

# Supporting Live Development of SOAP and CORBA Servers

Sajeeva L. Pallemulle, Kenneth J. Goldman, and Brandon E. Morgan  
Washington University in St. Louis  
Campus Box 1045, One Brookings Drive  
St. Louis, MO 63130-4899, USA  
{sajeeva, kjg, bem2}@cse.wustl.edu

## **Abstract**

*We present middleware for a Server Development Environment that facilitates live development of SOAP and CORBA servers. As the underlying implementation platform, we use JPie, a tightly integrated programming environment for live software construction of Java applications. 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 by automating the server deployment process, allowing developers to devote their full attention to the implementation of server logic. Moreover, the live development model enables the construction of server applications while they are running, connected, and communicating with test clients. Combined with our Client Development Environment, these features facilitate the live, simultaneous construction of both the client and server applications.*

## **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, simultaneous development depends upon both endpoints having a consistent view of the common interface. 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, an approach that combines client and server development into a single unified activity is attractive in streamlining application

development and ensuring interface consistency between the client and server.

We present a Server Development Environment (SDE) 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 build upon JPie to support live server development. Namely, we automatically detect additions, deletions and mutations in the set of server operations to update the server interface description as needed. Further, we completely abstract away the low level deployment details by automating the publication of the server interface description and the creation of server backend components, so developers can concentrate on the server logic. In conjunction with our Client Development Environment (CDE) [4], this results in a live integrated development process in which the client and server applications can be developed simultaneously. To preserve consistency, live changes in the server's interface are reflected in the running client program.

Our architecture supports technologies such as SOAP and CORBA that use an interface definition language (IDL) to communicate the server interface to the clients. SDE supports SOAP by building upon the Apache Axis [5] implementation and supports CORBA using OpenORB [6]. Our design can be extended to support other technologies that use IDLs.

This paper makes several contributions. We introduce techniques for automated server deployment, automated publication of the server interface, and the detection of stable changes in the server implementation. In addition, we present the design and implementation of a mechanism implemented jointly by SDE and CDE, which facilitates live, simultaneous client-server development.

The remainder of the paper is organized as follows. Section 2 provides background on distributed application development in SOAP and CORBA and presents brief overviews of JPie and CDE. Section 3 provides an overview of related work. Section 4 focuses on the SDE

user interaction mechanism for creating server applications. In Section 5, we present the SDE architecture and discuss the mechanisms used to create backend components and automate the publication of the server interface. Section 6 focuses on live, simultaneous client server development and the interaction between SDE and CDE. In Section 7 we discuss the performance and overhead of using SDE. We conclude, in Section 8, with a summary and directions for future work.

## 2. Background

For our initial implementation of SDE, we decided to concentrate on SOAP and CORBA. We chose two technologies to ensure that the design was sufficiently extensible to support multiple technologies. Both SOAP and CORBA use interface definition languages yet have different overall frameworks. This section presents background on SOAP and CORBA, as well as on JPie and CDE.

### 2.1. SOAP

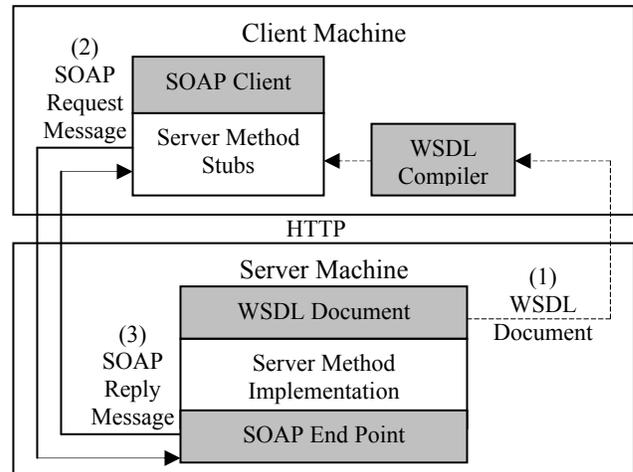
Servers that use SOAP are popularly known as Web Services. Web Services use the Extensible Markup Language, (XML) [7] to present the server interface to the clients as well as to communicate with those clients.

As shown in Figure 1, when a Web Service is established, it uses the Web Services Definition Language (WSDL) [8] standard to publish a WSDL document that potential client applications can use to gather information required to invoke methods on the Web Service.

WSDL is an XML-based schema that contains information such as the Web Service location, the methods available for remote invocation on that Web Service, and details on invoking 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 objects or data structures with clients as parameters and/or return values.

The client applications use the information published in the WSDL document to form an 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. It then constructs a 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.



**Figure 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.

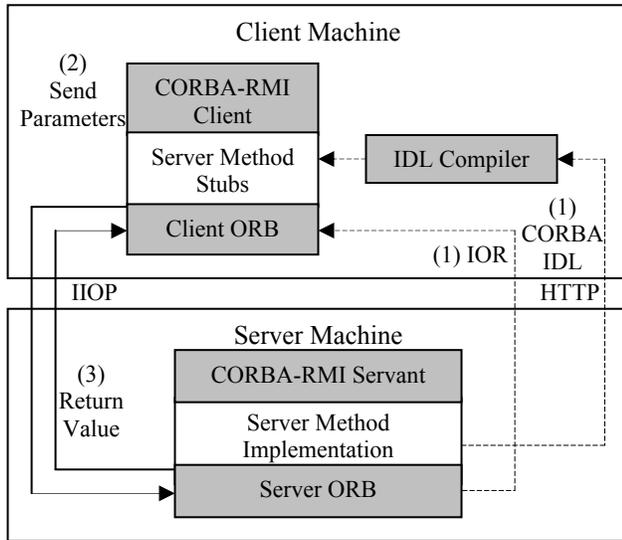
The underlying transport medium that supports this publish-request-response mechanism is provided by the Hyper Text Transport Protocol (HTTP) [9].

### 2.2. CORBA-RMI

The Common Object Request Broker Architecture (CORBA) defines a high-level communication model for distributed computing. In 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) [3]. 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. ORBs use IIOP (Internet Inter-Orb Protocol) [3] 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) [10, 11] to describe object interfaces and an Interoperable Object Reference [3] (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 *module* element is the root element of any CORBA-IDL document. An *interface* element, similar to a Java class, encapsulates instance variable declarations and method declarations. The *module* may contain uniquely identified *interfaces*.



**Figure 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 through 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 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 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 [12]. 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 take additional steps to update the calls. One of the important goals of the present work is to offer this level of consistency among the client and server applications through a live, simultaneous client-server development methodology.

### 2.3. Client Development Environment (CDE)

CDE [1] supports the live construction of SOAP and CORBA clients. In CDE, we extend JPie's live development model to automate addition, mutation, and deletion of dynamic server methods within dynamic clients. CDE 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. The current CDE implementation uses Apache Axis for SOAP support and the Dynamic Invocation Interface (DII) [3] implementation of OpenORB [6] as the basis for its CORBA support. In Section 6, we discuss a protocol jointly implemented by both CDE and SDE to support live, simultaneous client-server development.

## 3. Related Work

Although RMI is a natural extension of standard method call semantics, setting up the development tools for technologies such as SOAP and CORBA can be daunting. Therefore, development environments that encapsulate the low-level details and the execution environment have proven popular. In this section, we discuss several existing approaches for streamlining distributed application development using RMI.

Visual Studio.Net [13] builds upon the Microsoft .NET framework [14] to provide a number of mechanisms that reduce the Web Services development time. Visual Studio.Net provides automatic generation of a rudimentary Active Server Pages (ASP) [15] web client at each

deployment step of the Web Service. Through this rudimentary web client, developers can test the server manually prior to creating a client program. Since there is no actual client program at that point, dynamic server interface updates are not needed. However, once an actual client program is under development, the automatic ASP is no longer useful. Instead, whenever the server interface changes, the developer must obtain the new interface, manually change the client code to reflect the new interface, and then recompile and restart the client to continue testing.

Apache Axis can be combined with the Apache Tomcat Servlet Engine [16] to achieve a fast, automated deployment process for Web Services. The Axis implementation provides the Java2WSDL and WSDL2Java tools that can be used to generate the WSDL document, deployment descriptors used by Tomcat, and the server-side stub classes. The server stub classes can be modified to include the server method definitions. Finally, the source file can be included in the appropriate path within the Servlet engine to take advantage of the Axis/Tomcat automated Web Service deployment mechanism. Although the Axis implementation does not directly address the issue of dynamic changes, its design is flexible enough to incorporate this feature, and both CDE and SDE employ Axis tools.

WebObjects [17] is another platform that facilitates simplified development of Web Services. The Direct to Web Service [18] component of WebObjects incorporates a Web Services Assistant based on the Apache Axis implementation of SOAP. This tool supports defining methods using a provided GUI. Direct to Web Service is particularly suited for building a Web Services front end to a database. The Web Services Assistant provides tools that help developers map database calls into Web Service operations. WebObjects is designed for development of Web Services against a fixed interface and does not address the issue of dynamic server interface changes.

The BEA Tuxedo [19] development environment simplifies CORBA server development by providing a number of tools that automatically generate backend components as well as method stubs that can be mutated to implement server logic. It provides a number of additional tools such as a naming service and secure communication mechanisms. However, the deployment process is much more involved and requires a thorough understanding of low level details. Therefore, Tuxedo can be considered as a tool geared toward experienced programmers for developing robust CORBA applications. The design of Tuxedo does not allow for dynamic server interface upgrades due to the lack of automation in the deployment process and the presence of static server classes.

The technologies that we have discussed hide low-level details of the RMI model by using an Integrated Development Environment (IDE) and/or a well-defined

API to abstract away deployment details. Our SDE furthers this goal by completely relieving the programmer of the need to deploy the application. In addition, SDE employs a publication strategy (See Section 5.6) to automate the publication of the server interface as needed. Moreover, the combination of SDE and CDE provides the additional functionality of live, simultaneous development.

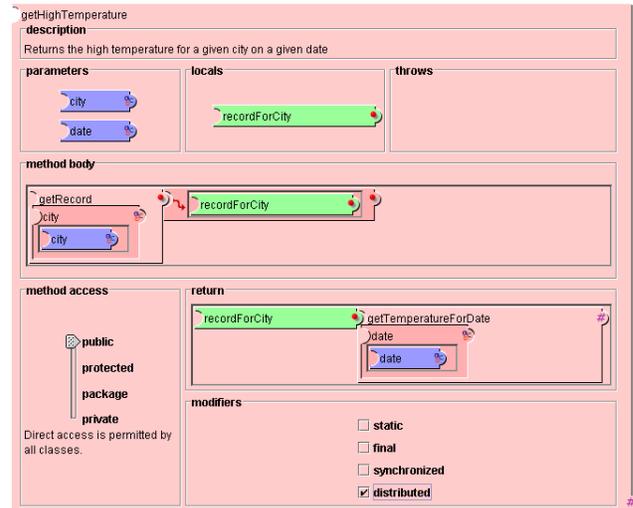
#### 4. Developing Servers with SDE

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

To create a server application that uses SOAP, the JPie-SDE user extends a provided class, called SOAPServer that acts as a gateway to the SDE system. When the new subclass of SOAPServer is being loaded into JPie, the SDE subsystem detects this and creates the required backend components for deployment and immediately publishes a basic WSDL definition that is useful in live, simultaneous client-server development (See Section 6).

To create a CORBA-RMI server, the JPie-SDE user must extend a different provided SDE gateway class called CORBAServer. As soon as the class is created, a basic CORBA-IDL document and an IOR are published to enable live, simultaneous client-server development. The rest of the user interaction parallels the SOAP client development scenario.

To add a method declared in the dynamic class to the server interface, the user selects the ‘distributed’ modifier from the modifier list as shown in Figure 3.



**Figure 3:** Live class modifications in JPie are by direct manipulation of graphical representation of programming constructs. To include a method in the server interface the user selects the ‘distributed’ modifier.

SDE is able to detect distributed methods by inspecting the ‘distributed’ modifier. New server interface descriptions are published as changes are made to the method signatures of these distributed methods. To remove a method from the server interface, the user can either delete the method or deselect the ‘distributed’ modifier.

Once SDE starts monitoring a subclass of SOAPServer or CORBAServer, the user can control the automated server interface publication using the SDE Manager Interface. The user can control the publication frequency by specifying a timeout value (see Section 5.6). In addition, the SDE Manager GUI allows users to control the integrated HTTP server used to publish server interfaces. The user may also view the WSDL/CORBA-IDL that corresponds to each server under development.

## 5. SDE Architecture

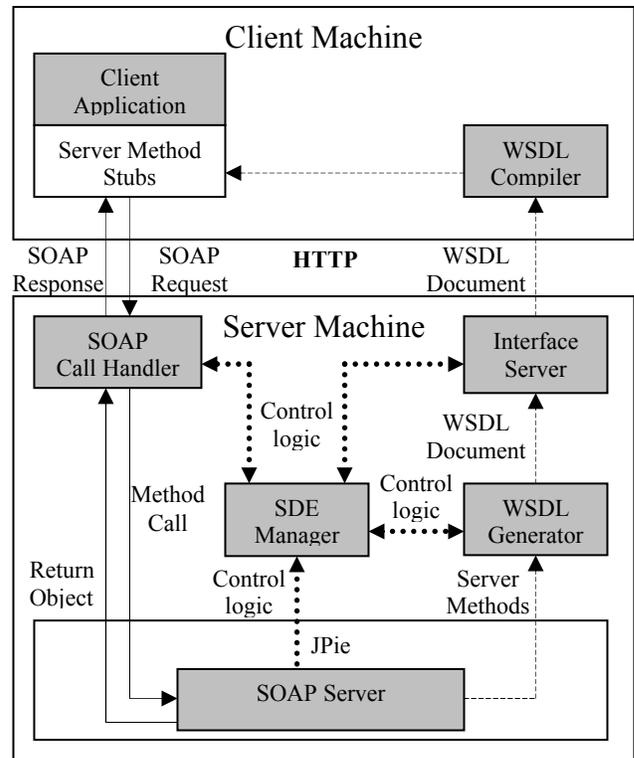
SDE has three main responsibilities. It must detect the presence of server classes within JPie, construct and deploy the RMI call handlers for each of those classes, and automate the publication of the server interface in an intelligent manner. In conjunction with CDE, SDE must also provide concurrency control between the RMI call path and the server interface update mechanism.

In this section, we first introduce the high-level components of SDE 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 in Section 2. We proceed with a description of the implementation details in the context of SDE’s class hierarchy, which accommodates the two subsystems into a single framework. We then discuss the concurrency control mechanisms that we employ to handle interleaving of server method updates and server method calls. Finally, we present the strategy for detecting server interface changes and determining the frequency of publication.

### 5.1. SOAP Subsystem Overview

As seen in Figure 4, the SOAP subsystem consists of five high-level client components. The SDE Manager oversees the subsystem initialization and acts as the central point of communication between the other components.

The SOAP Server acts as the base class for dynamic classes that interact with the SOAP subsystem. The WSDL Generator is in charge of detecting the addition, deletion, and mutation of server methods within the SOAP Server instance and creating new WSDL documents as required. The Interface Server acts as a simple HTTP server that publishes the WSDL documents to the public domain. Finally, the SOAP Call Handler acts as the communication end point that performs the translation between SOAP and Java for remote method invocations.



**Figure 4:** The SOAP Subsystem has three main information paths. The dashed lines represent the path used in publishing the server interface. The solid lines represent the path used in servicing remote method calls. The dotted lines represent the flow of control information within the subsystem.

**5.1.1. Initialization.** When a user extends the SOAP Server to create a dynamic class within JPie, an event is generated to signal the SDE Manager to include the new dynamic class in its list of managed classes. Then the SDE Manager creates both a WSDL Generator and a SOAP Call Handler, passing a reference to the SOAP Server to each component. The WSDL Generator registers itself as a listener to changes in the method signatures within the SOAP Server and creates a minimal WSDL document<sup>1</sup> by obtaining the endpoint address from the SOAP Call Handler through the SDE Manager.

**5.1.2. Server Interface publication.** To determine whether to update the WSDL definition, we employ a notification mechanism where the WSDL Publisher listens for changes being made on the SOAP Server instance. This mechanism is discussed in detail in Section 5.6. As discussed in Section 5.7, outdated RMI calls may also trigger updates to the WSDL document. Once the new

<sup>1</sup> The minimal WSDL document contains the SOAP Endpoint address but does not contain any server operation definitions.

WSDL Document is produced it is simply forwarded to the Interface Server for publication.

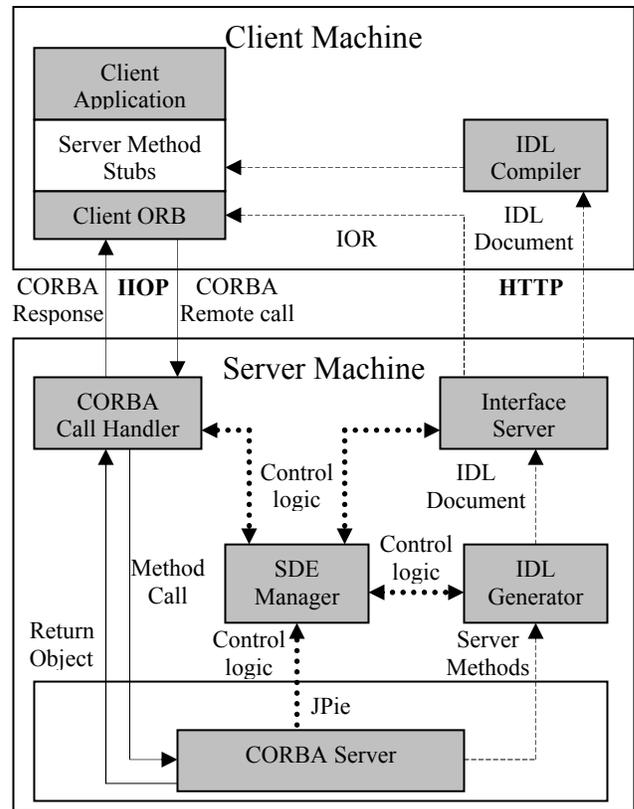
**5.1.3. Request/Response Handling.** The RMI call path within both SOAP and CORBA subsystems was designed to maximize the separation of concerns as described in Section 5.4. In the SOAP subsystem, if calls arrive before an instance of the SOAP Server class has been created, the SOAP Call Handler immediately sends a reply containing a SOAP Fault with a ‘Server not initialized’ message. After initialization, the SOAP Call Handler parses incoming SOAP Requests to create method calls that can be invoked on the SOAP Server instance. If the SOAP Request is malformed, a SOAP Fault with a ‘Malformed SOAP Request’ message is sent to the client. If a method call is successfully created, the SOAP Call Handler searches for a matching method in the current server interface. If a match is found, then that method is invoked on the SOAP Server instance, and if an exception is not thrown, the result is encoded in a SOAP Response and sent to the client. If an exception is thrown during the execution of the server method, a SOAP Response containing a SOAP Fault that encapsulates the exception is sent to the client. If the method call does not match any method in the current server interface, the SOAP Call Handler forces a server interface update if necessary (See Section 5.7) and then sends a “Stale Method” fault to the client.

## 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. In the CORBA subsystem, the CORBA Call Handler is a simple wrapper around the Server ORB, and the low level communication details are handled by making OpenORB API calls. The same Interface Server is used by both subsystems for simplicity. Figure 5 shows the structure and information flow in the CORBA subsystem.

**5.2.1. Initialization.** Initialization of the SDE manager is performed under the same circumstances described in Section 5.1.1. When a user extends the CORBA Server to create a dynamic class within JPie, an event is generated to signal the SDE Manager to include the new dynamic class in its list of managed classes. The SDE Manager creates both an IDL Generator and a CORBA Endpoint, passing a reference to the CORBA Server to each component. The IDL Generator registers itself as a listener to changes in the method signatures within the CORBA Server and creates a minimal CORBA-IDL document<sup>2</sup>. The Server ORB is initialized by the CORBA End Point and finally the IOR is published via the Interface Server.

<sup>2</sup> The minimal IDL document only contains a module element.



**Figure 5:** There are three main information paths in the CORBA Subsystem. The dashed lines represent the path used in publishing the server interface. The solid lines represent the path used in servicing remote method calls. The dotted lines represent the flow of control information within the subsystem.

**5.2.2. Server Interface Updates.** We chose our update model to mirror the SOAP subsystem since the concerns discussed in Section 5.1.2 also apply to the CORBA subsystem. The IDL Publisher listens for changes being made on the CORBA Server instance to determine whether to update the CORBA-IDL definition (See Section 5.6.) As discussed in Section 5.7, outdated RMI calls also trigger updates to the CORBA-IDL document. Once the new CORBA-IDL Document is produced, it is simply forwarded to the Interface Server for publication.

The Dynamic Skeleton Interface (DSI) [3] technology allows applications to provide implementations of the operations on CORBA objects without static knowledge of the object’s interface. We use DSI to avoid reinitializing the Server ORB when the server methods or types change.

**5.2.3. Request/Response Handling.** The components used in making RMI calls also mirror the components used in the SOAP subsystem. In this case, the incoming calls are received by the Server ORB. Unlike in the SOAP

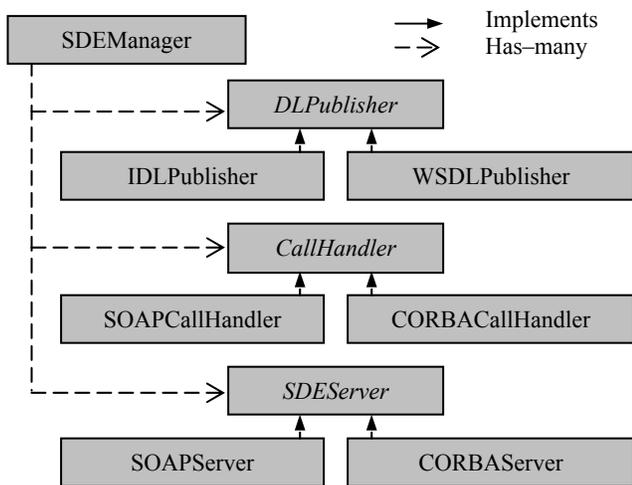
subsystem, the Server ORB implementation handles all malformed requests. The wrapper logic in the CORBA Call Handler component determines the validity of the call. If it is a valid call, a method call is made on the CORBA Server, and the return value is sent to the client through the Server ORB. If a call is not valid, a server interface update is triggered, if necessary (See Section 5.7), before a ‘Stale Method’ exception is sent back to the client. As in the SOAP subsystem, any exceptions thrown during the invocation of the method call is wrapped in a generic exception type and sent back to the client.

### 5.3. Representing Server Methods in JPie

As discussed in Section 4, when a user extends a class of type SDEServer, the list of possible modifiers for all methods defined in that class is augmented with the option of a ‘distributed’ modifier. Users add or remove methods from the published interface by selecting or deselecting this modifier within JPie. With this approach, we were able to develop SDE as an optional plug-in to JPie.

### 5.4 Class Hierarchy

To implement the components described in Sections 5.1 and 5.2, we designed a class hierarchy that allows multiple technologies to be easily integrated into the system. This allows key components such as the SDE Manager to be technology independent. Figure 6 shows three interfaces, each of which provide the blueprint to a component that performs a critical role within the SDE architecture.



**Figure 6:** Each technology incorporated into SDE must implement a generator to publish the server interface, a communication backend that handles incoming requests and sends reply messages, and an extensible class that will serve as the base type for dynamic classes using that technology.

### 5.5. Concurrency in Server Applications

In SDE, at most one instance of each dynamic class that extends SOAPServer or CORBAServer can exist at a time. Also, our Call Handlers are designed to be completely multithreaded. This allows the server to handle incoming calls efficiently and eases the performance bottleneck created by the mechanism described in Section 5.7 that attempts to maintain the consistency of the published server interface and the actual implementation in the server class.

### 5.6. Detection of Server Interface Changes

When a change is made to the server logic within the server dynamic class, those changes take immediate effect globally within JPie. When method signatures in the server application change, SDE needs to make the corresponding changes in the published server interface description to maintain consistency of the server interface on both the client and the server. On the other hand, since the generation and publication of the server interface description is a relatively expensive operation, eliminating unnecessary operations within the DL Publisher is important for overall system performance. One possible approach is change-driven: publish a new server interface description with each change. However, this approach would often lead to publishing transient server interface descriptions (those that occur while the developer is in the middle of editing the class), which is not only expensive at the server, but also may lead to unnecessary changes at the client. Another approach is to poll: check the interface at regular intervals, publishing if necessary. However, the periodic approach could still publish a transient interface. Moreover, that transient interface could persist at the client side until the next polling interval. Therefore, we have developed a mechanism that is change driven, but waits for a stable interval to avoid overly aggressive publishing. In addition, we incorporate a reactive mechanism that forces publication of the current interface whenever a client attempts to make a call on a stale method.

Our mechanism uses a timeout, which can be changed by the user through the SDE Manager GUI. Each DL Publisher listens to changes in the corresponding dynamic class by monitoring the JPie undo/redo stack. Each change to the server class restarts a timer. When the timer expires, the DL Publisher generates the new server interface. In addition, the user may manually prevent or trigger the publication of the server interface description at any time through the SDE Manager GUI. The control of the timer and the actual IDL generation is independent of each other, and there may be a running timer while an IDL generation is in progress. In that case, if the timer expires before the completion of the IDL generation, then another IDL generation will take place as soon as the current IDL

generation finishes. Client calls for stale method signatures may also trigger updates as described in Section 5.7.

This heuristic effectively publishes the server interface as needed while reducing the chance of publishing transient interfaces.

### 5.7. Client Invocations of Stale Methods

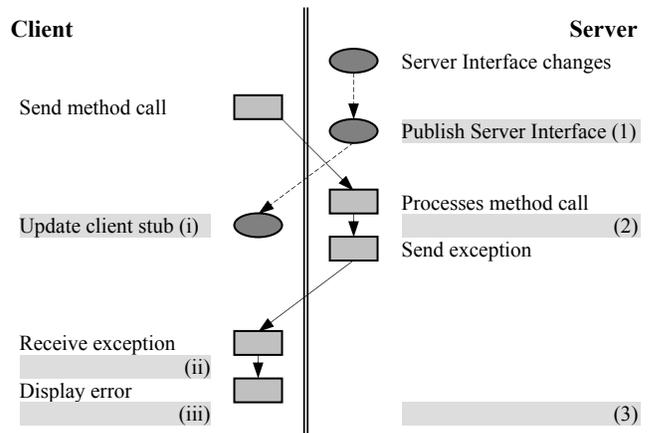
When a Call Handler receives a client request for a stale method, we must guarantee that the published server interface description is current before replying to the client with an exception. This is because if the client inspects the server interface description upon receiving the exception, the change in the method signature must be apparent. This mechanism enhances the server interface publication frequency by taking the frequency of client calls into consideration.

When a Call Handler receives a call to a stale method, it notifies the SDE Manager and delays the processing of incoming messages. The SDE Manager then prompts the corresponding DL Publisher to publish a new server interface description as needed. If the timer is not running and if there is no ongoing IDL generation, then we are guaranteed that the published server interface description is already current. If the timer is not running and there is an IDL generation in progress, then we are guaranteed that at the end of that publication operation, we will have the most current server interface description. In this case, we simply wait until the end of the operation before the SDE Manager is notified. If there is an ongoing IDL generation and the timer is running, we must wait until the current and the next IDL generations are completed to guarantee that the most current server interface description is published. The DL Publisher notifies the SDE Manager of the completion of the operation. The SDE Manager passes the notification back to the Call Handler. The Call Handler then sends an exception with the 'Stale Method' message to the client and resumes processing incoming messages. In CDE, this message is handled as described in Section 6.

Since publication is triggered only when the published interface is out of date, a rouge client cannot overwhelm the server by sending multiple calls to stale methods to trigger needless IDL generations.

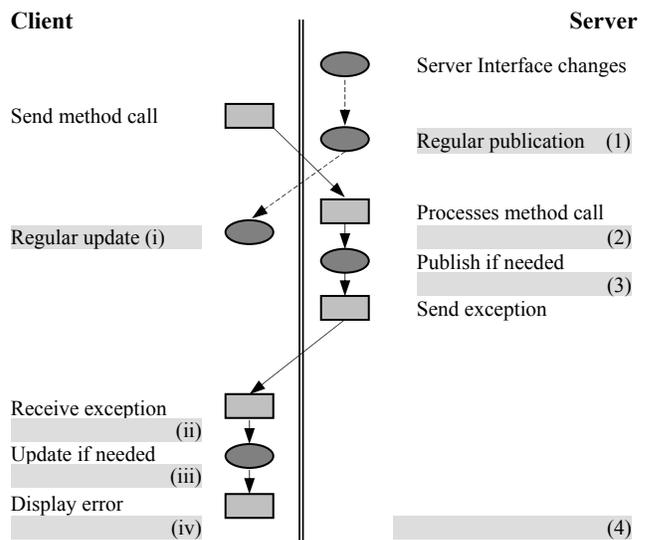
## 6. Live Client-Server Development

In live, simultaneous client-server development, both the RMI call path and the server interface update path may be active concurrently. Therefore, when the server interface changes, a race condition may arise between the two paths leading to inconsistent behavior in the CDE (e.g. the server reports that a method is stale, but the client has not yet the updated interface).

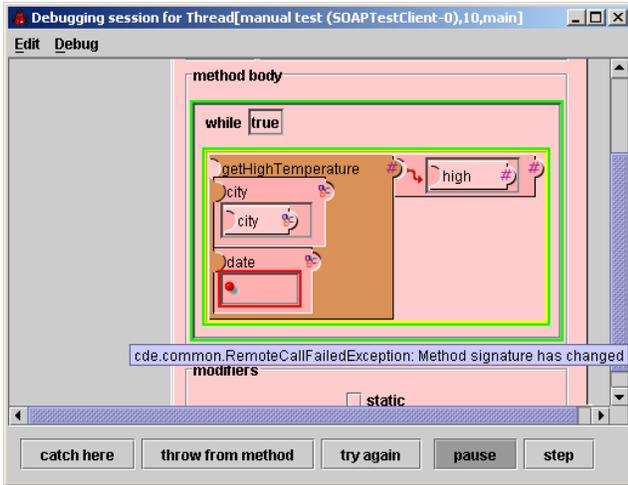


**Figure 7: Active publishing** - The server interface update path and the RMI call path are completely independent of each other. Only cases (1, i), (1, ii), (2, ii) produce the desired behavior of making the error obvious when the exception is reported back to the client developer.

The gray bars in Figure 7 illustrate the possible points at which the server interface is published (1, 2 or 3) and the client stub is updated (i, ii, iii). Only combinations (1, i), (1, ii), and (2, ii) ensure that the client developer is clearly able to see changes in the server interface when they are prompted by a server exception. In all other cases, the lack of a visible error in the client code will make resolution impossible until the client performs an update.



**Figure 8: Reactive publishing** - The server interface update path and the RMI call path have points of synchronization at both the client and server sides. In this case, for any combinations of (1-4, i-iv) the recency guarantees will be met.



**Figure 9:** When the “Stale Method” exception is received by the client dynamic class, the JPie debugger detects the exception and prompts the user. The goal of SDE and CDE is to make the error apparent to the client programmer.

To overcome the inconsistency shown in Figure 7, we developed the mechanism shown in Figure 8 that is implemented jointly by CDE and SDE.

In CDE, when a “Stale Method” exception is received by the client backend, the client view of the server interface is updated to the currently published one. The exception is then sent to the dynamic class that made the original RMI call. The JPie Debugger [12] detects the exception and displays it to the user as shown in Figure 9. When the user inspects the error, the server interface change is clearly visible. At this point, the user may make the necessary changes and resume normal execution.

After the server sends a “Stale Method” exception back to the client, there is a possibility that the server method signature is subsequently changed (during forced publication) to match the original call. In this case, we do not want the user to see the exception since the method call is actually consistent with the server interface. Therefore, when CDE receives a “Stale Method” exception from the server, it first compares the relevant method signature in the latest server interface description with the signature of the RMI call that triggered the update. If the signatures are identical, the call is silently re-invoked. Otherwise, the exception is propagated to the JPie debugger.

## 7. Performance

SDE adds some overhead to the RMI call structure, so an increase in the round trip time (RTT) of a RMI call is inevitable. Experimentation has shown that this overhead is within 15% (shown in Table 1) in comparison to static RMI servers, which is reasonable for development work.

**Table 1:** RTT times for client-server communication

Number of Parameters	Average RTT (seconds)		
	1	5	10
SDE SOAP/Axis	0.45	0.46	0.46
Axis-Tomcat/Axis	0.40	0.40	0.41
SDE CORBA/OpenORB	0.35	0.35	0.36
OpenORB/OpenORB	0.31	0.31	0.31

To determine the performance of SDE, we measured the average round trip time (RTT) of SOAP calls between a SDE SOAP server running within JPie and a simple static Axis client. We compared these figures with the RTT between the same Axis client and a static Axis server running within Apache Tomcat. We repeated the experiment using a SDE CORBA server, a static OpenORB server and a static OpenORB client. We used Java’s *getTimeInMillis* system call, and the average time was calculated over a thousand calls. We used an Apple Powerbook running OS 10.3 with a 1 GHz PowerPC processor and 512 MB of RAM as the client and a Dell Optiplex running Windows XP Professional with a 3.2 GHz Intel Pentium 4 processor with 1 GB of RAM as the server. The two machines were connected to the same T1 Local Area Network. All RMI calls returned a string value, and the parameters were also strings.

At the end of the development phase, the dynamic SDE server can be converted into a static SOAP or CORBA server through JPie’s built-in application export mechanism [20].

## 8. Conclusion

This paper introduced live server development using the RMI model as well as live, simultaneous client-server development. We also presented mechanisms that abstract away server deployment details, allowing SOAP and CORBA-RMI server development to become a natural extension of ordinary Java application development.

One of our goals for SDE was to reduce the learning curve involved in developing distributed applications using the RMI model. By eliminating the setup and deployment steps, we provided an environment where developers can devote their complete attention to the creation of server logic. SDE extends JPie to provide an appealing interactive environment in which novice RMI application developers can create and modify clients and servers.

Our second goal of supporting live client-server development has also been successfully implemented with the combination of CDE and SDE. Our experience indicates a significant reduction in development time from the traditional modes of distributed application development. We plan to use CDE-SDE as the basis for a client-server project in Washington University CSE 123, a course that uses JPie to provide a hands-on introduction to

computer science for non-majors without programming background [21].

An additional feature that is being investigated is the ability to switch, at runtime, the technology being used to communicate between the client and the server in a deployed system. Although some SOAP to CORBA bridging technologies [22, 23] offer static 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 and SDE. Our experience with that application will help motivate future work on CDE, SDE, and JPie in general.

## Acknowledgements

We thank Chris Gill and Michael Plezbert for support during our background study. We thank Vanessa Clark for her contribution in the design and implementation of CDE. This work was supported in part by National Science Foundation grant 0305954.

## References

- [1] *Simple Object Access Protocol (SOAP) 1.1*, World Wide Web Consortium, June 2003. <http://www.w3c.org/TR/SOAP>
- [2] *Common Object Request Broker Architecture (CORBA): Core Specification 3.0.3*, Object Management Group, March 2004. <http://www.omg.org/docs/formal/04-03-01.pdf>
- [3] K. J. Goldman, "Live Software Development with Dynamic Classes," Washington University, Department of Computer Science and Engineering, Technical Report TR-2004-81, August 2004
- [4] S. L. Pallemulle, V. H. Clark, and K. J. Goldman, "Supporting Live Development of SOAP and CORBA clients," Department of Computer Science and Engineering, Washington University in St. Louis, Tech. Rep. TR-2004-56, September 2004.
- [5] *Apache Axis Users Guide*, Apache Software Foundation, 2004. <http://ws.apache.org/axis/java/user-guide.html>
- [6] C. Wood, J. Daniel, and M. Rumpf, *The Community OpenORB Manual*, The Community OpenORB Project, 2004. <http://openorb.sourceforge.net/docs/1.4.0/OpenORB/doc/orb.html>
- [7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, *Extensible Markup Language (XML)*, 1st ed., World Wide Web Consortium, October 2000. <http://www.w3c.org/TR/RECxml>
- [8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL)*, 1st ed., World Wide Web Consortium, March 2001. <http://www.w3c.org/TR/wsdl>
- [9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol (HTTP)/1.1," June 1999.
- [10] *IDL to Java Language Mapping Specification*, 1st ed., Object Management Group, August 2002. <http://www.omg.org/docs/formal/02-08-05.pdf>
- [11] *Java to IDL Language Mapping Specification*, 1st ed., Object Management Group, September 2003. <http://www.omg.org/docs/formal/03-09-04.pdf>
- [12] K. J. Goldman, "An Interactive Environment for Beginning Java Programmers," *Science of Computer Programming*, vol. 53, no. 1, pp. 3–24, October 2004.
- [13] *Using Visual Studio .NET*, Microsoft Corporation, 2004. <http://msdn.microsoft.com/vstudio>
- [14] *Microsoft .NET Framework*, Microsoft Corporation, 2004. <http://msdn.microsoft.com/netframework/technologyinfo/>
- [15] *Microsoft ASP.NET Overview*, Microsoft Corporation, 2004. <http://msdn.microsoft.com/asp.net/technologyinfo/>
- [16] *The Apache Jakarta Tomcat 5.5 Servlet/JSP Container*, Apache Software Foundation, 2004. <http://jakarta.apache.org/tomcat/tomcat-5.5-doc/index.html>
- [17] *WebObjects Overview*, Apple Computer Inc., 2004. <http://developer.apple.com/documentation/WebObjects/>
- [18] *Developing Direct to Web Services Applications*, Apple Computer Inc., November 2002.
- [19] *BEA Tuxedo Product Overview*, BEA Systems Inc., 2001. <http://edocs.bea.com/tuxedo/tux80/overview/index.htm>
- [20] B. H. Brinckerhoff, K. J. Goldman, "Learning Curve Management in Educational Programming Environments," September 2004, submitted for publication
- [21] K. J. Goldman, "Washington University CS123: Introduction to Software Concepts," December 2003. <http://www.cse.wustl.edu/~kjj/cs123>
- [22] *Orbix 6.1 Technical Overview*, IONA Technologies, December 2003, <http://www.iona.com/whitepapers/Orbix6.1TechOverview.pdf>
- [23] *Artix Technical Brief*, IONA Technologies, April 2004, <http://www.iona.com/whitepapers/0404ArtixTechBrief.pdf>