Comparing CORBA and Web-Services in view of a Service Oriented Architecture

Hayyan R. Sheikh
Regis University,
Graduate School of Computer
and Information Sciences.

ABSTRACT

The concept of Service Oriented Architecture revolves around registering services as tasks. These tasks are accomplished collectively by various disparate components seamlessly connected to one another. The task of interlinking these components may be considered amongst the most convoluted and difficult tasks currently faced by software practitioners. This paper attempts to show that although middleware technologies can be solely utilized to develop service oriented architecture, however such architecture would severely lack quality, interoperability and ease of implementation. In order to resolve these complexities and complications this paper proposes Web Services as an alternative to Middleware, for the realization of a fully functional interoperable and an automated SOA which conforms to the characteristics of a SOA. This paper provides an abstract implementation model of a SOA using both middleware and web services. It then attempts to point out the implementation and accepted benefits of the latter, especially when legacy applications are involved. Emphasize is laid out on the significance of interoperability since it assists in mobility and other corporate benefits. The paper concludes that when interoperability along with its benefits of mobility, expansion, costs, simplicity and enterprise integration are required in the construction of a SOA then web services should be the definite integration choice. The paper also highlights the importance of object oriented middleware, along with situations in which it might be preferred over web services.

Keywords

Middleware, CORBA, Web Services, SOA, Interoperability

1. INTRODUCTION

SOA is a popular methodology for seamlessly integrating heterogeneous components with each other, leading to the concept of service consumers and service providers. According to this methodology a service can individually or with the help of other services deliver and accomplish various business functionalities and objectives.

This service may itself be a functionality of a centralized application, or it may be constituted from the functionalities of various disparate distributed applications that may be spread over multiple heterogeneous enterprises.

Once established these services are then exposed for utilization to consumers, which themselves may be other services. Thus the application of services for shaping a SOA involves two steps, the first involves the interlinking of disparate applications with one another and the second step consists of exposing these services for usability purposes.

Currently majority of software practitioners either employ object oriented middleware components (COM, EJB and CORBA) or web services for interlinking disparate applications and then exposing them as services, thus reusing existing applications saving both time and effort. Although both of these techniques are feasible for establishing a SOA, however a service architecture developed exclusively by middle ware would not only severely lack critical SOA qualities such as interoperability, ease of implementation, configuration and connectivity but it would also strictly limit the potential growth of a business enterprise in a number of ways including the inability of an organization to freely integrate with other organizations and potential business entities functioning on different technology models. Conversely an architecture model constructed from web services is interoperable, business reliant and un-perplexed. It offers ease of connectivity and is far more business friendly.

This paper is divided into three sections the second section explains a service oriented architecture, its importance and characteristics. The third section highlights the middleware (EAI) approach to SOA. In view of the fact that the fundamental concept behind any component interlinking technology is to allow component reuse, this section provides a functional sample of the mechanisms and hurdles involved in establishing services using CORBA as a middleware between legacy and modern applications. The fourth section highlights how those drawbacks could be removed by substituting middleware with Web Services.

2. SOA

Service Oriented Architecture assists in the development of reusable components which are based on an existing framework. This methodology basically compromises of an IT infrastructure in which each loosely coupled heterogeneous component supports interoperability and is independent. These components have the ability to interact with one another despite being developed in inconsistent platforms and programming languages [1]. SOA facilitates and encourages software practitioners into developing systems for users, who would be shielded from the underlying software complexities and still be able to take full advantage of the system [2]. By employing this architecture, previous complex, manual and tedious data processing tasks could be automated

2.1 Need for SOA

SOA caters to the demands of lighter, faster and more reliable interoperability mechanisms along with B2B inter-communication, assisting both the developers and the business enterprises. A software developer can benefit from this methodology by integrating and reusing existing heterogeneous components with other components thus saving time and effort. Moreover the developer is also shielded from the underlying complexity and is capable of successfully integrating complex components with one another despite being oblivious to their internal working and structure. An organization benefits by lower costs in IT infrastructure since enterprise applications operating in SOA no longer need large scale expensive mainframes for functioning, as the load can be divided equally onto lighter machines.

This architecture extends the business functionality of an enterprise by allowing B2B and B2C connectivity thus opening new business prospects and diversifying the potential customer base. Managers can benefit from SOA by securing a greater customer base and easily integrating with other enterprises.SOA can also prove to be beneficial to customers as it could facilitate them with a list of available services that could be graded on factors such as quality and cost. Since SOA supports interoperability, customers can access these services from modern light-weight devices.

2.2 Characteristics of a SOA

Abuosba and El-Sheikh state that “researchers have provisioned various SOA implementations but a general formal standardized definition of this methodology remains absent”. SOA is not a specification but it is an implementation methodology. There is no formal definition of SOA and to date no formal specifications of its functions and characteristics exist. The primary objective of SOA is to address the problem of integrating heterogeneous components to seamlessly integrate with each other, since this arduous task is rarely easy and very challenging. Following is a list of characteristics [4] that should be exhibited by services in a SOA.

1-All the services in a particular system are autonomous and self sufficient. Services can be dynamically located and invoked during runtime.

2-Services support all modes of operation and are interoperable. Services are distributed and can be accessed via network.

3-Services should possess the tendency to expand and be automated..

3. EAI APPROACH TO SOA

One of the fundamental motivations behind object oriented EAI (Enterprise Application Integrators) was to boost distributed computing by providing a mechanism of communication between two heterogeneous, autonomous and loosely coupled applications.

Geihs defines middleware as the “software layer between the operating system including the basic communication protocols and the distributed applications that interact via the network”. This software infrastructure facilitates the interaction among distributed software modules, in other words middleware component facilitate the operating system with the ability to communicate over the network [5][6]. None of these proprietary integrators have ever been truly successful since they provide a solution using a set of branded technologies thus severely limiting its usability and compatibility. By using proprietary technologies for integration purposes, developers have to eventually confront the convoluted and the arduous challenge of re-integrating an integrated solution thus reducing an applications scalability and further intensifying its perplexity [7].

In order to understand the limits and shortcomings of a SOA built exclusively using middleware, it is necessary to understand how middleware manages the flow of information from one endpoint to another. Patterns such as the Proxy Design Pattern and the Broker Architecture Pattern are considerably similar to those employed by the proprietary integrators. These patterns involve dividing an application into various components. Armstrong et al. defines component architecture as “a specification of a set of interfaces and rules of interaction that govern the communication among components and other necessary tools, such as repositories and composition tools"[8].

3.1 Broker Architecture Pattern

The Broker architecture pattern depicted in Figure-1 is basically an extension of the proxy design pattern and addresses its shortcomings. The Broker architecture pattern requires the presence of a broker which acts an intermediary between the client and the server proxies. It should be noted here that the client and server do not directly communicate with the broker but instead use the client and the server proxies as intercessors. Upon startup the broker registers the server and assigns it a communication port. When the client needs to communicate with the server, it passes the request to the client proxy. The client proxy acquires the details of the server from the broker, packages it with the request and forwards it to the broker. The broker in return passes the request to the Server Proxy which ultimately sends the request to the Server. This cycle is followed in reverse for sending the response to the client from the server. Once the broker receives a response from the Server proxy, the broker passes the response to the client proxy. The client proxy in turn copies the result on to the clients memory for accessibility. Most of the EAI integrators such as Microsoft COM+, OMG, CORBA and Java RMI utilize patterns resembling the Broker pattern for distributed computing [7].

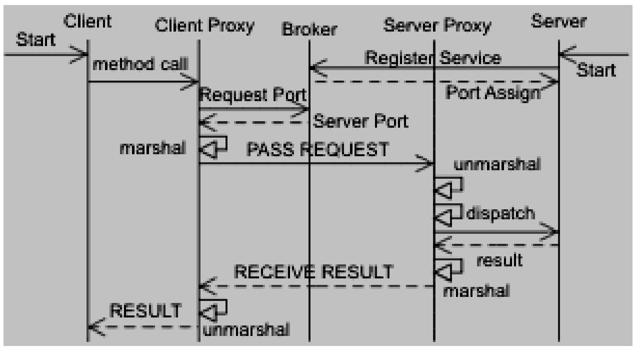


Figure 1: Broker Architecture Pattern

3.2 Distributed computing using Broker Architecture Pattern

Consider the example of a highly automated hotel resort with various branches in a particular city. Assume that the enterprise application employed commonly by each branch consists of separate application components for booking, billing and scheduling each of which has been developed in different environments. These applications have all been integrated using object oriented middleware Figure-2. These branches are interconnected to each other using proprietary middleware technologies; in this case CORBA is employed. An overview of their interconnections, which gradually grew with the passage of time using middleware technology based on the broker architecture pattern, is depicted in Figure-3.

.

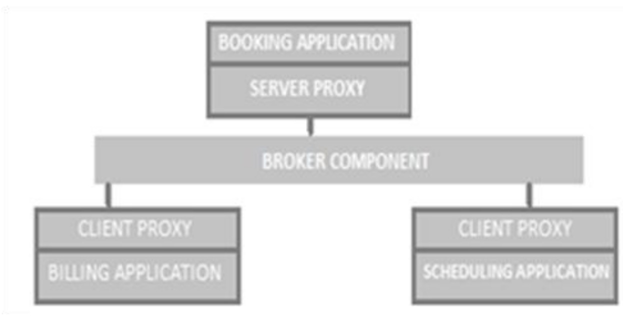


Figure 2: Three components integrated using middleware used by a single branch

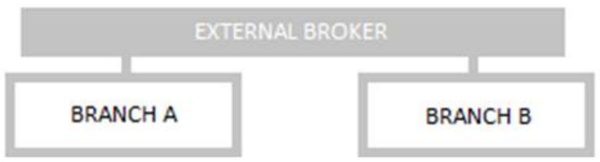


Figure 3: Interconnecting branches using middleware

From the figure it is practically easy to point out the drawbacks of implementing a wide scale distributed enterprise relying purely on an object oriented proprietary middleware technology. A total of 3 broker components will be involved when interconnecting two branches, one broker in each branch and an external broker. This shows that as the application grows, the quantity of its connectors also increase, in this case the proxy components and the broker components. CORBA is considered to be the most interoperable middleware currently available in the market [9]. The abstract implementation depicted in Figure 4 , without dwelling into the details of memory management and exception handling demonstrates an oversimplified version of how two disparate application (one being legacy) could be interconnected with each other using CORBA as a middleware. The purpose of this model other than giving a simplified overview of the CORBA architecture and its implementation is to demonstrate the complexities in type mapping along with location and configuration dependencies. In this example, the functionality of the Booking Application which provides details of rooms is exposed as a service. This service is then accessed and consumed by the Billing Application. The first stage involves the generation of stubs and skeletons from an IDL file. These stubs and skeletons are then added to the client and server applications and act as intercessor proxies.

These proxies connect to the ORB naming service. The naming service allows objects to be registered using friendly names thus simplifying accessibility. However the naming service needs to be configured properly before it can be utilized. The Booking application is developed in C++ which would demonstrate CORBA capability to integrate with native code and the Billing Application in JAVA. The Booking Application registers its service using a friendly name on the Broker employing the naming service .The billing application is provided with the address of the naming service of the Broker Application. The comments in the code would assist in understanding its functionality.

3.3 Drawbacks of wide scale integration using middleware EAI

Some of the drawbacks of wide scale integration using middleware are as follows

3.3.1 Lack of Distributed Transparency

The objective of middleware is to hide implementation details and complexities from the applications employing it for the purpose of distributed communication. However this objective is not accomplished in the current scenario because every time a new component is added and the overall software architecture is extended various previous components will have to be reconfigured to accommodate the changes thus restricting dynamic connectivity along with an increase in overall configuration perplexity. This is apparent from the above CORBA example as the Billing application requires the address or the IOR (Interoperable object Reference) of the Booking application .It also needs to be updated every time the Booking Application is migrated.

3.3.2 Implementation Complexity and a Maintenance Nightmare

The primary benefit of a SOA is to reuse the functionalities of existing legacy applications. CORBA no doubt achieves this benefit but with the huge added cost of complexity. CORBA to C++ mapping is extremely complex and requires a considerable understanding and experience. CORBA based applications can also prove to be a maintenance nightmare. It can be seen from the above implementation example that the proxy components of the applications are generated using the CORBA IDL .If any object is later extended or its properties changed the IDL file will need to be modified as a result all the proxy components will need to replaced by the newer ones..

3.3.3 Limited Connectivity

The above system only supports a single communication (TCP/IP) protocol and does not support the concept of generic connectivity. The term generic connectivity refers to connectivity provided by modern portable devices such as cell phones. If the concept of generic connectivity was supported by this system, a user could simply run queries against the system using a cell phone. The middleware currently available does not support modern computing devices since they are too large and resource costly to run in mobile devices with limited resources [10].

3.3.4 Increase in dependency

From Figure-3 it can be seen that an additional broker component is required to integrate two branches with each other. Thus using object oriented propriety middle ware increases dependency on additional software components and to some extent additional infrastructure is also required. Although two distributed CORBA based applications can interact with one another using IIOP, however these applications would not be able to utilize extensive features facilitated by CORBA such as security and transactions [10]. Furthermore such applications require extensive configuration and are highly dependent on other components thus significantly reducing the chances of encountering CORBA based calls from everyday conventional systems. It can also be seen from the above example that the applications are completely dependent on the CORBA generated proxy components.

3.3.5 High Runtime Environments and Configuration

ORPC (Object Remote Procedures Call) protocols such as DCOM and CORBA require highly efficient and a specific runtime environment [11]. In the implementation example the naming service needs to be explicitly configured for proper functionality. In some scenarios CORBA middleware would be more resource costly than the actual application itself.

3.3.6 Limited Interoperability

Popular available middleware’s such as COM (Component Object Model), DCOM (Distributed COM), and COM+ middleware technologies are Microsoft specific and EJB (Enterprise Java Beans) is Java specific and CORBA (Common Object Request Broker Architecture) though is language and platform independent however as mentioned above the interoperability level between different CORBA products are limited [9]. Hence it can be safely assumed that the integration mechanism in Figure-4 between different branches using middleware would be highly technology and platform dependant

4. WEB SERVICES

Web services are an emerging standard focused on providing disparate application components the ability to integrate with each other using open XML standards for communication [11]. Web Services could be considered as an alternative to CORBA since it provides a mechanism for disparate components to communicate with each other. The following section discusses the web services framework along with its working and potential benefits over object oriented proprietary middleware for developing enterprise friendly service oriented architecture.

4.1 Need for Web Services

The initial working of middle ware components consisted of developing "wrappers" around objects to expose visibility and then using application "connectors" to connect and integrate functionality of these components with other application components. However as enterprise applications and functionality requirements grew larger the number of these connectors also grew exponentially, as shown in the hotel resort example above. The exponential rise in these "component connectors" not only increased application requirement and dependencies but it also complicated application architecture. In order to address these complications the SOA methodology primarily using web services is adopted. This methodology involves exposing the generic functionality of an autonomous heterogeneous application so that it may be used by other applications as a loosely coupled component, thus reducing the number of inter-application connectors from n(n-1) to n connectors since the number of connectors tend to grow exponentially during expansion [3] .

4.2 Web Services Framework

The Web Services Framework encapsulates three basic categories - communication protocols, service descriptions, and service discovery. The communication protocols determine how data should be exchanged and transported among different components in a SOA, service descriptions provide details of a particular service and the service directory provides a collective list of available services. An important standard from each category is discussed below.

4.2.1 Communication Protocol – SOAP

Businesses now days interact with each other and its customers mostly using the internet, hence it is necessary for the communication mechanism to be platform independent, international, secure and lightweight for speed and flexibility thus setting a standard. Currently one of the most popular protocols which conform to the above mentioned qualities is the Simple Object Access Protocol also known as SOAP.

SOAP facilitates interoperability among a wide range of programs and platforms’, making existing applications and components accessible to a broader range of users. SOAP is built on existing open transport protocols like HTTP, SMTP and MQSeries. However web services mostly employ the Http communication protocol thus the name "web services". SOAP is basically an XML based message which is transported over a communication protocol having the primary advantage of http interoperability when it comes to component-component interaction.

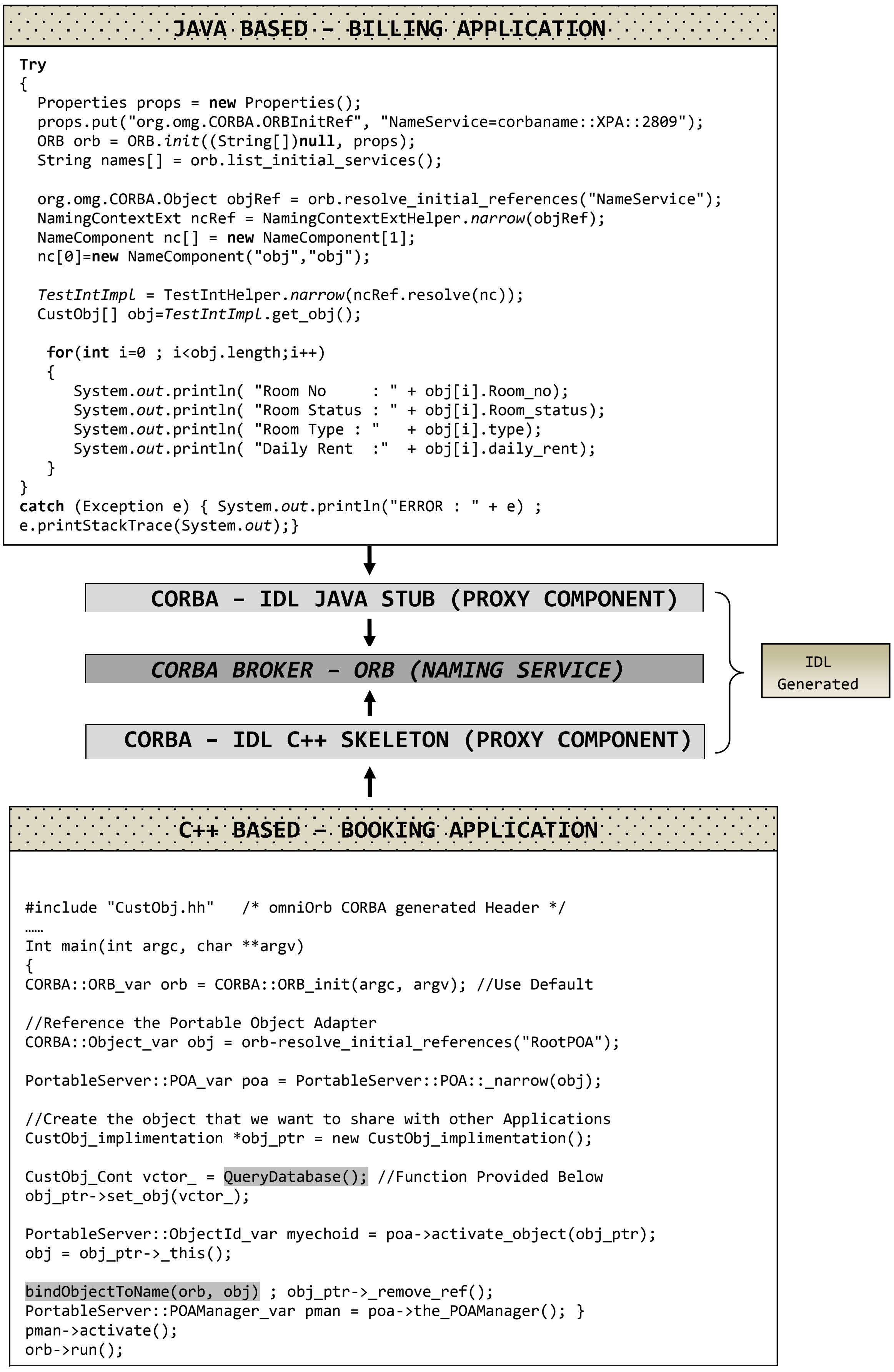
4.2.2 Service Descriptions – WSDL

No doubt communication protocols in a Web Services framework plays an integral role in transporting a message across a distributed environment however this communication protocol would be of no practical value if there was no way to interact with a web service. Fortunately the method of interaction with a web service is defined in the WSDL.WSDL is an XML based specification for the Service Description category of the Web Services Framework and is an acronym for Windows Services Descriptive Language. The WSDL provides a description of the service, this description includes the name of the function to call along with the parameters to pass and expect from a function. The WSDL highlights the type of operation whether it’s a one way operation, request-response, solicit-response or simply a notification operation. The parameters types required by WSDL are also specified in the WSDL file. This in turn shows that Web Services are strongly typed. By utilizing the WSDL a consumer can establish meaningful communication with a service provider.

4.2.3 Service Discovery – UDDI

The UDDI is similar to a phone directory of web services and divides the information about web services into 3 categories known as - white, yellow and green pages. The white pages includes the contact details, the yellow pages provides the names of services categorized by business and services types and green pages provide technical details of services.

The UDDI registry basically consists of four data types that include Business Entity, Business Service, Binding Template, and the TModel. The information and details of a business including the type of services provided is included in the Business Entity. The business is the actual service provider. The technical details for a web service are defined in the Business Service and its binding Templates. Each Binding Template contains a reference to one or more TModels. TModels are a vital constituent of UDDI and serve as a conformance aid to a particular category, specification or an identifier system. .


JAVA BASED – BILLING APPLICATION
CORBA – IDL JAVA STUB (PROXY COMPONENT)
CORBA BROKER – ORB (NAMING SERVICE)
CORBA – IDL C++ SKELETON (PROXY COMPONENT)
IDL Generated
C++ BASED – BOOKING APPLICATION

```
Try
{
  Properties props = new Properties();
  props.put("org.omg.CORBA.ORBInitRef", "NameService=corbaname::XPA::2809");
  ORB orb = ORB.init((String[])null, props);
  String names[] = orb.list_initial_services();

  org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
  NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
  NameComponent nc[] = new NameComponent[1];
  nc[0]=new NameComponent("obj","obj");

  TestIntImpl = TestIntHelper.narrow(ncRef.resolve(nc));
  CustObj[] obj=TestIntImpl.get_obj();

   for(int i=0 ; i<obj.length;i++)
   {
      System.out.println( "Room No     : " + obj[i].Room_no);
      System.out.println( "Room Status : " + obj[i].Room_status);
      System.out.println( "Room Type : "   + obj[i].type);
      System.out.println( "Daily Rent  :"  + obj[i].daily_rent);
   }
}
catch (Exception e) { System.out.println("ERROR : " + e) ;
e.printStackTrace(System.out);}
```

```
#include "CustObj.hh"   /* omniOrb CORBA generated Header */
……
Int main(int argc, char **argv)
{
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv); //Use Default

//Reference the Portable Object Adapter
CORBA::Object_var obj = orb-resolve_initial_references("RootPOA");

PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

//Create the object that we want to share with other Applications
CustObj_implimentation *obj_ptr = new CustObj_implimentation();

CustObj_Cont vctor_ = QueryDatabase(); //Function Provided Below
obj_ptr->set_obj(vctor_);

PortableServer::ObjectId_var myechoid = poa->activate_object(obj_ptr);
obj = obj_ptr->_this();

bindObjectToName(orb, obj) ; obj_ptr->_remove_ref();
PortableServer::POAManager_var pman = poa->the_POAManager(); }
pman->activate();
orb->run();
```

Figure 4: Abstract implementation of interlinking two different components using CORBA

C++ bindObjectToName() Function

```
/* This Function Binds a Friendly Name to the
Registered object in this case the naming service so
that it could be accessed easily by other Applications
Each object will have an id and a name*/

static CORBA::Boolean bindObjectToName
(CORBA::ORB_ptr orb, CORBA::Object_ptr objref)
{
 //Since We are using CORBA Naming Service
 CosNaming::NamingContextExt_var NContext;

 // Obtain a reference to the Name service:
 CORBA::Object_var obj;
 obj = orb->resolve_initial_references("NameService");

/*Narrow the reference returned.The object is still
not a naming context until itis narrowed – narrowing
can be considered as a typecast*/

 NContext = CosNaming::NamingContextExt::_narrow(obj);

 CosNaming::Name contextName;
 contextName.length(1);
 contextName[0].id   = (const char*) "obj";
 contextName[0].kind = (const char*) "obj";

 // Bind the context to root.
 NContext ->bind(contextName,objref);
 return 1;
}
```

C++ QueryDatabase() Function

```
/*This Function Queries the Database and return the object
The following example employs SOCI an open source library
to connect and query the database. The object types have
been generated by the IDL , furthermore CORBA has its own
object types such as sequences a substitute of vectors */

CustObj_Cont QueryDatabase()
{
 int count; CustObj_Cont vctor_ ;
 session sql(oracle, "service=orcl user=hayyan
 password=admin");
 sql << "select count(*) from booking", into(count);
 std::vector<int> Room_No(count);
 std::vector<int> Daily_Rent(count);
 std::vector<std::string> Room_Availability(count);
 std::vector<std::string> Room_Category(count);
         //Run a SQL Statement
 sql << "select * from booking",
 into(Daily_Rent), into(Room_Availability),
 into(Room_Category), into (Room_No);
 vctor_.length(count);
 for(int i=0 ; i<count;i++) {
   CustObj obja;
   obja.daily_rent =  Daily_Rent.at(i);
   obja.Room_status= (Room_Availability.at(i)).c_str();
   obja.type= (Room_Category.at(i)).c_str();
   obja.Room_no = Room_No.at(i);
   vctor_[i]=obja; }
 return vctor_;
 }
```

C++ BASED – OBJECT IMPLIMENTATION

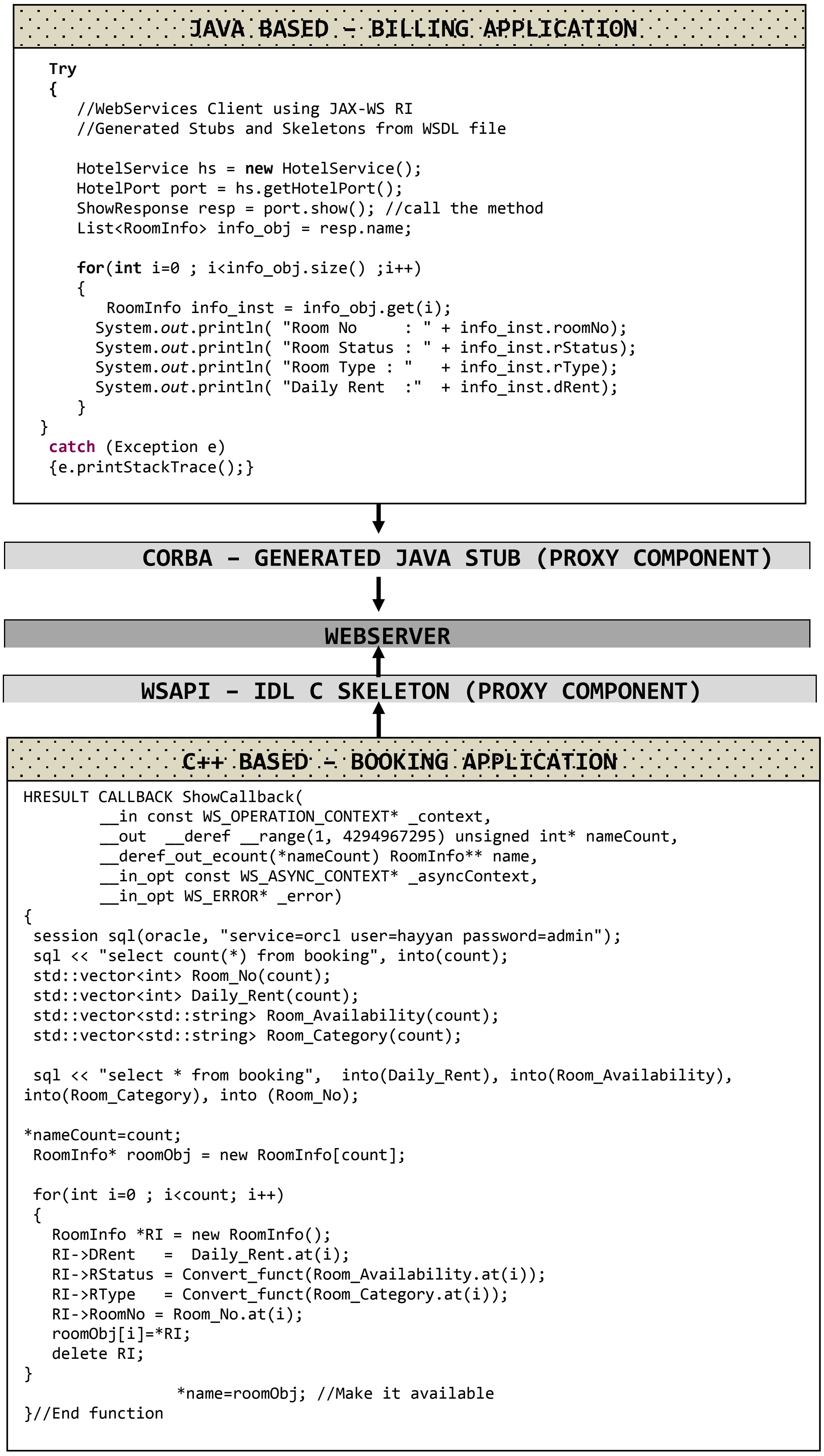
```
/*This code impliments the object and is based on the IDL File used for generating
  The stubs and skeletons by CORBA IDL */

class CustObj_implimentation : public POA_TestInt
{
private:
        CustObj_Cont_var VctorHolderVar; //Notice the object contains member variables. The properties of these
objects
        CustObj_Cont VctorHolder;      //had been defined in the IDL , in case the properties change the IDL file will
                                       //need to be rebuilt and in source segmented the _var is a CORBA type and
                                      //resembles safe pointers. This allows CORBA to manage memory.
public:
        virtual CustObj_Cont* get_obj();
        virtual void set_obj(const ::CustObj_Cont& rx);
};

void CustObj_implimentation::set_obj(const ::CustObj_Cont& rx)
{VctorHolder = rx ; //make a copy}

CustObj_Cont* CustObj_implimentation::get_obj() throw (CORBA::SystemException)
{
  VctorHolderVar = new CustObj_Cont(VctorHolder);
  return VctorHolderVar._retn(); //Let CORBA  manage Memory
};
```

Figure 4: Abstract implementation of interlinking two different components using CORBA


JAVA BASED - BILLING APPLICATION
CORBA - GENERATED JAVA STUB (PROXY COMPONENT)
WEBSERVER
WSAPI - IDL C SKELETON (PROXY COMPONENT)
C++ BASED - BOOKING APPLICATION

```
  Try
  {
     //WebServices Client using JAX-WS RI
     //Generated Stubs and Skeletons from WSDL file

     HotelService hs = new HotelService();
     HotelPort port = hs.getHotelPort();
     ShowResponse resp = port.show(); //call the method
     List<RoomInfo> info_obj = resp.name;

     for(int i=0 ; i<info_obj.size() ;i++)
     {
        RoomInfo info_inst = info_obj.get(i);
       System.out.println( "Room No     : " + info_inst.roomNo);
       System.out.println( "Room Status : " + info_inst.rStatus);
       System.out.println( "Room Type : "   + info_inst.rType);
       System.out.println( "Daily Rent  :"  + info_inst.dRent);
     }
 }
  catch (Exception e)
  {e.printStackTrace();}
```

```
HRESULT CALLBACK ShowCallback(
        __in const WS_OPERATION_CONTEXT* _context,
        __out  __deref __range(1, 4294967295) unsigned int* nameCount,
        __deref_out_ecount(*nameCount) RoomInfo** name,
        __in_opt const WS_ASYNC_CONTEXT* _asyncContext,
        __in_opt WS_ERROR* _error)
{
 session sql(oracle, "service=orcl user=hayyan password=admin");
 sql << "select count(*) from booking", into(count);
 std::vector<int> Room_No(count);
 std::vector<int> Daily_Rent(count);
 std::vector<std::string> Room_Availability(count);
 std::vector<std::string> Room_Category(count);

 sql << "select * from booking",  into(Daily_Rent), into(Room_Availability),
into(Room_Category), into (Room_No);

*nameCount=count;
 RoomInfo* roomObj = new RoomInfo[count];

 for(int i=0 ; i<count; i++)
 {
   RoomInfo *RI = new RoomInfo();
   RI->DRent   =  Daily_Rent.at(i);
   RI->RStatus = Convert_funct(Room_Availability.at(i));
   RI->RType   = Convert_funct(Room_Category.at(i));
   RI->RoomNo = Room_No.at(i);
   roomObj[i]=*RI;
   delete RI;
}
                *name=roomObj; //Make it available
}//End function
```

Figure 5: Abstract implementation of interlinking two different components using Web services

```
C++ (Booking Application) Boot-Up

int main()
{
HRESULT hr = S_OK;WsError error;
HR(error.Create(0,0)); // property count
hr = Run(L"http://+:81/Hotel",   error);  //Hosting Address
}

/*This function allocates the space necessary to host and expose the function as a Webservice – The
exposed function is attached to the proxy through a callback */

HRESULT Run(__in PCWSTR url,__in_opt WS_ERROR* error)
{
WsHeap heap; HR(heap.Create(1200,        0,0,0,  error));
const WS_STRING address ={static_cast<ULONG>(wcslen(url)),       const_cast<PWSTR>(url)};

//Bind the Generated Stub with Functions to call
HotelBindingFunctionTable functions ={ShowCallback};

WS_SERVICE_ENDPOINT* endpoint = 0;
HR(HotelBinding_CreateServiceEndpoint(0,&address,&functions,0,0,0,heap,&endpoint,error));
const WS_SERVICE_ENDPOINT* endpoints[] = { endpoint };

//Create an endpoint and establish a connection.
WsServiceHost serviceHost;
HR(serviceHost.Create(endpoints,_countof(endpoints),0, 0,error));
HR(serviceHost.Open(0, error));
wprintf(L"Press any key to STOP the server.");
_getch();
HR(serviceHost.Close(0, error));
return S_OK;
}
```

Figure 5: Abstract implementation of interlinking two different components using Web services

4.3 Functioning of Web Services

Figure-6 shows the basic working and flow of a web service. The first step involves discovering the service provider from a directory; in this case UDDI is employed. Once the details of a service provider are acquired and processed by a client, it can then access the services abstract descriptor. Using WSDL the client can analyze the parameter requirements of the service provider (i.e. the parameter that the service requires as an input and the parameters that will be returned).The final step involves the client querying the service itself. The service might return the results to the clients instantly or it might query other services and then return the results.

4.4 Web Services - Distributed Computing

In section 3 a functional service oriented model was developed using CORBA, the same functional model could be developed using Web Services. In the model shown in Figure-5, the native web service has been developed using WSAPI. The skeletons and stubs have been generated by WSUTIL library in native C. This service is interoperable and could cater to the needs of various clients running on distinct platforms. Different open source and mature libraries such as GSOAP are also available that facilitate developing Web services for C++ based applications. The Java based billing application, acts as a client consumer and connects to the service provider using JAX-WS. Objects and types are defined in the WSDL including the mode and address of communication. From the following model it can be seen that consuming and constructing a web service is simpler and far less complicating than connecting to a middleware CORBA naming service. Furthermore no external configuration is required as in the case of a naming service for CORBA based components; also in most scenarios no external libraries are necessary for implementation.

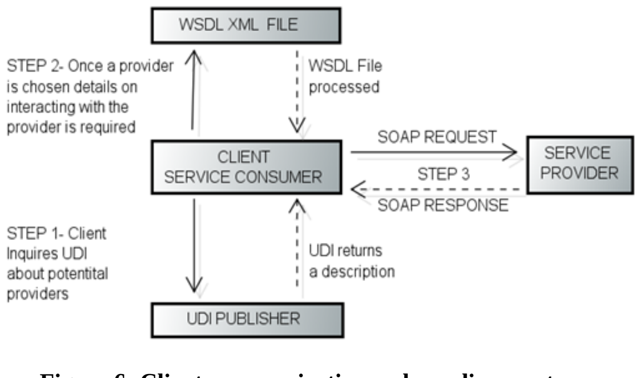

WSDL XML FILE
STEP 2- Once a provider is chosen details on interacting with the provider is required
WSDL File processed
CLIENT SERVICE CONSUMER
SOAP REQUEST
STEP 3
SOAP RESPONSE
SERVICE PROVIDER
STEP 1- Client Inquires UDI about potentital providers
UDI returns a description
UDI PUBLISHER

Figure 6: Client communication order – discover to dispatch

4.5 BENEFITS OF WEB SERVICES

Some of the potential benefits of Web Services are as follows:

1. Web services allow service providers and vendors to sell services over the internet simply by publishing in the UDDI. Notice there is no need of external brokers. Furthermore incase the server (service provider) is migrated only the address in the WSDL needs to be modified. Clients can therefore inquire the WSDL before establishing a connection with the service provider.

2. In web services the interface is separated from the implementation and platform. This allows ease of maintenance, simplicity along with extensibility.

3. Web services as mentioned above employ the SOAP protocol, which is based on XML. The XML standard is generic and interoperable thus facilitating web services with the same qualities of cross-platform and language interoperability. In our distributed hotel resort example all the interlinked applications (Booking, Scheduling and Billing) were based on the same operating system platform. Now if web services were employed this limitation would have been removed.

4. Web services are license free, thus less costly as compared to their propriety object oriented counter parts.

5. CONCLUSION

Implementing and integrating a distributed information system spanning across multiple enterprises comes across heterogeneity and distribution issues. In order to address these issues either middleware services or web services are employed. Both have their benefits and potential drawbacks. Middleware services as the name implies resides in the middle between the components as an intermediary and facilitates the operating system to carry out distributed task with other components that use the network. Some of the necessary qualities expected from a middleware are outlined below [12]

1. **Scalability**: refers to supporting a large number of clients and server which is a very crucial requirement for internet based distributed applications. If all the components of a system are considered scalable then the system is also considered to be scalable. It should manage memory and network resources efficiently. This means that events should only be sent over the network if a subscriber is interested in them.

2. **Interoperability**: The entire concept of middleware revolves around the idea of interoperability, meaning that heterogeneous components built in different languages and running on different platforms are able to communicate with one another instantly and efficiently.

3. **Reliability**: The characteristic of reliability is very important in a middleware. Each component that interacts with the middleware will have specific concerns regarding consistency, reliability and guarantee of message delivery. A middleware can only guarantee delivery of a message if it includes features such as component failure verification, failure bypass and fault tolerance. These features ascertain that a message will be delivered to its destination even if a component failure occurs. Techniques such as persistent events and replication help to make a more durable middleware

4. **Usability:** It is necessary that mechanism for integrating with middleware in any programming language is clear and fully documented. Furthermore the interfacing application should be oblivious and decoupled from the underlying messaging complexities and internal event architecture of the middleware.

5. **Expressiveness**: Applications based on distributive components profit from articulated defined mechanisms for dealing with and subscribing to events.

6. REFERENCES

[1] Newcomer.E , Lomow G., (2005). Understanding SOA with Web Services, Addison Wesley.

[2] Foster, I. (2005, May 6). Service-Oriented Science. Science , 308

[3] Welke, R., Hirschheim R. & Schwarz A.(2011,Feburary). Service Oriented Architecture Maturity. Computer ,IEEE, 44(10), 61-67

[4] Abuosba, K.A. & El-Sheikh .A.(2008,August).Formalizing Service-Oriented Architectures. IT Professional,13(5), 34-38

[5] Geihs, K. (2002, August 7). Middleware challenges ahead . Computer , IEEE , 34(6), 24-31.

[6] Karastoyanova , D., & Buchmann, A. (2003). COMPONENTS, MIDDLEWARE AND WEB SERVICES. Proceedings of IADIS International Conference WWW/Internet 2003,IADIS Press,2.

[7] Stal, M. (2002, October). Web services: beyond component-based computing. Communications of the ACM, 45(10), 71-76.

[8] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., & McInnes, L. (2006, August 3). Toward a Common Component Architecture for High-Performance Scientific Computing. Eighth IEEE International Symposium on High Performance Distributed Computing.

[9] Henning, M. (2006, June). The Rise and Fall of CORBA. Queue - Component Technologies - ACM, 4(5)

[10] Box, D. (2000, March). A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages. MSDN Magazine, (2000)

[11] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhy, N., & Weerawarana, S. (2002, August 7). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI . Internet Computing, IEEE , 6(2), 86-93

[12] Pietzuch, P., & Bacon, J. (2002, July). Hermes: A Distributed Event-Based Middleware Architecture. 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)