

We use CORBA to implement our mediator query system. Though CORBA is a very good architecture for distributed systems, we still meet some difficulties in achieving a real scalable query system, because the common use of firewalls will block CORBA IIOP communication. Here, we apply our mechanism with using XML and Java Servlets to expand our system across firewalls.

6.3.1 Implementation

We now have two mediator query systems as above, and there is a firewall separating them. To enable their communication, the `QueryMed` object must be able to be called by an object (say, another mediator object) from another enclave outside the firewall.

In our implementation, the `QueryMed` object that would be called by outside is associated with a Servlet component. The Servlet component forwards the requests from outside to the `QueryMed` object immediately, thus the `QueryMed` object can accept HTTP requests from outside. We use TOMCAT Servlet engine [37] in our implementation.

On the client side (caller side), we have created a new class, `HttpGateway`, which is the Shadow Mediator object and is used to connect to the Servlet component of the target mediator. `HttpGateway` class implements the same interface, i.e. `QueryEngine` interface, as the `QueryMed` mediator object does. Besides, `HttpGateway` also implements another interface, `HttpQueryGateway`, for its special need. This interface is shown in Figure 6.5.

```
public interface HttpQueryGateway
{
    public String medURL();
    public void medURL(String U);
}
```

Figure 6.5: `HttpQueryGateway`, another interface that `HttpGateway` Class implemented

The `medURL()` method in the interface is used to specify the URL, or the IP address of the target mediator, which is located in another CORBA enclave. This methods should be invoked by its server only, which contains the `HttpGateway` at the same host.

If a mediator wants to call another mediator located at another CORBA enclave, it only needs to call the corresponding `HttpGateway` object. (Actually, that mediator can treat that `HttpGateway` object as the real target mediator object.) The `HttpGateway` object will convert all the necessary parameters into XML format, and then send the request message to the target mediator by HTTP. The target mediator has a Servlet component and will receive the HTTP calls. It then converts the XML parameters back to their original format.

We can summarize the procedures for communication by referring to the scenario shown in 6.6. The scenario is that Mediator `M1` wants to make a query to Servlet component `SC` of the mediator `M2` in another CORBA enclave. The procedures are:

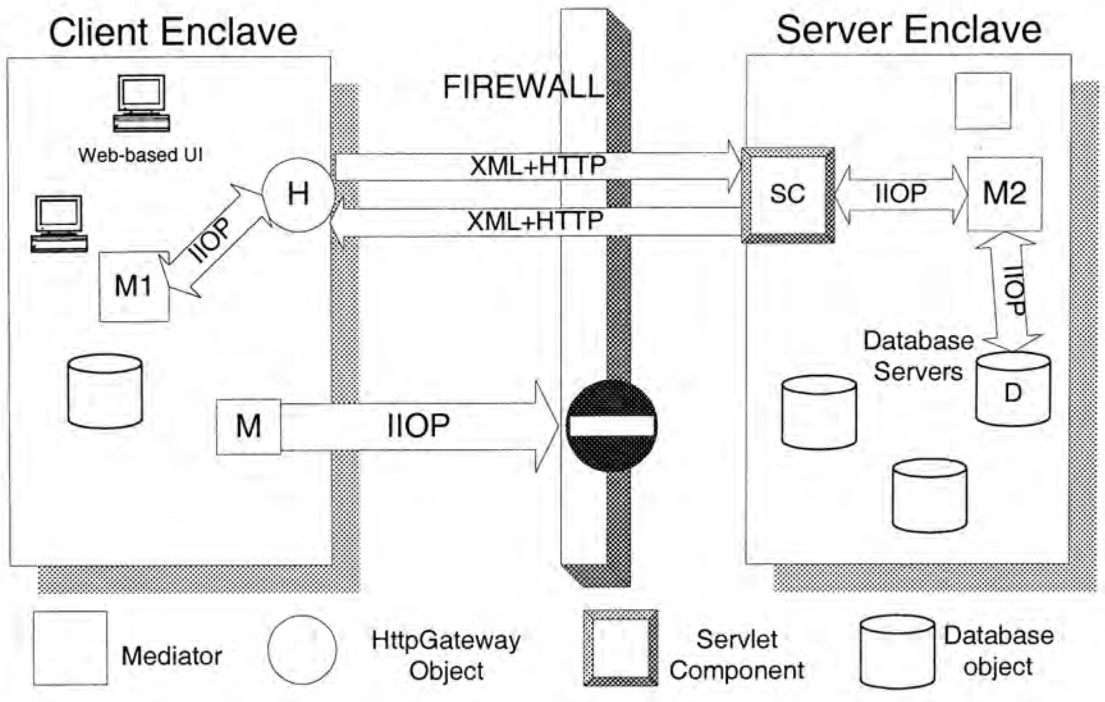

Figure 6.6: The architecture of our query system

1. Mediator `M1` calls `HttpGateway` object `H` with ordinary IIOP connection.

2. `H` converts the IIOP calls to HTTP calls with parameters converted into XML format.

3. The Servlet component, `SC`, of the target mediator gets the HTTP calls from `H` and converts them back to ordinary calling to the target mediator, `M2`.

4. `M2` keeps on calling other Database object `D`, the result is returned to `M2`, and `M2` further returns it to `SC`.

5. `SC` converts the result in XML format, and returns it with `HTTP` calls to `H`.

6. `H` returns result back to Mediator `M1` by using ordinary IIOP return method.

```
<request>
    <QueryEngine type="interface">
        <query type="operation">
            <parameter ref="in" order="1">
                <SysPara>
                    <long name="qid">3984982418240339</long>
                    <long name="timeout">2000</long>
                    <short name="maxlayer">3</short>
                </SysPara>
            </parameter>
            <parameter ref="in" order="2">
                <string name="QueryStatement">
                     where <news>$B</news> in "database.xml"
                     <keyword>satellite</keyword> in $B
                     construct <result> $B </result>
                </string>
            </parameter>
        </query>
    </QueryEngine>
</request>
```

Figure 6.7: An sample request message in XML for calling a mediator object

We have described that parameters are converted to XML format for transmission. Here shown in Figure 6.7 is a sample of such XML request messages with parameters embedded. Figure 6.8 shows a typical response message in XML format. We use tags to state the objects that are being called, the method being invoked, the required parameters and their types, and the values of those parameters.

We can see that both simple data types (like `String` type variable of XML Query Statement) and complicated class objects (like the `SysPara` class of other enhancement parameters) can be well represented by XML. Basically, it is believed to be able to handle all kinds of data structures because of XML's semi-structured nature.

```
<response>
   <QueryEngine type="interface">
      <query type="operation">
         <return>
            <string>
               <news> <source>South China Morning Post
               </source> <date> <day>15</day><month>4
               </month> <year>2000</year> </date> <title>
               Press warning appro priate, says Beijing
               </title> <content>Beijing yesterday defended
               remarks made by senior SAR-based official
               Wang Fengchao that local media should avoid
               reporting separatist views.</content> </news>
            </string>
         </return>
      </query>
   </QueryEngine>
</response>
```

Figure 6.8: An sample response message in XML returns from a mediator object

6.3.2 Across Heterogeneous Systems with DTD

To achieve a scalable system, we need to deal with the heterogeneity of different local systems. We set up some standard formats for different systems to follow in order to communicate with other systems. We need two standards, one is structure of data, and another one is the interface of the system components. If the structures of data cannot be compromised, we will have confusion of communication. If the interfaces cannot be compromised, we even cannot invoke other components of the system. Both important information can be obtained from CORBA IDL files.

To reach a compromise on a standard for data, we use DTD as the grammar book for XML data. This DTD is obtained from the corresponding IDL file by our conversion schema. IDL gives an interface for programmer to develop objects that have the same interface. But IDL itself is not enough, as for parameters passing with using XML and HTTP, we also need to define the parameter format in XML by DTD. The DTD for parameters is shown in Figure 6.9. Hence, different systems can follow the DTD and understand the parameter formats. By following all those mentioned, we can achieve a scalable query without any firewalls or heterogeneous systems problems.

```
<!-- For Request Messages -->
<!DOCTYPE request [
    <!ELEMENT QueryEngine (query)>
        <!ATTLIST QueryEngine type (#CDATA)>
    <!ELEMENT query (parameter*)>
        <!ATTLIST query type (#CDATA)>
    <!ELEMENT parameter (SysPara | string)>
        <!ATTLIST parameter ref (#CDATA)>
        <!ATTLIST parameter order (#CDATA)>
    <!ELEMENT SysPara (long,long,short)>
        <!ATTLIST SysPara name (#CDATA)>
    <!ELEMENT long (#CDATA)>
        <!ATTLIST long name (#CDATA)>
    <!ELEMENT short(#CDATA)>
        <!ATTLIST short name (#CDATA)>
    <!ELEMENT string (#CDATA)>
        <!ATTLIST string name (#CDATA)>
]>


<!-- For Response Messages -->
<!DOCTYPE response [
    <!ELEMENT QueryEngine (query)>
        <!ATTLIST QueryEngine type (#CDATA)>
    <!ELEMENT query (return)>
        <!ATTLIST query type (#CDATA)>
    <!ELEMENT return (string)>
    <!ELEMENT string (#CDATA)>
]>
```

Figure 6.9: The DTD for the parameter passing of simulated calls

```
module QueryEngineApp
{
   struct SysPara
   {
      long qid;
      long timeout;
      short maxlayer;
   };

   interface QueryEngine
   {
      string query(in SysPara para, in string QueryStatement);
      void subscribe(in QueryEngine qe, in string topic);
      void notify(in string newContent);
   };
};
```

Figure 6.10: The IDL design of our system

6.4 Adding the Callback Feature to the Mediator System

To better help the users in obtaining the information they need, one important feature of modern information systems is allowing users to specify some topics of information they want to subscribe. Whenever there is an update of the specified information, the digital library can inform the subscribed users immediately. This feature requires callbacks.

To allow callbacks, we add two methods to the `QueryEngine` interface. One is `subscribe()`, which takes a `string` as parameter to specify the topic of information that the caller wants to subscribe; and an object with `QueryEngine` interface as another parameter to specify the object requests for callback. To be generic, all user interface objects, mediator objects, shadow objects, and database objects would implement this interface. Figure 6.10 shows the new IDL file.

A conceptual diagram of our system mechanism for callbacks is shown in Figure 6.11. And below is the step-by-step desciption of the procedures:

1. Mediator `M1` calls `HttpGateway` object `H1` with ordinary IIOP connection. `M1` also puts itself as one of the parameter in `subscribe()` method. (Same invocation method as calling the target mediator for normal callback)
2. When `H1` observes that it is a callback invocation, it generates a Servlet component (`SC1`), which is assoicated with `M1`, immediately.
3. `H1` sends the IIOP calls to HTTP calls with parameters converted into XML format. The information of `SC1` will also be sent to the server side. These information are embedded into the parameter tag as attributes.
4. When `SC2` observes that it is a callback invocation, it generates a shadow client object, `H2` (shadow of `M1`), immediately. `H2` is initialized by the information of `SC1` (such as IP address, port number).
5. `SC2` will invoke `M2`'s `subscribe()` method substituting `M1` by `H2` in the parameter position, such that `M2` will invoke `H2` when callback is needed.
6. Whenever there is a callback, `M2` calls `H2` `notify()` and H2 will send the request to `SC1`. Finally, `M1` `notify()` method will be invoked by `SC1`.

6.5 Connecting our CORBA System with Other Environments

Merging only CORBA systems would be a great limitation for system integration. Here, we demostrate how we apply our mechanism to allow CORBA objects, DCOM objects and Java RMI objects to be able to call each other.

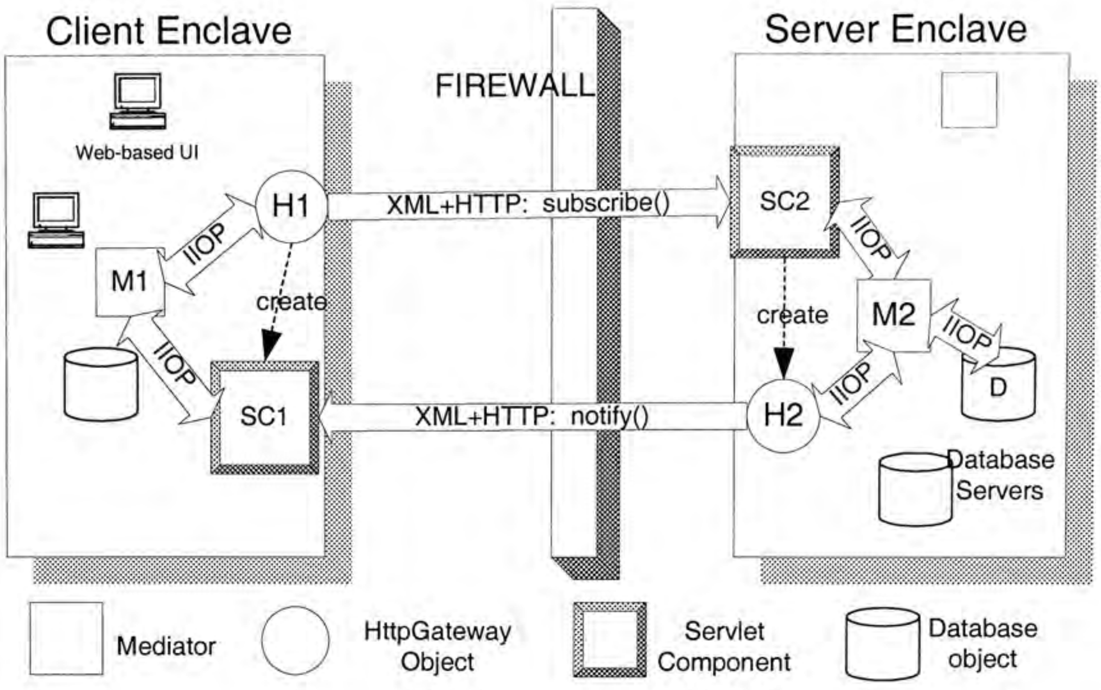

Figure 6.11: Mechanism for supporting callbacks in our query system

Our target is to expand our system across heterogeneous distributed environments. To make the whole system to be more generic, we carefully design the MIDL of the DCOM system and interfaces of RMI components to be very similar to our existing CORBA system, such that calling the DCOM mediators or Java RMI mediators would have no difference as calling the CORBA mediator objects. For simplicity of the example, we use the CORBA IDL in Figure 6.3 to develop our DCOM system and Java RMI system.

6.5.1 Our Query System in DCOM

Our DCOM system is developed on Windows 2000 operating systems, with using Microsoft Visual J++ for implementation. Our implementation is based on the MIDL file shown in Figure 6.12. From the MIDL, we can find out that it is basically the same as the IDL of CORBA. One thing worth to point out is `query()`, the return value is specified in the parenthesis with marking as `retval`. It is because the default return type in DCOM object is `HRESULT`, hence the real return value is defined inside the parenthesis.

```
import "oaidl.idl";
import "ocidl.idl";

typedef struct SysPara
{
    long qid;
    long timeout;
    short maxlayer;
}SysPara;

[ uuid(AC6EDE04-ADF2-4324-BB8C-B350295BFD5E) ]
interface ICOMQueryEngine : IDispatch
{
    HRESULT query([in] SysPara para,
                  [in] char * queryStmt
                  [out, retval] char ** rtn);
};

[ uuid(AC6EDE03-ADF2-4324-BB8C-B350295BFD5E), version(1.0) ]
library QuerySystemLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
        uuid(AC6EDE02-ADF2-4324-BB8C-B350295BFD5E),
    ]
    coclass QueryEngine
    {
        [default] interface ICOMQueryEngine;
    };
};
```

Figure 6.12: The MIDL file for the query system in DCOM enclave

6.5.2 Our Query System in Java RMI

Our Java RMI system is developed in the Unix environment, but it can be run in any operating systems. Our implementation is based on the Java interface definition files shown in Figure 6.13. They are basically the same as the IDL of CORBA system.

One special thing to point out is the struct type of `SysPara` in IDL. As Java interface definition does not support struct type, a new class of `SysPara` is defined instead. But it is mapping to the same XML schema as struct type in XML.

```
/* SysPara.java */
public class SysPara implements java.io.Serializable{
    public long qid;
    public long timeout;
    public short maxlayer;

    public SysPara() {
        qid=-1;
        timeout=-1;
        maxlayer=-1;
    }
}
```

```
/* QueryEngine.java */
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface QueryEngine extends Remote
{
    String query(SysPara para, String queryStmt)
            throws RemoteException;
}
```

Figure 6.13: The DTD for the parameter passing of simulated calls

6.5.3 Binding Heterogeneous Systems

With the interface definition files of DCOM system and Java RMI, the same XML schema can be mapped from those interfaces. Hence, the mediator objects of all systems would have the same interface for calling, hence the scalability of the binded system is greatly increased. Figure 6.14 shows how the mediator objects in heterogenous distributed environments communicate with objects in other enclaves. A common XML schema is the key part to achieve this communication.

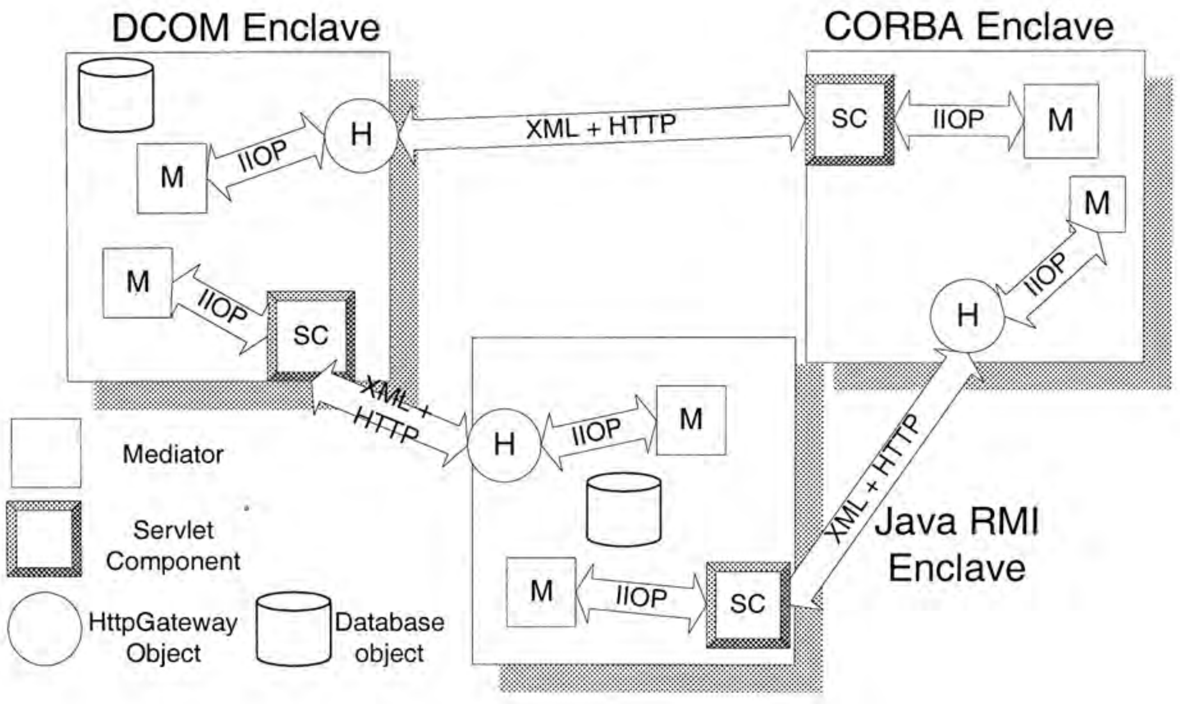

Figure 6.14: Query system in heterogeneous environments with our mechanism

In fact, this is for the demostration of a generic query system across heterogeneous distributed systems with applying our mechanism in it. By matching the newly designed interface definition with the existing XML schema, a highly generic and scalable mediator-based query system is achieved. In normal way of system integration, we use the interface definition files to generate the XML schema for data transmission, but not using the XML schema to design the interface definition.