Supporting Live Development of SOAP and CORBA Clients

Sajeeva L. Pallemulle
sajeeva@cse.wustl.edu

Vanessa H. Clark
vclark@wustl.edu

Kenneth J. Goldman
kjg@cse.wustl.edu

Department of Computer Science and Engineering
Washington University in St. Louis

Abstract

We present middleware for a Client Development Environment that facilitates live development of client applications for SOAP or CORBA servers. We use JPie, a tightly integrated programming environment for live software construction in Java, as the target platform for our design. JPie provides dynamic classes whose signature and implementation can be modified at run time, with changes taking effect immediately upon existing instances of the class. We extend this model to automate addition, mutation, and deletion of dynamic server methods within dynamic clients. Our implementation simplifies distributed application development by masking technical differences between local and remote method invocations. Moreover, the live development model allows server-side changes to be dynamically integrated into a running client to support simultaneous live development of both the client and server.

1. Introduction

Remote method invocation (RMI) using the client-server paradigm has become a prominent model for developing distributed applications. The Simple Object Access Protocol (SOAP) [1] and the Common Object Request Broker Architecture (CORBA) [2] are two leading technologies that support this model. Although SOAP and CORBA differ significantly in design and usage, the implementation of RMI applications using these technologies follows a similar pattern.

The development of client-server applications using the RMI model requires the creation of separate client and server applications. Therefore, synchronizing the common interface at both endpoints is necessary for simultaneous development. The traditional approach to this problem has been to interleave the editing and testing phases through the deployment of the two applications at various stages of development. However, this approach delays completion and does not fully eliminate the possibility of violating the common interface. Hence, an approach that combines the editing and testing phases into one unified step is particularly attractive in order to streamline application development.

We present a Client Development Environment (CDE) as an extension of JPie, a tightly integrated development environment supporting live construction of Java applications. JPie embodies the notion of a dynamic class whose signature and implementation can be modified at run time, with changes taking effect immediately upon existing instances of the class. [3] We extend JPie to dynamically and automatically add, update, and delete server methods within dynamic client classes in response to server-side changes. Just as methods and their respective calls are developed live within a single application in JPie, we support a live integrated development process in which the client and server applications can be developed simultaneously, with server-side interface changes taking immediate effect on the client program, preserving consistency.

This paper presents the CDE architecture and implementation. In particular, we explain how CDE automatically maps the server interface to the server methods within the client-side dynamic classes. This allows for seamless integration of dynamic changes in the server interface with live instances of the client application, requiring minimal developer involvement. CDE does this while presenting a unified method invocation mechanism for both remote and local calls.

Our architecture supports technologies that use an interface definition language (IDL) to communicate the server interface to the clients. SOAP and CORBA are widely used technologies that satisfy this criteria and the initial implementation of CDE supports both. For SOAP support, we build on the Apache Axis [4] implementation of SOAP. Similarly, we use the OpenORB [5] implementation of CORBA as the basis of our CORBA support. Our design can also be extended to integrate other technologies that use interface definition languages and the remote method invocation model.

The remainder of the paper is organized as follows. In Section 2, we review distributed application development in SOAP and CORBA and present a brief overview of JPie. Section 3 provides an overview of related work. Section 4 focuses on the CDE user interaction mechanism

for creating client applications. In Section 5, we present the CDE architecture and discuss the mechanism used to integrate server interface changes with client-side dynamic classes. Section 6 focuses on the performance and overhead of CDE. We conclude, in Section 7, with a summary and directions for future work.

2. Background

For our initial implementation of CDE, we decided to concentrate on both SOAP and CORBA. We chose two technologies rather than one to help ensure that the design was sufficiently extensible to support other technologies in the future. Both SOAP and CORBA make use of interface definition mechanisms, yet have different overall frameworks. This section presents the necessary background on SOAP and CORBA, as well as on JPie.

2.1. SOAP

Servers that use SOAP are popularly known as Web Services. Web Services use the Extensible Markup Language, (XML) [6] to present the server interface to the clients as well as to communicate with those clients. As shown in Figure 1, when a Web Service is established, it uses the Web Services Definition Language (WSDL) [7] standard to publish a WSDL document that potential client applications can use to gather information they require to invoke methods on the web service.

WSDL is an XML-based schema that contains information such as the location of the web service, the methods that can be remotely invoked on that web service, and how to invoke those methods. The WSDL standard supports direct encoding of a small subset of Java object types and permits the encoding of complex data structures using XML. These complex types enable web services to exchange user defined object or data structures with clients as parameters and or return values.

The client applications use the information published in the WSDL document to form a XML document known as a SOAP Request that encapsulates the remote method call in a standard textual format. The SOAP Request is then sent to the web service.

The web service uses the method and parameter information encoded in the SOAP Request to invoke the method call with the appropriate parameters. Then it constructs an XML document called the SOAP Response that encapsulates the data returned from the method call in a standard XML format. The SOAP Response is then sent back to the client. The client receives the SOAP Response, decodes it, and returns the data to the calling program.

The underlying transport medium that supports this publish-request-response mechanism is provided by the Hyper Text Transport Protocol (HTTP) [8].

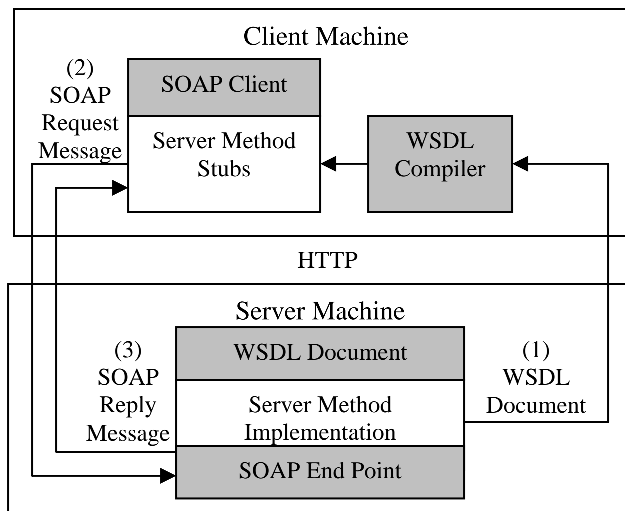

Fig. 1: The client-server interaction using SOAP proceeds in three steps. First, the server interface definition is obtained by the client. Then the client parses this definition and uses the resulting method stubs to make remote method requests using SOAP.

2.2. CORBA-RMI

The Common Object Request Broker Architecture (CORBA) defines a high-level communication model for distributed computing. For the scope of this paper we consider only the RMI aspect of CORBA. The most important notion in the CORBA-RMI specification is an Object Request Broker (ORB) [9]. In a client-server system that uses CORBA-RMI, the Client ORB and the Server ORB form the communication endpoints. They direct invocations and results between remote objects located on client and server sides. ORB implementations use IIOP (Internet Inter-Orb Protocol) [9] to communicate over a network. Unlike HTTP, which only allows text to be transported over it, IIOP supports a wide range of primitives, data structures and object references.

Unlike SOAP, CORBA decouples the interface definition from the location information. CORBA-RMI servers use CORBA Interface Definition Language (CORBA-IDL) [9] to describe object interfaces and an Interoperable Object Reference [9] (IOR) declaration to encode and provide the server URL and port data to the clients. A CORBA-RMI client must attain both a CORBA-IDL document as well as an IOR in order to establish a communication link with a server.

The CORBA-IDL document consists of a standard set of elements. The *module* element is the root element of any CORBA-IDL document. CORBA developers using Java as the host language will notice that each *interface* element, similar to a Java class, encapsulates instance variable declarations and method declarations. The *module* may contain number of uniquely identified *interfaces*.

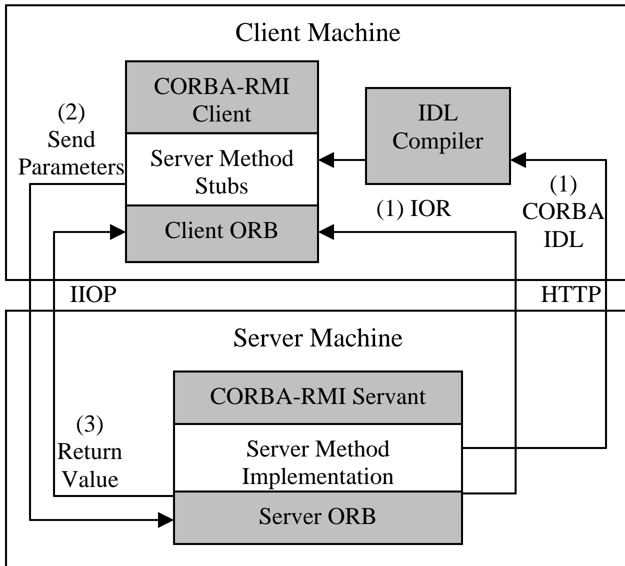

Fig. 2: Initially the CORBA-IDL and IOR definitions are retrieved from the server. Using the IOR the client ORB is initialized. Remote methods defined in the CORBA-IDL are invoked on the client ORB, which contacts the CORBA Servant though the server ORB to obtain the return object.

The CORBA-IDL to Java mapping permits the type of the instance variables, method parameters, and return values to be the Java Strings and primitive types int, double, float, char, and boolean, or any Java type that is declared by an *interface* element within the *module* element of a CORBA-IDL document.

As shown in Figure 2, to establish a communication link to the server, a client uses an IOR to initialize the client ORB. The client ORB then establishes a communication link with the server ORB described by the IOR. After the initialization, the client application invokes the methods defined in the CORBA-IDL document. When such an invocation is made, the call is intercepted by the client ORB and sent to the server ORB over an IIOP connection. The server ORB intercepts the call, finds the object that can handle the request, invokes the corresponding method with the parameters passed in, and returns the results to the client ORB. The client ORB then passes the return object back to the calling program.

2.3. JPie

JPie is a tightly integrated programming environment for live construction of Java applications [10]. JPie treats programming as an application in its own right, providing a visual representation of class definitions and supporting direct manipulation of graphical representations of programming abstractions and constructs. Exploiting Java's reflection mechanism, JPie supports the notion of a dynamic class that can be modified while the program is running. Dynamic classes are built from components such as dynamic methods and dynamic fields, which directly correspond to the respective classes in the Java's reflection mechanism. However, the dynamic versions can be instantiated and mutated. This functionality can be used to, among other things, change method signatures within live object instances. Dynamic classes fully interoperate with compiled classes, including polymorphism, and methods may be overridden on the fly.

Of particular interest is the fact that JPie maintains consistency of declaration and use. For example, if the name or parameter list of a method is changed, JPie automatically updates all calls to that method accordingly. This is different from typical textual programming environments, in which the programmer must update every call whenever a method name is changed or a formal parameter list is reordered. One of the important goals of the present work is to maintain this level of consistency for client development in client-server applications, even in the face of server-side interface changes that must be reflected in the client.

3. Related Work

In spite of the fact that RMI is a natural extension of standard method call semantics, setting up the development tools for technologies such as SOAP and CORBA can be a daunting task. Therefore, client development environments that encapsulate the low-level details of the technology and the execution environment have proven popular among developers. Providing direct access to the server interface in a manner that is familiar to the developers has been a natural next step that some such systems have adopted with varying degrees of success. In this section, we discuss several systems that do not address the issue of live client/server development but help streamline distributed application development using RMI.

Visual Studio.Net [11] builds upon the Microsoft .NET framework [12] to reduce the web services development time. In client-side development, Visual Studio.Net uses Web References [13], which are proxy classes created on the client to connect to the web service. Web References present the client developers with an object interface that contains the server method declarations. Hence, Web References can be used as regular objects within the client application. This implementation is similar to the approach we take in CDE. However, an important difference between the two is that Web References must be refreshed with each change in the server interface. Additionally, the client code may need recompilation to account for conflicts. Therefore, the Visual Studio.Net approach is not particularly suitable for developing client applications against a dynamically evolving server interface.

CapeConnect [14] offers a set of tools for integrating existing middleware components such as CORBA objects and Enterprise Java Beans [15] to a web service front end. CapeConnect is supplemented by two different client development tools. The Web Client Generator can be used to create thin clients implemented with HTML pages and JavaScript [16]. The SOAPDirect [17] Application Programming Interface (API) provides a simplified development model for communicating with CapeConnect using SOAP. SOAPDirect API includes a generic SDRequest object that can be configured to invoke different server methods and receive a reference to the return value. In addition, the API defines an SDDataType object that can be used in creating complex data types. Hence, the SOAPDirect API abstracts away the details of creating SOAP Requests and parsing SOAP Responses. However, client development with SOAPDirect requires the application developer to gain an understanding of the WSDL standard as well as direct knowledge of the available server methods. It also does not address the issue of handing dynamic changes to the server interface.

WebObjects [18] is another set of tools that facilitates simplified development of web services. For client development, WebObjects provides a class named WOWebServiceClient, which can be instantiated with the server URL. Once such an object is created, a call to its invoke method with the service name, operation and an object array containing the arguments will result in a SOAP call to the server. Thus, WebObjects abstracts away the low-level implementation details to some degree. However, to handle server interface changes, the RMI calls within the client code must be manually changed and recompiled. Hence, WebObjects is particularly suited for only client development against a static server interface.

The Apache Axis implementation of SOAP introduces a *Call* object [19], which can be customized to call any server method defined in a single web service. This implementation allows developers to construct RMI calls at runtime. This feature also abstracts away the low-level details of converting RMI calls into SOAP Requests. However, if the server interface changes, the programmer must manually change the client code and if new data types were introduced by the server, then the new WSDL document must be parsed and the appropriate client stubs must be generated and compiled. Therefore, Apache Axis is not suited for client development against a dynamic server interface without enhancements.

The Dynamic Invocation Interface (DII) [20] is an evolution of the CORBA-RMI model that facilitates the runtime construction of RMI calls. This model allows developers to forgo the creation of client stubs and instead construct RMI calls using *Request* objects that encapsulate all the information regarding the relevant server method. This is made possible by the use of an *Any* object, which can be configured to hold any object type. Although this feature allows developers to forgo any changes to the client backend on a server interface change, the client application itself may have to be changed and recompiled. Hence, DII does not reach the level of flexibility that we introduce in CDE.

The technologies that we have discussed hide low-level details of the RMI model, either by using an Integrated Development Environment (IDE) and or a well-defined API to invoke remote methods. Our CDE achieves this goal by allowing developers to use the local method invocation semantics to invoke remote methods. However, none of these RMI technologies addresses the issue of integrating dynamic changes of the server interface into live client instances. We build on the Apache Axis *Call* objects and the CORBA DII facility to address this issue through the mechanisms described in Section 5.6.

4. Developing Clients with CDE

Before discussing the middleware implementation details, we describe the interaction between JPie users and CDE in developing client applications.

To create a client application that uses SOAP, the user extends a provided class that acts as a gateway to the CDE system (Figure 3a). When the new subclass is being loaded into JPie, the CDE subsystem detects this and prompts the user to enter the WSDL location as shown in Figure 3b. CDE then adds stub methods for each method available in the server interface into the subclass. These stub methods act as any other method defined in the class from the user perspective. Figure 4 shows the stub methods and usage for a sample class.

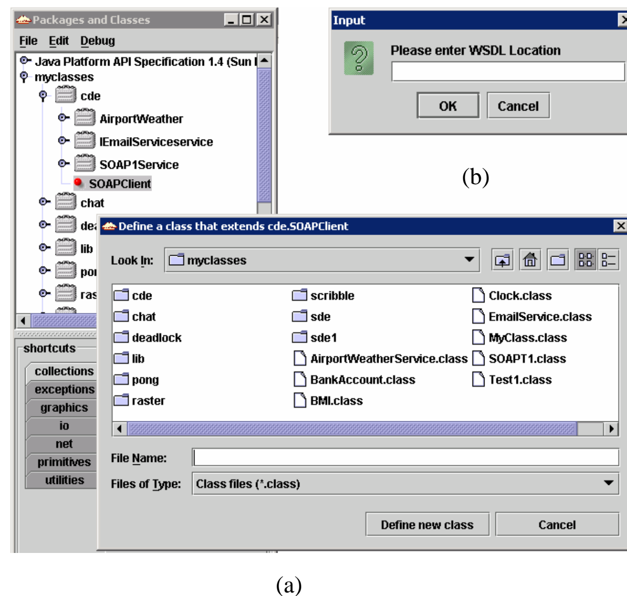

Fig. 3: A subclass of a technology specific class (e.g. SOAP or CORBA) must be created to interact with CDE as shown in (a). Once this class is created, the user is prompted for the relevant initialization information as shown in (b)

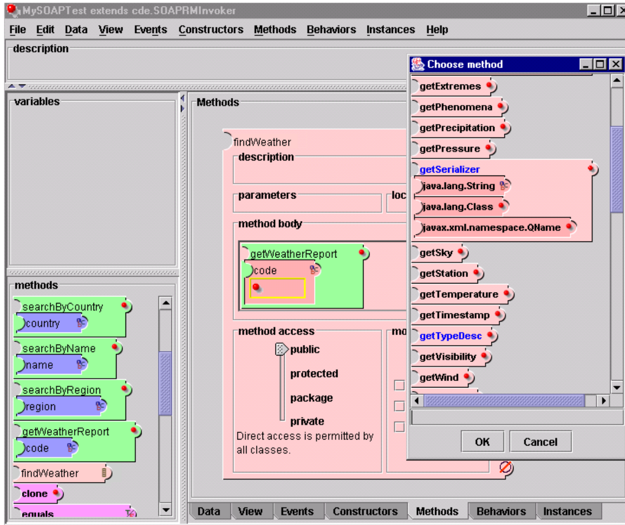

Fig 4: Stubs that represent server methods can be used similarly to any local method in the class.

To create a CORBA-RMI client, the user must subclass a different provided CDE gateway class and both the CORBA-IDL and IOR locations must be specified. The rest of the user interaction parallels the SOAP client development scenario. In both cases the information provided by the user is saved onto disk allowing CDE to automatically retrieve the relevant information, without user input, when the client class is reloaded.

As the server interface changes, CDE merges these changes into the relevant subclass. If the signatures of the stub methods are changed, and there are live instances of the client, then the JPie debugger will automatically prompt the user to resolve the conflicts (if any) caused by the change. If the server method represented by a stub method no longer exists, then the user is given the option of deleting the stub method. CDE will also inform the user when a parameter within a stub method must be removed.

Once the clients are loaded in JPie, the user can control the automatic updates as well as the update frequency using the CDE Manager Interface. In addition, the CDE Manager Interface allows users to force an immediate update and view the IDL that correspond to each client loaded in JPie.

5. CDE Architecture

In this section, we first introduce the high-level components of CDE by focusing on initialization, and information flow in method invocations. We present the SOAP and CORBA-RMI subsystems separately and compare them with the generic architecture models discussed earlier. Then we describe implementation details in the context of CDE's class hierarchy, which accommodates the two subsystems into a single framework. Finally, we present our strategy for integrating server-side interface changes into live client instances.

5.1. SOAP Subsystem Overview

As seen in Figure 5, the SOAP subsystem consists of five high-level client components. The CDE Manager oversees the subsystem initialization as well as integration of server interface changes for all client applications. The SOAP RM Invoker acts as the base class for dynamic classes that interact with the SOAP subsystem. The WSDL Parser is in charge of providing the CDE Manager with the available server methods by parsing the WSDL document, as well as translating method calls provided by the SOAP Remote Caller into SOAP Requests. The SOAP Remote Caller sends SOAP Requests and translates SOAP Replies into return objects. Finally, the Request Handler acts as a communication protocol independent liaison between the SOAP RM Invoker and the SOAP Remote Caller.

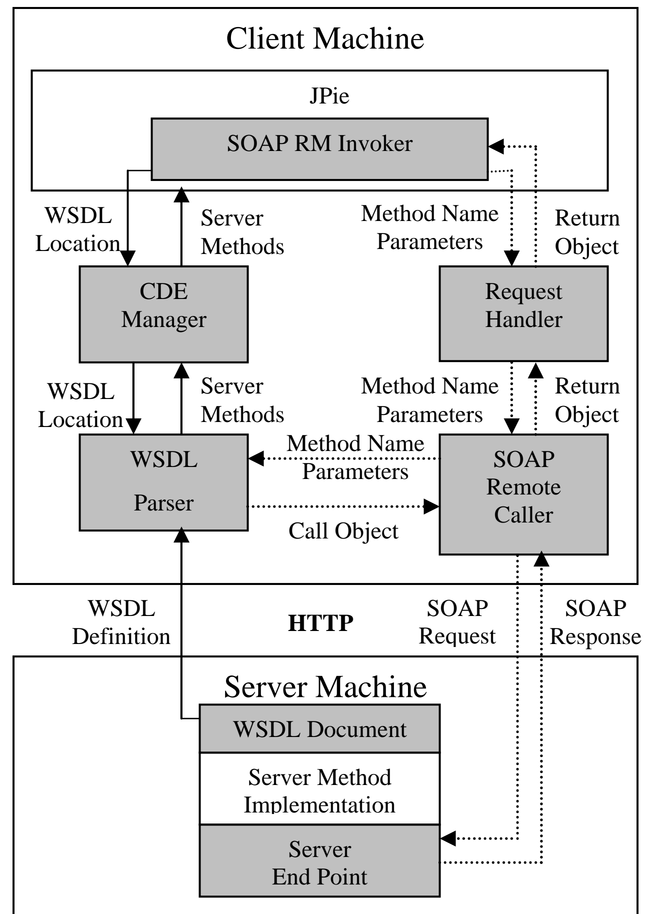

Fig. 5: There are two information paths in the SOAP Subsystem. The solid lines show the path used in updating the server interface and the dotted lines show the path used in invoking server methods.

5.1.1. Initialization. The CDE Manager is initialized when a subclass of SOAP RM Invoker (or the equivalent CORBA-RMI class as discussed in Section 5.2.1) is first loaded in JPie. This occurs when the user either opens an old client class or initiates the creation of a new client class as discussed in Section 4.

In order to reduce the workload placed on the CDE Manager, the initialization sequence is designed to be initiated by the CDE Manager and carried forward by the rest of the components in the subsystem. When a user extends the SOAP RM Invoker to create a dynamic class within JPie, an event is generated to signal the CDE Manager to include the new dynamic class in its list of managed classes. Then the CDE Manager prompts the user for the location of the WSDL document. Using that location, the CDE Manager creates a WSDL Parser, and passes a reference to that parser back to the SOAP RM Invoker. The SOAP RM Invoker then creates the Request Handler and the SOAP Remote Caller using that reference. During the internal initialization of the WSDL Parser, it fetches the WSDL document from the server, and parses the WSDL to generate a list of server methods. The CDE Manager uses this list to include the available server methods within the corresponding dynamic class as described in Section 5.5.

5.1.2. Server Interface Updates. Although we could have used a model where the WSDL is examined only on user demand, we chose an approach where the CDE Manager periodically prompts the WSDL Parser to check for changes in the WSDL document after initialization. If changes are detected, the document is parsed and the new list of methods is used by the CDE Manager to integrate the changes with the corresponding dynamic class using the mechanism described in Section 5.6. This approach ensures the consistency of the server interface to a higher degree and relieves the user from the burden of periodically prompting for updates. If the server supports a subscription model, then events from the server could be used to trigger updates.

5.1.3. Request/Response Handling. The RMI call path within both SOAP and CORBA subsystems was designed to maximize a separation of concerns as described in Section 5.3. In the SOAP subsystem, the dynamic class, which is a subclass of the SOAP RM Invoker, passes the server method invocations to the Request Handler. The Request Handler receives the call, extracts the method name and parameter details, and passes them to the SOAP Remote Caller. The SOAP Remote Caller passes that information to the WSDL Parser, which returns an Axis Call object encapsulating the call. The successful invocation of the Call object generates a return object. The SOAP Remote Caller inspects the Call object to detect whether an exception has occurred. If an error is not detected, the return object is passed back to the Request Handler. If an error has occurred, then a new exception that encapsulates the error is sent back to the Request Handler. The Request Handler forwards the return value or exception back to the dynamic class.

5.2. CORBA-RMI Subsystem Overview

The CORBA subsystem is structurally similar to the SOAP subsystem. However, there are differences in the interaction among components. Figure 6 shows the structure and information flow in the CORBA subsystem.

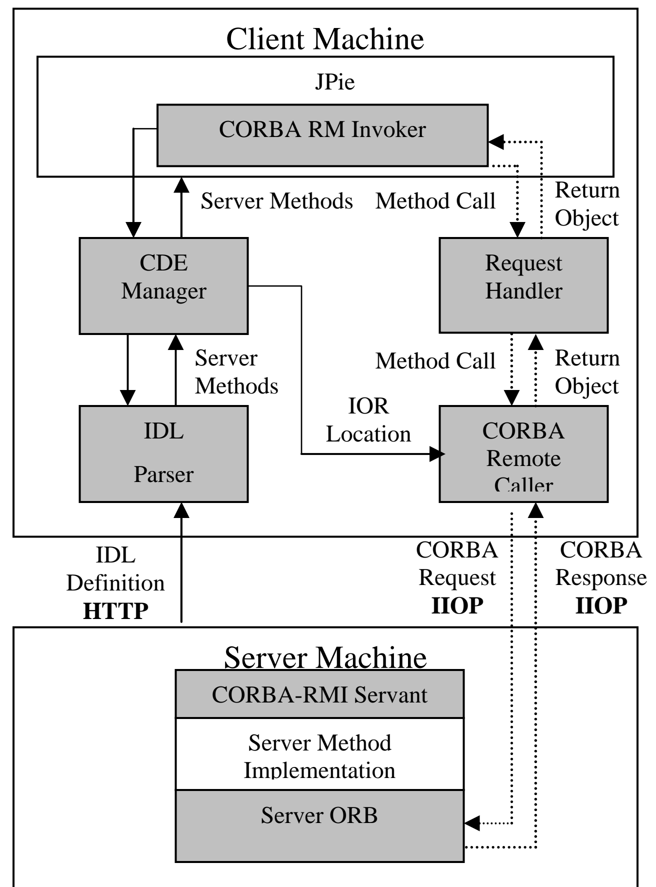

Fig. 6: There are two information paths in the CORBA Subsystem. The solid lines show the path used in updating the server interface and the dotted lines show the path used in invoking server methods.

5.2.1. Initialization. Once again, the initialization of the CDE manager is performed under the same circumstances described in Section 5.1.1. As before, we strive to decrease the workload of the CDE manager during initialization by decoupling the initialization of the Client ORB and the IDL Parser. When a user extends the CORBA RM Invoker to create a dynamic class within JPie, an event is generated to signal the CDE Manager to include the new dynamic class in its list of managed classes. When this event is

detected, the CDE Manager prompts the user for the location of the IDL and the IOR. The CDE Manager uses the IDL location to create the IDL Parser and the IOR location to create the CORBA Remote Caller, which uses the IOR fetched from that location to initialize the Client ORB. A reference to the CORBA Remote Caller is passed back to the CORBA RM Invoker to be used in initializing the Request Handler. During the internal initialization of the IDL Parser, it fetches the IDL document from the server and parses it to generate a list of server methods. The CDE Manager uses this list to include the available server methods, within the corresponding dynamic class, through the CORBA RM Invoker.

5.2.2. Server Interface Updates. We chose our update model to mirror the model used in the SOAP subsystem since the concerns discussed in Section 5.1.2 are still valid for the CORBA subsystem. After initialization, the CDE Manager periodically prompts the IDL Parser to check for changes in the IDL document. If changes are detected, the document is parsed and the new list of methods is used by the CDE Manager to integrate the changes with the corresponding dynamic class using the mechanism described in Section 5.6. Again, server events could also be used to trigger updates.

5.2.3. Request/Response Handling. Once again, the components that take part in making RMI calls mirror the components used in the SOAP subsystem. In this case, the dynamic class that is a subclass of CORBA RM Invoker passes the server method invocations to the Request Handler. The Request Handler passes the call to the CORBA Remote Caller. The CORBA Remote Caller uses that information to make a method call on the Client ORB. The Client ORB sends the request out over IIOP and receives a reply object. Then it simply passes this object back to the Request Handler (through the CORBA Remote Caller) if no errors have occurred. If an error has occurred, then a new exception is generated using the available error information and sent back to the Request Handler. The Request Handler forwards the return object or exception back to the dynamic class.

5.3 Class Hierarchy

To implement the components described in section 4.1 and 4.2, we designed a class hierarchy that allows SOAP, CORBA, and other technologies to be easily integrated into the system. This allows key components such as the CDE Manager to be technology independent. Figure 7 shows the three interfaces that each technology must implement. Each interface provides the blueprint to a component that performs a critical role within the CDE architecture. The Request Handler's role as the technology independent focal point of CDE is also highlighted.

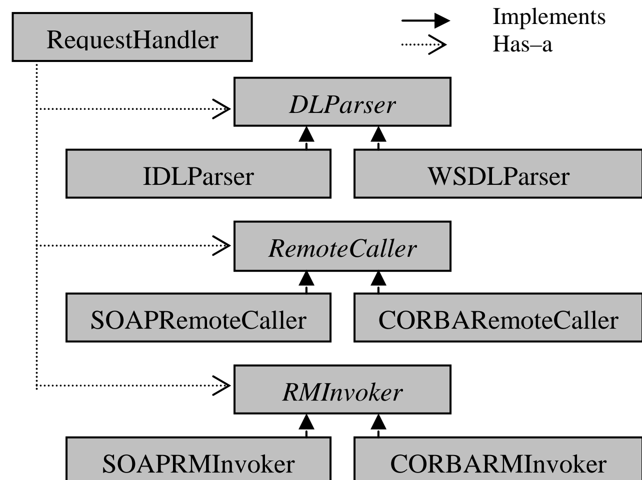

Fig. 7: Each technology incorporated into CDE must implement a parser for the server interface, an adapter that will convert a Java method call to the appropriate request and the response to the relevant Java type, and an extensible class that will serve as the base type for dynamic classes using that technology.

5.4. Concurrency in Client Applications

CDE supports concurrency in client applications by allowing multiple active RMI calls to exist at any given time. In the SOAP subsystem, we achieve this by creating a new Call object for every request that the SOAP Remote Caller receives. Each call object creates a new connection to the server, and hence its performance is not affected by other Call objects that are active at that time.

For the CORBA subsystem, we used the multithreading capabilities of the OpenORB implementation to directly support multithreading in the client application.

5.5. Representing Remote Methods in JPie Clients

As discussed in Section 2.3, instances of the class DynamicMethod represent the methods defined in dynamic classes. In JPie, not only can dynamic methods be added and removed on the fly, but their formal parameter lists and method bodies can be changed dynamically as well. In the case of CDE, we wanted special dynamic methods that could be added and removed, and whose formal parameter lists could be changed in response to server-side changes. However, we did not want the CDE user to be able to change the parameter lists or to open and edit the method bodies since these are defined by the server. Therefore, we extended DynamicMethod to define CDEStubs, which embody a client-side implementation for generating server requests corresponding to a particular server method. We add CDEStubs to each appropriate subclass of

SOAPRMInvoker or CORBARMInvoker to represent methods that can be called on the corresponding server. These CDEStubs are automatically added, removed, and modified over time by CDE in order to maintain consistency with the server's published interface.

5.6. Dynamic Changes to the Server Interface

When method signatures in the server interface change, CDE aggressively attempts to make the corresponding changes in the client applications as seamlessly as possible as follows.

The CDEManager performs all the changes, additions, and deletions of CDEStubs in all subclasses of SOAPRMInvoker and CORBARMInvoker. When the DLParser parses the DL document, it creates a list of CDEMethods, where each CDEMethod holds the method signature of the corresponding server method. The CDEManager uses a matching algorithm, as explained below, to pair up each existing stub on the client side with the closest matching method in the newly published server interface. This automates the process of merging the differences between the elements in this list and the list of current CDEStubs in the dynamic class.

5.6.1. Considerations in Interface Matching. Care must be taken in resolving differences between the current list of CDEStubs and the new list of CDEMethods. Suppose a JPie user creates a Java statement that calls a remote method, which subsequently changes on the server. If the CDEManager simply deleted the old method and replaced it with a new one, the user would have to manually edit the call site to call the new method, transferring the appropriate parameter values from the method call no longer in use. However, if the CDEmanager is able to match up the old calls to the new ones, reordering actual parameters and removing extra actual parameters as needed, the user can simply be alerted through the built in debugger in JPie to add values for newly added formal parameters.

5.6.2. Matching CDEStubs and CDEMethods. We use the Stable Marriage algorithm [21] to match CDEStubs and CDEMethods. To support this operation, we use a new Java interface called MethodHolder that is implemented by both CDEStub and CDEMethod. Since the Stable Marriage algorithm expects to match up elements sets of equal size, we use a new Java class, named TempMethodHolder that implements MethodHolder, to pad the sets when the number of CDEMethods and CDEStubs are not equal.

For comparison of CDEStubs and CDEMethods during the Stable Marriage algorithm, we use a heuristic ranking system for both CDEStubs and CDEMethods. To accomplish this, each CDEMethod forms a set of rankings by calculating a ranking for each element in the CDEStub list. We then perform the same operation for each element in the CDEStub list.

To calculate the rankings, we first assign weights to corresponding attributes between sets of two MethodHolders taken from opposite lists. A match in the method name carries the highest possible weight. The next height weight is assigned to matching return types and the third highest weight is assigned for matching parameters that have the same type. The smallest weight is available if the parameter name is the same. If an exact match is not found for each of these instances, then a weight of zero is assigned for that attribute. The total weight for each MethodHolder is the sum of its attribute weights. The rankings are then calculated by each MethodHolder by ordering the corresponding MethodHolders in the order of decreasing total weights. Thereby, TempMethodHolders rank all the corresponding MethodHolders (CDEStubs or CDEMethods) the same and both CDEStubs and CDEMethods assign the worst possible ranking to TempMethodHolders.

Once the Stable Marriage algorithm has generated the required mapping, the CDEManager must decide how to add, update, or delete CDEStubs according to how they were matched. There are three cases.

Case 1 - A CDEStub was matched with a TempMethodHolder (i.e. the server has discarded the method): In this case, we inform the user before removing the corresponding CDEStub from the dynamic class.

Case 2 - A CDEMethod was matched with a TempMethodHolder (i.e. the server has added a method): In this case, we simply construct a new CDEStub using the method signature of the CDEMethod and add it to the dynamic class.

Case 3 - A CDEMethod was matched with a CDEStub (i.e. the server has modified a method): This case may indicate a change to the method name, return type or the formal parameter list.

- Case 3.1 - The method name has changed: We simply reflect this change in the CDEStub.
- Case 3.2 - The return type has changed: We reflect this change in the CDEStub and inform the user.
- Case 3.3 - The formal parameter list has changed: We use the Stable Marriage algorithm to match the new parameters with the old ones.

5.6.3. Changing Parameters of a Stub. For Case 3.3, we introduce a Java interface named ParameterHolder and three classes named StubParameter, MethodParameter, and TempParameter that implement this interface. We then construct MethodParameter and StubParameter lists using the CDEMethod and the CDEStub. If the numbers

of StubParameters and MethodParameters are unequal, TempParameters are added to the shorter list.

The rankings for the stable marriage algorithm are once again based on aggregate weights. The highest attribute weight is available if the Java types of two ParameterHolders match. A smaller attribute weight is added if the names of the parameters match. The aggregate weights are calculated for each ParameterHolder in the opposite list and the rankings are assigned based on the decreasing order of total weights. This means that the TempParameters are assigned the worst ranking by both StubParameters and MethodParameters. TempParameters assign the same ranking to all other ParameterHolders.

Once the Stable Marriage algorithm has finished matching up the parameters, the CDEManager must decide how to add, update, or delete CDEStubs according to how they were matched. There are three cases.

Case 3.3.1 - A StubParameter was matched with a TempParameter: In this case, we inform the user before removing the corresponding parameter from the CDEStub.

Case 3.3.2 - A MethodParameter was matched with a TempParameter: In this case, we simply add a new parameter to the corresponding CDEStub.

Case 3.3.3 - A MethodParameter was matched with a StubParameter: This case implies that the parameter name, the parameter type, or both have changed. In all these cases we simply apply this change to the corresponding parameter in the CDEStub and inform the user to make the appropriate changes in the actual parameter expressions.

Many changes made to the dynamic class with this mechanism are virtually transparent to the JPie user. User input is only required when a CDEStub is to be deleted or when a parameter within the CDEStub must be changed or deleted. If there is a missing actual parameter due to a new formal parameter being added, this will be detected by the JPie debugger at the next execution of that method call and the execution will pause while the user supplies the appropriate arguments.

If the client application was offline while the server interface was changed, then CDE will merge the changes at the next startup of the client application according to the mechanism we presented earlier.

6. Performance

Since CDE introduces a level of complexity into the RMI call structure, an increase in the round trip time (RTT) of a RMI call is inevitable. We were conscious of this fact during the design of CDE and further experimentation has shown that this decrease is within reasonable bounds.

To gauge the performance of CDE we measured the average RTT of SOAP calls to three web services found at the XMethods [22] online web services directory. Each web service was built using a different implementation of SOAP as indicated in Table 1.

We measured the RTT using three different techniques. First, we used a simple Java program that measured the average time taken to send a predefined SOAP request to the server and receive the relevant reply. Second, we used a static Apache Axis client application that measured the average RTT by making a RMI call using the same predefined parameter values. Third, we used a CDE client and the same predefined parameter values to measure the average RTT. To measure the time we used Java's *getTimeInMillis* system call and the average time was calculated over one hundred trials.

Table 1: RTT times for SOAP Calls

Server	RTT (seconds)		
	Java Client	Axis Client	CDE Client
AirportInfo (.NET)	1.66	2.00	2.40
AirportWeather (CapeConnect)	2.00	2.35	2.59
WHOIS (Axis)	2.88	3.25	3.57

At the end of the development phase the dynamic CDE client can be converted into a static Axis client through the JPie application export mechanism. Hence, the performance overhead introduced by CDE is only present during the development phase. In this context, the per-call overheads listed in Table 1 are acceptable.

Although we have not carried out comprehensive testing for the CORBA subsystem, we expect the performance overhead to similar. Further, experimentation is also needed to gauge the performance of CDE and JPie as the number of managed clients increase.

7. Conclusions

This paper introduced the concept of live client development using the RMI model. We also presented a novel approach to presenting the server interface to the developer that simplifies RMI development into a natural extension of mainstream Java application development.

One of our goals for CDE was to reduce the learning curve involved in developing distributed applications using the RMI model. The construction of Client applications for fully developed web services and CORBA servers using CDE, provides developers a virtually local application development experience by mimicking the local method invocation structure for RMI. We have constructed a variety of sample applications, using established servers [23] that use different implementations of SOAP and CORBA, to test the validity of our claim.

Our experience indicates a significant reduction in development time (including the JPie setup time) from the traditional modes of distributed application development.

Our second goal of integrating changes in the server interface into live client instances has also been successfully implemented in CDE. We tested our implementation through our Server Development Environment (SDE) [24], which was a natural extension to CDE. CDE and SDE combine to support live client and server development effectively.

An additional feature that is being investigated is the ability to interchange the technology being used to communicate between the client and the server while live development and information exchange is taking place. Although CapeConnect and other SOAP to CORBA bridging technologies [25] offer technology bridging capabilities, we feel that live modification will result in a more fluid development experience. We are currently implementing a medium-sized mail service application in JPie using CDE. Our experience with that application will help motivate future work on CDE, SDE, and JPie in general.

Acknowledgements

We thank Dr. Christopher Gill and Michael Plezbert for their support during our background study. This work was supported in part by the National Science Foundation under CISE Educational Innovation grant 0305954.

References

[1] World Wide Web Consortium, Simple Object Access Protocol (SOAP) 1.2, http://www.w3.org/TR/SOAP/, June 2003.

[2] Object Management Group, Common Object Request Broker Architecture: Core Specification, 3.0.3 ed., http://www.omg.org/docs/formal/04-03-01.pdf, March 2004.

[3] Kenneth J. Goldman. Live Software Development with Dynamic Classes, Aug. 2004, Submitted for publication.

[4] Apache Software Foundation, Apache Web Services Axis Project, http://ws.apache.org/axis/index.html, March 2004.

[5] The Community OpenORB Project, http://openorb.sourceforge.net, March 2004.

[6] Wide Web Consortium, Extensible Markup Language (XML) 1.0, 3 ed., http://www.w3.org/TR/REC-xml, Feb. 2004.

[7] Wide Web Consortium, Web Services Definition Language (WSDL) v.1.1, http://www.w3.org/TR/wsdl, Mar. 2001.

[8] Wide Web Consortium, Hypertext Transfer Protocol (HTTP), http://www.w3.org/Protocols, April 2004.

[9] Common Object Request Broker Architecture (CORBA/IIOP) Specification, v. 3.0.2, http://www.omg.org/technology/documents/formal/corba_iiop.htm, Mar. 2004.

[10] Kenneth J. Goldman. An interactive environment for beginning Java programmers. Science of Computer Programming, 53(1):3–24, October 2004.

[11] Microsoft Corporation, Microsoft Visual Studio .NET Overview, http://msdn.microsoft.com/vstudio/, Mar. 2004.

[12] Microsoft Corporation, .NET Framework 1.1 Overview, http://msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx, Mar 2004.

[13] Strahl, R, Creating and using Web Services with the .NET framework and Visual Studio.Net, http://www.westwind.com/presentations/dotnetwebservices/DotNetWebServices.asp, July 2002.

[14] Cape Clear Software Ltd, Cape Clear Business Integration Suite Version 4 Overview, http://www.capeclear.com/products/index.shtml, Dec. 2003.

[15] Sun Microsystems, Enterprise JavaBeans Specification, v.2.1, Nov. 2003.

[16] Netscape Communications Corporation, JavaScript Specification, v.1.1, Mar 2001.

[17] Cape Clear Software Ltd, Using Web Services with SOAPDirect, http://www.capeclear.com/products/manuals/three/Users_Guide/html/soapdirect_three_clients.html, Mar 2004.

[18] Apple Computer, Inc, Developing Direct to Web Services Applications, v 1.0, Nov 2002.

[19] Apache Software Foundation, Apache Axis API Documentation, http://ws.apache.org/axis/java/apiDocs/org/apache/axis/client/Call.html, Jan. 2004.

[20] Kris Magnusson et. al. Java Enterprise in a Nutshell. O'Reilly, 2002.

[21] D. G. McVite and L.B Wilson. The Stable Marriage Problem, Communications of the ACM Volume 14 , Issue 7. July 1971, pp. 486 – 490.

[22] XMethods, http://www.xmethods.com/ve2/index.po;jsessionid=9YCpuT-mGMAOb5JjUAe-E7gl(QhxieSRM), Mar. 2004.

[23] Sajeeva L. Pallemulle, Kenneth J. Goldman. Live Development of SOAP and CORBA Servers, In preparation.

[24] SOAP to CORBA Bridging Software Project. Mar 2004. Open Source Development Network. http://sourceforge.net/projects/soap2corba, Mar 2004.