RC23861 (W0602-026) February 1, 2006
Computer Science

IBM Research Report

Deployment-time Binding Selection to Improve the Performance of Distributed Applications

Sangjeong Lee, Kang-Won Lee, Kyung Dong Ryu,
Jong-Deok Choi, Dinesh Verma
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Deployment-Time Binding Selection to Improve the Performance of Distributed Applications

Sangjeong Lee Kang-Won Lee* Kyung Dong Ryu Jong-Deok Choi Dinesh Verma
{leesang, kangwon, kryu, jdchoi, dverma}@us.ibm.com
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

Abstract

We investigate a novel technology to improve the performance of a distributed application by configuring its components in an optimal way using the information about the operation environment discovered during the deployment of the application. To enable such capabilities we have developed a general architecture for deployment time optimization, called *Blue Pencil*. In this paper, we present how Blue Pencil can improve the performance of distributed applications by selecting the most appropriate binding between a client and a server using only static information such as the service interface definition, the location of the service, and information about the application server. Based on the performance study using real world Web services and internal benchmarks, we report that: (1) modern SOAP implementations are highly efficient and can perform even better than RMI under certain workloads and operation environment, and (2) the proposed binding selection algorithm is effective, and can improve the response time performance by 32% – 80 % on average even for highly optimized SOAP implementations.

Keywords: Deployment time optimization, Web service performance, SOAP, RMI, J2EE

1. Introduction

In enterprise computing, distributed applications require interoperation among multiple components that may be based on different platforms. For seamless integration of heterogeneous modules, the distributed computing community has developed layers of middleware abstractions. They include standardized network socket interfaces, remote procedure call (RPC) and remote method invocation (RMI) models, common object request broker architecture (CORBA), and more recently service-oriented architecture (SOA) enabled by Web services. In particular, CORBA and Web services aim to provide communication among distributed objects regardless of their platform and language differences. Such middleware layers provide good abstractions to software developers so that they can focus on core algorithms and minimize the effort in communication and data exchange. However, naïve adoption of such technologies can adversely impact the performance of distributed applications.

In particular, communicating over SOAP (Simple Object Access Protocol) has been reported to be slower than that over RMI by orders of magnitude [1][2][3][4]. However, programmers may still choose to write an application on top of SOAP abstraction even when lighter bindings are sufficient for several reasons. First, SOAP is general and widely applicable since it is platform and language independent. Moreover, since it uses HTTP as its transport protocol, it can travel through firewalls. In contrast, Java RMI based on JRMP (Java Remote Method Protocol) is language-specific since both client and service modules should speak Java. Although RMI over IIOP (Internet Inter-ORB Protocol) is a viable alternative to SOAP-based Web service to achieve language and platform independence, it is often considered as expensive as Web services. Finally, when the application is developed, the programmers usually do not know how the application will be used and where it will be deployed. Thus they may decide to be on the safe side and assume SOAP binding, which is more general.

* Corresponding author: Kang-Won Lee; Email: kangwon@us.ibm.com; Phone: +1-914-784-7228

In [5], we have proposed a generic architecture for deployment time optimization called *Blue Pencil*. In a departure from conventional techniques employed at the development time or the run time of an application, Blue Pencil tries to optimize the composition of a distributed application and the bindings between its constituents based on the configuration of a target environment where the application will be executed. This paper investigates the effectiveness of deployment time optimization in the context of J2EE distributed applications.

In particular, the objective of this paper is to propose a novel framework to enable optimally deploying applications by selecting the best bindings. To this aim, we conduct detailed performance study of SOAP and RMI using various Web service applications. We then propose a binding selection algorithm based on the insights obtained from the performance analysis. We also present the structure of the Blue Pencil components relevant to binding selection and configuration discovery. More specifically, the main contributions of this paper are as follows:

1. *Performance analysis of SOAP and RMI in the latest J2EE implementations*: We examine the performance of several Web services in both open source (JBoss Application Server) and commercial J2EE implementations (WebSphere® Application Server).
2. *Design of a novel binding selection algorithm for deployment time optimization*: We present the design of our binding selection algorithm based on the performance analysis of SOAP and RMI and present how it is used in Blue Pencil.
3. *Performance evaluation of the proposed algorithm*: We validate the efficacy of the proposed algorithm by quantifying the benefits using several public Web services and internal Web service benchmarks.

While binding selection is only one of many aspects of deployment time optimization, we demonstrate that it can still provide significant performance benefits by itself. From our performance evaluation, we observe that our algorithm can improve the performance of a Web service application by 32% – 80% in response time even in a J2EE environment with a highly optimized SOAP processing layer.

The remainder of the paper is organized as follows: Section 2 provides a background on deployment time optimization, and RMI and SOAP. Section 3 presents preliminary performance results to motivate the rest of the paper. Section 4 presents a simple model to estimate the relative performance of RMI and SOAP. Section 5 presents the modules for deployment-time binding selection in our Blue Pencil framework. Section 6 presents performance numbers from several real world Web services, benchmark, and a tiered Web service scenario to show the efficacy of the proposed scheme. Section 7 summarizes the related work in this area, and finally Section 8 concludes the paper with future research directions.

2. Background

In this section, we present a brief overview of the deployment time optimization approach. We then provide a summary of SOAP and RMI protocols highlighting the features that are relevant to our discussion.

2.1 Overview of Deployment Time Optimization

A distributed application comprises multiple components. For example, a single application may consist of several modules that span over a client machine, an application server, and a database system. When constructing a distributed application there can be several alternatives that provide a similar functionality. One example is remote object binding via either Web service, CORBA, or Java RMI, which is the

primary focus of this paper. Another example is different types of database drivers with different features and performance characteristics. For example, a type 2 JDBC driver may provide better performance than a type 1 JDBC driver, since they can exploit native interfaces; but a type 1 driver is more portable since it is all Java and is based on open technologies.

The basic idea of deployment-time optimization is to improve the performance of an application by selecting the most efficient components and configuring them in an optimal manner, exploiting the information about a target operation environment that is available at deployment time. By components, we mean libraries, database drivers, network protocols, middleware components, and other modules that are needed for applications to run.

While the concept of deployment time optimization is intuitive and simple, the potential benefit of this new approach can be great. In most application scenarios that we have examined, however, enabling deployment time optimization is not straightforward. First, it is fundamentally difficult to predict the impact of selecting one configuration over another without actually trying them. Second, the information about the target environment may not be readily available. Third, the granularity of the application may be too coarse so there are only limited ways to combine them, and thus it may be difficult to attain significant performance gain. To address these issues, we have developed the Blue Pencil architecture. We present the design of Blue Pencil and its modules that are related to the binding selection process in Section 5.

2.2. Overview of RMI and SOAP

Remote Method Invocation (RMI) is a Java API to enable invoking methods of remote objects. There are two flavors of RMI: Java RMI based on JRMP (Java Remote Method Protocol) and RMI-IIOP (Internet Inter-ORB Protocol) used in CORBA.[1] RMI-IIOP allows Java application can talk to a remote object implemented in other languages such as C++.

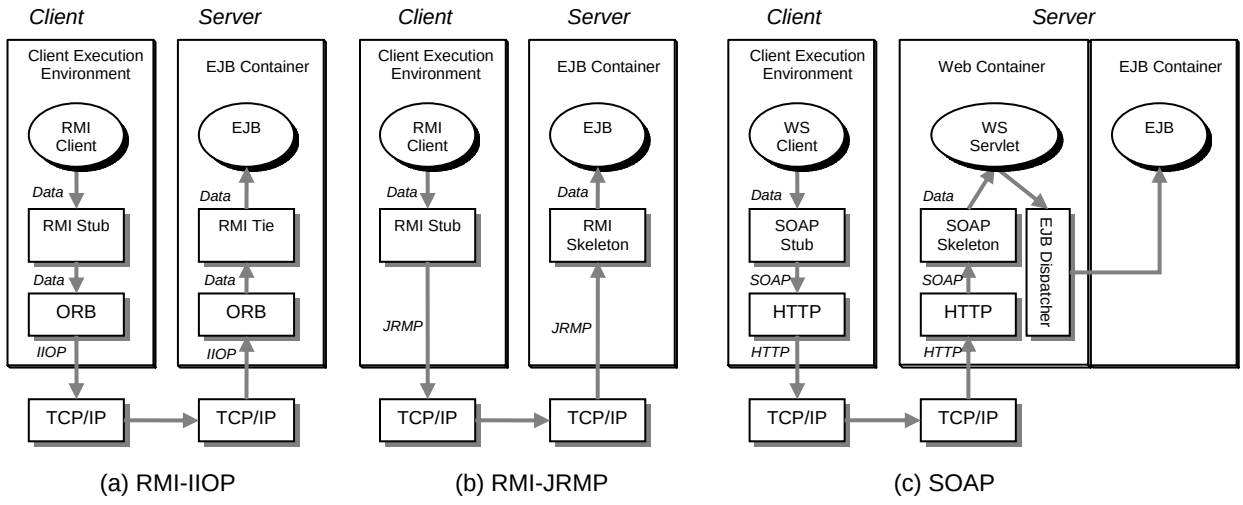


Figure 1. Process Diagram for Request Path of RMI-JRMP, RMI-IIOP, and SOAP

[1] In general, a transport protocol in CORBA is called GIOP (General Inter-ORB Protocol). IIOP is a special instance of GIOP, where the underlying network protocol is TCP/IP.

Figure 1 (a) illustrates the processing on the request path of RMI-IIOP method invocation of Enterprise Java Bean (EJB) in the J2EE framework. To invoke a method of a remote EJB, the client calls the corresponding method of its RMI stub. The stub then passes the invocation information to an ORB (object request broker). The ORB translates the information into an IIOP message. When the server-side ORB receives the IIOP message it processes the message and invokes a corresponding Java method of an RMI tie, which is a server-side skeleton. The RMI tie, in turn, invokes a corresponding method of the EJB and returns results. The response traverses the same path back to the client. In case of RMI-JRMP (Figure 1 (b)), the processing steps does not involve ORBs, and instead the messages are marshaled using Java serialization.

SOAP (Simple Object Access Protocol) is an XML-based protocol to allow applications to make remote method call over HTTP. Using XML to encode messages, SOAP provides a true language and platform independence. SOAP is the main communication vehicle for Web services. Figure 1 (c) illustrates the processing on the request path of SOAP. Compared to RMI, we notice one difference – the server has a Web container, which contains a special servlet. The servlet handles HTTP messages between clients and service EJBs. When the servlet receives an HTTP request from the client, it passes the message to the SOAP skeleton, which parses the message and creates corresponding Java objects. Then it delivers the objects to the Web service servlet, which in turn invokes the method of the back-end EJB. The response returns on the same path back to the client.

Although RMI and SOAP incur similar processing at high level, SOAP incurs more processing overhead since it needs to encode native objects to and from XML format. We study this issue more closely in the next section.

3. Performance of RMI and SOAP

Related works [1][2][3][4] have reported that RMI is up to a few orders of magnitude faster than SOAP in various applications. However, recent advances in SOAP/XML technologies have greatly improved SOAP's performance rendering such presumptions to be not always valid. In this section, we revisit the performance issues of SOAP and RMI using two popular J2EE application servers: JBoss and WebSphere.

3.1 Workload and configuration

For the first set of experiments, we have used the latest JBoss Application Server (version 4.0.3) developed by an open source community. For workload, we have selected two Web services with different characteristics: (1) Google Web service, which is a simple Web service API to invoke the Google's search function, and (2) WSBench, which is an IBM's internal Web Services performance benchmark that models after complicated financial transactions (e.g. money transfer). For testing, we setup two Pentium 4 machines running Windows XP as a client and a server, connected via fast Ethernet.

The Google Web service takes keywords and the number of search results to be returned, as input, and returns the search results including the URL and the snippets of the page. To measure the binding performance of Google Web service, we have created a local testbed consisting of a client and a server, and simulated their interactions. For this, we have collected search results for the most frequently used keywords at Google.com and populated the server with them. The actual Google service limits the number of search results by ten. However, in our experiment, we have varied this parameter from 10 to 100 to see the impact of message volume changes.

The structure of WSBench messages is more complex. More specifically, the request messages contain account information such as account number, amount of money to be transferred, currency information,

bank id, person's name. They also contain security-related parameters such as customer id, password, and encryption type. The response messages contain information about the result of money transfer and the server side information such as status description, user's name, and severity. They also contain information about the server, transaction date/time, and application information. To simulate different types of financial operations, WSBench provides parameters to change the size of request and response messages. We have varied the size of each message from 1KB to 100KB.

3.2 Preliminary performance

	Parameters	Response Time (msec)			Throughput (reqs/sec)		
		SOAP	RMI (IIOP)	SOAP/RMI	SOAP	RMI (IIOP)	RMI/SOAP
Google	10	1232.4	122.6(48.0)	**10.0x(25.7x)**	1.62	561.8(515.5)	**346x(318x)**
	50	4946.8	486.3(241.9)	**10.2x(20.4x)**	0.40	216.9(213.7)	**542x(534x)**
	100	9322.7	895.2(454.1)	**10.4x(20.5x)**	0.21	146.0(116.6)	**680x(555x)**
WSBench	10KB	5801.6	286.3	**20.3x**	0.34	137.0	**397x**
	50KB	27735.4	2539.3	**10.9x**	0.07	38.4	**549x**
	100KB	62953.8	7697.7	**8.2x**	0.03	16.0	**503x**

Table 1. SOAP and RMI Performance with JBoss

Table 1 summarizes the response time and the throughput performance of the two Web services with JBoss. For Google, we report the numbers with SOAP, RMI-JRMP, and RMI-IIOP (in parenthesis). For WSBench, however, we could not obtain the results for RMI-IIOP since the server returns an unknown error message.[2] Thus we only report the numbers for SOAP and RMI-JRMP in WSBench case. Also in JBoss, SOAP's performance is extremely slow; thus we could only run two threads on the client, whereas for RMI we ran 50 threads to fully exercise the server.

From the table, we find that RMI performs much better than SOAP regardless of underlying transport protocols. For Google search services, RMI-JRMP is up to 10 times faster in response time and up to 680 times higher in throughput than SOAP. Similarly, RMI-IIOP is up to 20 times faster in response time and up to 555 times higher in throughput compared to SOAP. For WSBench, RMI-JRMP is up to 20 times faster in response time and up to 514 times higher in throughput than SOAP. Overall, this experiment confirms the previous findings in the literature.

We now perform the same experiment in the latest WebSphere Application Server (WAS) environment (version 6.0.2). Before we start performance comparison, it is worth noting a few important differences between WebSphere and JBoss. First, WAS only supports IIOP for an RMI client to access an EJB. Second, WAS includes cutting-edge optimization technologies that have been developed to accelerate SOAP and XML processing; thus its SOAP performance will be much better than that of JBoss.[3] Finally, we ran 50 threads on the client for both SOAP and RMI experiments. Table 2 presents the performance number of the two Web services in the WebSphere environment.

[2] We believe it has to do with Java Calendar objects since we found that JBoss failed to handle them in other applications too.

[3] They include pre-compiled serializers and deserializers, schema-specific XML parsing techniques, and highly optimized SOAP processing layers. Detailed description about these technologies is outside the scope of this paper.

	Parameters	Response Time (msec)			Throughput (reqs/sec)		
		SOAP	RMI	SOAP/RMI	SOAP	RMI	RMI/SOAP
Google	10	227.6	84.7	**2.69x**	219.7	590.3	**2.69x**
	50	599.7	276.5	**2.17x**	83.4	180.8	**2.17x**
	100	1185.0	516.2	**2.30x**	42.2	96.9	**2.30x**
WSBench	10KB	313.5	452.2	**0.69x**	159.5	110.6	**0.69x**
	50KB	918.2	1768.1	**0.52x**	54.5	28.3	**0.52x**
	100KB	1734.1	3476.9	**0.50x**	28.8	14.4	**0.50x**

Table 2: SOAP and RMI Performance with WAS

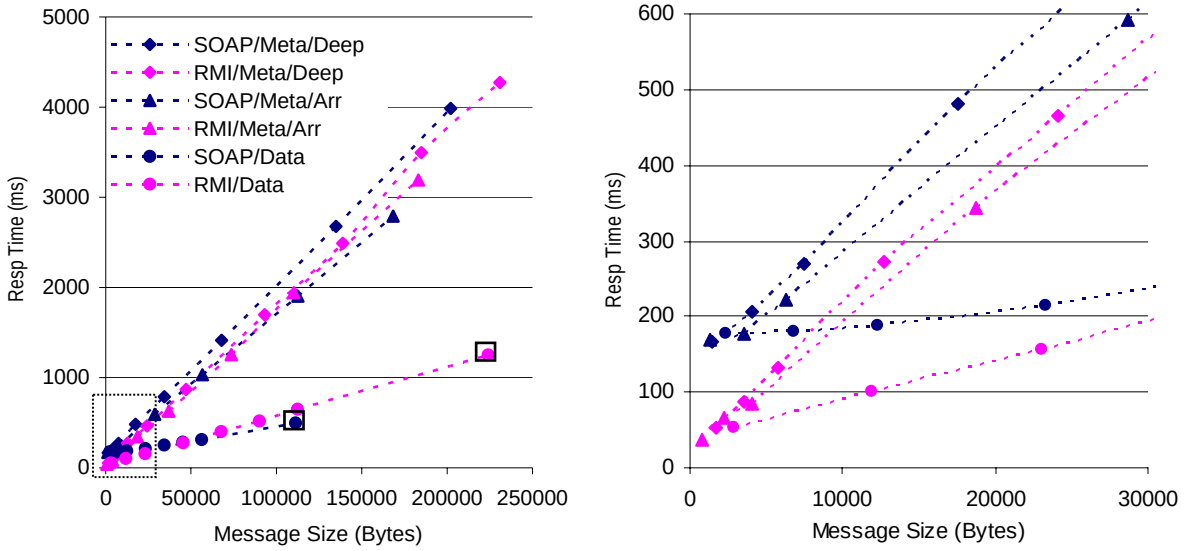
From the table, we observe that the performance of SOAP is significantly better than in the JBoss case despite the increase in the workload. For Google Web services, we find that RMI still performs 2.7 times better in terms of average response time and throughput performance. However, in the WSBench case the performance of SOAP is actually better in terms of response time and throughput.

The preliminary experiment results in this section provide important lessons. First, the performance characteristics of SOAP and RMI bindings are highly dependent on a particular application server environment. Therefore, it is important to configure applications using such information. Second, the conventional wisdom saying "use RMI bindings over SOAP whenever possible" is no longer valid in all cases. With the increasing emphasis on Web services technologies, we expect the performance of SOAP will continue to improve further. As a result, we conclude that optimization techniques that consider both configuration information and the characteristics of a workload will provide a great benefit.

3.3 Understanding the Performance Inversion

Previous section has shown that under certain circumstances SOAP can perform better than RMI. This is counterintuitive since SOAP is based on XML, which is very verbose, and translating native objects into and from XML messages is known to be slow. To understand this performance inversion, we analyze how data objects are represented in IIOP and XML messages, and find that the inversion results from the extensive use of Java String objects and Java Calendar objects. More specifically, IIOP employs UTF-16, which requires two bytes for each character, whereas SOAP uses UTF-8 encoding, which requires only one byte for each character. Furthermore, Calendar objects become extremely enlarged in RMI-IIOP – adding about 1000 bytes to describe the Calendar type, which consists of more than 100 different primitive fields and consuming about 500 bytes to describe their values. On the other hand, SOAP needs at most 24 bytes to represent Calendar in XML's dateTime format, e.g., "2005-09-09T18:18:30.830Z". In other words, the subpar performance of RMI-IIOP with WSBench results from its overblown message sizes.

To gain further insight, we look into the correlation between the message size and the response performance. For this, we have generated three different types of objects – varying parameters that affect the structure and size of messages. The first two types of objects contain only a small number of strings and integer types but have a very complex structure. Thus we call them *meta-data-oriented* (denoted by *Meta* in the graphs). The first type of meta-data-oriented objects is generated with a deep class nesting, up to 10 levels in this experiment (denoted by *Deep* in the graphs). The second type is generated by a large array (denoted by *Arr* in the graphs), whose size varies from 100 to 3,000. In addition, we consider objects that are largely populated by data. We call such objects *data-oriented* (denoted *Data* in the



(a) Overall graph for message size < 250KB (b) Magnified graph (dotted region in the left graph)

Figure 2. Relationship of Message Size and Response Time

graphs). In the experiment, we use an object with 20 integers and 10 strings which contain 100 to 10,000 characters each.

Figure 2 presents the correlation between the response time and the message size for different types of objects with SOAP and RMI. Each data points in the graphs are denoted by binding_type/object_type/meta_type. For this experiment, an RMI message and a SOAP message are created from the same Java object. The chart provides a few interesting insights. First, in most cases, the RMI message size and the SOAP message size for the same object are quite different. For instance, a large data-oriented message which is about 120KB in RMI-IIOP is about 230KB in SOAP (the right most points marked by squares in RMI/Data and SOAP/Data in Figure 2 (a)). Second, the initial response time, which is the y-intercept of each curve in Figure 2 (b), is much higher for SOAP (about 160 ms) than RMI (about 20 ms). We find this is due to the heavy initial cost of extra HTTP routing via a Web container to deliver requests to a J2EE container as presented Figure 1. Therefore, RMI offers a better response time in the range where the size is less than 10KB. Third, the slopes for RMI and SOAP plots are almost the same for the same type of objects (Figure 2 (a)). This observation draws an interesting conclusion: if we can estimate the message size of RMI and SOAP between the client and the server, we can effectively predict the response time performance of those protocols and infer which binding will perform better between SOAP and RMI.

4. Message Size Estimation Model

This section presents our model to estimate the average message size for a given service under SOAP and RMI-IIOP protocols. We also discuss some practical issues and limitations of the proposed algorithm.

We present a detail model for SOAP and RMI message size estimation in Appendix. For our discussion, however, we only need to know that both SOAP-HTTP and RMI-IIOP messages consist of three major parts: (1) constant part (e.g. HTTP header, envelope tag in SOAP, and IIOP header, message-specific header in RMI); (2) variable part that can be calculated from WSDL (e.g., namespace, payload tags in SOAP, and fixed primitive data types, fixed data types in a constructed data types in RMI); and (3)

variable part that cannot be calculated from WSDL (i.e., payload data). Thus, in order to estimate average message size, we need to have an idea about the actual data values in the message.

From the aforementioned observation, it is a challenge to reliably estimate which binding will incur larger messages size on average without the knowledge of actual data patterns. Another related problem is that some parameters in the message may actually be null. The ratio of null parameters differently affects the message size estimation for SOAP and RMI – the size of RMI messages will be significantly reduced by null objects since it only consumes 4 bytes per null object. On the other hand, SOAP still needs to insert meta-data information (e.g., tags) even for null values.

We can address this challenge in various different ways depending on the availability of parameter value hints. First, consider a scenario when the service side collects history about the requests that it has seen in the past and provide this information to a message size estimation module. If the history provides enough details on the key parameters, our proposed algorithm will accurately predict the message size. However, this approach will incur too much overhead on the server; thus we do not consider this option. Second, instead of service-specific history, we use general hint for each service type. For example, when we deploy a client for bank application, we can use general parameter space that has been derived from some other bank services. The accuracy of this approach may not be as good as the first approach; however, it does not require service-side instrumentation. Finally, we can estimate the size without any hint and use only type definitions in WSDL and a value range that is large enough to cover most cases. For example, the length of a number parameter is clearly bounded by their definitions. The null probability of parameters is defined in 0 – 100 % range. For other types of variables, we can define the ranges from empirical values. Another factor that bounds the parameter value space is the resulting message size estimate. If we place a maximum bound on the message size itself, it will conversely place bounds on each parameter values. We explore the second and the third options in this paper.

5. Implementation

In [5], we have presented the Blue Pencil architecture, which has been designed to enable deployment time optimization. In this section, we present the components of Blue Pencil, and how they facilitate binding selection.

5.1 Smart Stub

To allow a client application to select a binding depending on the configuration, the service implements multiple bindings, e.g., SOAP, remote RMI, and local RMI. The service then publishes this information in a WSDL file. [4] When developing a client, the software developer refers to the interface definition in the WSDL file and design the client based on that interfaces.

Blue Pencil provides a programming environment for creating applications that are componentized and flexible. For this problem, we need a module to generate a client side proxy that encapsulates the functionality of both SOAP and RMI stubs. For this, we have developed a plug-in, called the *smart stub generator*, which automatically generates a client side proxy called a *smart stub*. This functionality can be realized in an integrated development environment, or as command line tools similar to *rmic* (RMI compiler).

[4] We can use an extension to WSDL, such as WSIF (Web Service Invocation Framework) extension [14], to allow specifying multiple bindings for a single service.

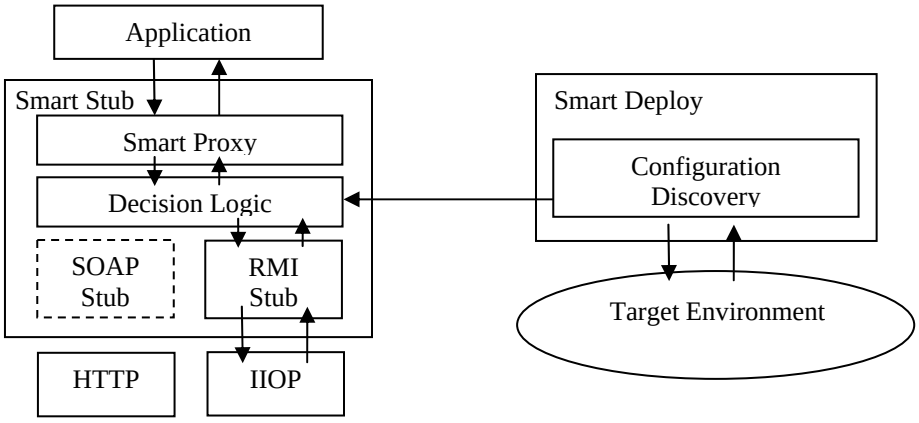


Figure 3. Structure of a Smart Stub

Figure 3 presents the structure of a smart stub, and its interaction with the application and the configuration discovery module. To the application, it exposes simple interfaces through the smart proxy. When the application is deployed by an installation module, called the *smart deploy*, the installation module invokes a configuration discovery module inside it to discover the configuration information. Then it hands the information to the decision logic in the smart stub. Based on this input, the smart stub decides which binding to instantiate. The example in the figure shows the case when an RMI binding has been chosen – in this case, only an RMI stub will be created. Thus the processing overhead on the critical path is the same as the RMI binding except the thin layer of smart proxy.

```
import RemoteObjectFoo;

// remote object creation
RemoteObjectFoo foo = new RemoteObjectFooProxy();

// method call
foo.bar();
```

Figure 4. Accessing Remote Object Using Smart Stub

Writing applications using the smart stub interface is similar to the case of writing a standard RMI or SOAP application. Figure 4 illustrates an example of creating and accessing a remote object via smart stub. In this example, `RemoteObjectFooProxy` is a proxy that has been auto-generated by the smart stub generator using the WSDL file of a RemoteObjectFoo service. Creating a handle for that remote object is as simple as just calling the constructor of the proxy. Once the handle is created, then it can be used in the same manner as a local instance.

5.2 Configuration Discovery

In our scenario, the configuration discovery module must first identify the types of binding that the service supports. Then it needs to determine the characteristics of the operation environment, e.g., the type and the version of application server. After that it needs to determine the relative location of the service and the client – for example, whether they share the same ORB, on the same subnet, or connected via wide area network. Based on this information, the decision logic in the smart stub will select an appropriate binding. This selection policy may be represented using a series of *if-condition-then-action* rules. Figure 5 presents a sample C-style pseudo code of the binding selection procedure.

```
BindingSelection (serverInfo, locationInfo, existsFirewall) {
  if (locationInfo == same_classloader) then
        use EJB_local_interface;
   if (locationInfo == same_ORB) then
         use EJB_remote_interface_with_local_RMI;
  if (existsFirewall == false) then
        if (serverInfo == JBoss)
             use EJB_remote_interface_with_remote_RMI;
        if (serverInfo == WebSphere)
             binding = selectBySizeEstimate();
             use binding;
        // code for other server types
  }
  else use SOAP_binding;
}
```

Figure 5. Pseudo Code for Binding Selection Algorithm

If the client and the server are deployed to run in the same class loader, they can communicate via EJB local interface, which is equivalent to making a native Java call. On the other hand, if both client and server share the same ORB instance, they can communicate via local RMI (with remote EJB). In this case, we can bypass the IIOP translation steps and therefore, this invocation is more efficient than calling via remote RMI. We can check if a client and a server share the same ORB instance by calling `javax.rmi.CORBA.Util.isLocal` method. If they are on different machines, but there is no firewall in between, they can potentially use RMI-IIOP using EJB remote interface with remote RMI. If the application server in the target environment is JBoss, we know it is always better to use RMI. However, if the application server is WAS then we need to run the size estimation algorithm to determine which binding will perform better, and then use the binding that is predicted to be more efficient. Finally, if there is a firewall between the client and the service, they must communicate through SOAP over HTTP.

6. Performance Evaluation

For evaluation, we have set up a testbed and instantiated various Web services. We first describe the workloads that we have used. We then evaluate the effectiveness of our binding selection algorithm, and present the performance improvement by employing the proposed mechanism.

6.1 Workload Characterization

We have selected various Web services with different characteristics; they include Amazon Web Service, Google Web APIs, Microsoft MapPoint Web Service, and IBM's Web service benchmark, called WSBench. We have also created a multi-tier Web service emulating the behavior a travel reservation system. We have explained Google Web APIs and WSBench in Section 3.1. To summarize, the Google service provides keyword-based search, and thus its messages are heavily string-based and relatively small. WSBench is an internal benchmark that models after the financial transfer operations between banks. Thus its message structure is complex (involving arrays, mixed data types) and the message size can grow large (as large as a few hundred KBs). We believe this workload is ideal to represent batch data transfer or complex data transfer.

Amazon Web Service provides operation to inquire products, customers, and sellers. Their primary operations are search (by keywords) and lookup (by unique IDs). While the semantics of the two operations are different, their message formats are almost the same. Thus in this paper, we present the results with the search service only. The Microsoft MapPoint Web Service provides geographical services such as find address, calculate driving directions, and retrieve maps. The first operation consists of simple string-based messages. However, calculating route and map operations involve more heterogeneous data types including binary data type.

Service Name (*operation*)	Service Type	Message Dir	Message Size	Parameter Characteristics					
				Depth	Strlen	Num	Binary	Null%	Array
Google *doGoogleSearch*	Web search	Request	Small (1KB)	1	22	1.5	-	45%	0
		Response	Medium (10KB)	5	76	4.6	-	40%	6.8
Amazon *itemSearch*	Online shop	Request	Small (2KB)	3	10	-	-	87%	1
		Response	Medium to Large (15 – 116KB)	10	27	-	-	84%	7.1
MapPoint *findAddress*	Map	Request	Small (1KB)	2	12	-	-	33%	0
		Response	Small (2KB)	9	6	13.7	-	25%	1
MapPoint *calculateRoute*	Map	Request	Small (3KB)	10	11	17	-	36%	1.5
		Response	Large (21KB)	12	27	15	1618	23%	3.5
MapPoint *getMap*	Map	Request	Small (4.5KB)	8	9	14.6	1618	47%	2
		Response	Large (58KB)	7	9	17.5	42575	12%	1
WSBench *handleAddShort1*	Finance	Request	Small (1.6KB)	5	8	1	-	5%	0
		Response	Small (1.4KB)	5	11	1	-	0%	1
WSBench *handAddShort10*	Finance	Request	Medium (10KB)	7	11	4.3	-	0%	3.6
		Response	Medium (10KB)	5	19	1.3	-	0%	25
WSBench *handleAddShort100*	Finance	Request	Large (100KB)	7	11	4.9	-	0%	4
		Response	Large (100KB)	5	19	1.6	-	0%	264

Table 3. Summary of Workloads

Table 3 presents a summary of the workload's characteristics and the key parameters that affect the message size and consequently the performance of service invocation. In the table, we show the average SOAP message size, and average values for string length, number of digits, and binary object size in the messages. We also present null probability, average array size, and depth of nesting. The null probability indicates how likely a field may not have a value in messages. This parameter is relevant because certain Web services, e.g., Amazon, implement general APIs for multiple purposes; and when it invokes a particular service, it leaves all other fields null. The depth indicates the level of nesting when an object has other objects as its fields, and those objects again contain other objects. In general, the message structure naturally determines the nesting depth. The array size is an average of all array sizes in a message. We see, except WSBench, the average array size is relatively small in our workload. The parameter values that we observe in these workloads may provide guidance when we run the size estimation algorithm.

6.2 Evaluation of the Size Estimation Algorithm

Now, we evaluate the performance of the message size estimation algorithm presented in Section 4. As we discussed earlier, we explore two different approaches for message size estimation. The first approach does not rely on any hint except that each parameter type will have some reasonable value range and message size has a maximum bound. From our experience with different types of Web services, we set the maximum message size to 300 KB.[5] The other parameters that we use are as follows:

Parameter	String length	Binary data size	Null %	Array size
Value	5, 15, ... , 75	1.5KB, 4.5 KB, 15KB, 45KB	0% – 90 %	2, 5, 10

[5] Although some scientific Web service applications exchange large size messages in a Grid environment, such messages usually consist of arrays of numbers, which can be efficiently handled with RMI all the time. Thus we do not consider such workloads in this paper.

The second approach utilizes simple hints about a service. A hint can be a characteristic that can be derived from certain types of services such as maximum message size, average string length, or null probability. For example, a batch transaction service typically incurs large messages. A search service usually returns relatively long strings as results. Generic service APIs may incur messages with high null probability. In this paper, we investigate two different levels of hints to see how they affect the performance of our algorithm. In the first level, we use the maximum message size as a hint – where small messages are less than 10KB, medium messages are less than 50KB, and large messages are less than 300KB. In the second level, we use a tighter range of string size and null probability (± 10% of the actual) along with the max message size bounds, instead of using the default ranges.

Service	Accuracy (Frequency)			Decision
	No hint	Hint: A	Hint: A, B, C	
Google - *doGoogleSearch*	91.7 % (4400)	91.7 % (4400)	88.9 % (960)	RMI
Amazon - *itemSearch*	62.5 % (1000)	62.5 % (1000)	100 % (240)	RMI
MapPoint - *findAddress*	84.2 % (4040)	100 % (1644)	100 % (180)	RMI
MapPoint - *calculateRoute*	81.6 % (3781)	81.6 % (3781)	100 % (720)	RMI
MapPoint - *getMap*	95.2 % (3420)	95.2 % (3420)	100 % (450)	RMI
WSBench - *handleAddShort1*	97.1 % (4660)	97.1 % (4660)	100 % (360)	RMI
WSBench - *handleAddShort10*	77.1 % (3700)	77.1 % (3700)	100 % (360)	SOAP
WSBench - *handleAddShort100*	55.2 % (2220)	55.2 % (2220)	100 % (360)	SOAP

Table 4. Accuracy of Message Size Estimation (hint A: max message size, B: strlen, C: null%)

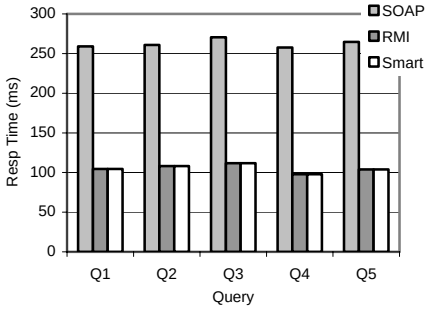
Table 4 presents the accuracy of the message size estimation algorithm with or without hints. The numbers in parenthesis represent the number of instances when the correct binding has been selected over the entire parameter space. The numbers in the "No hint" column present the case when the estimation was made without any hint specific to the Web services. The result shows that the message size estimation could select the right binding in all cases even without a hint. However, the accuracy for Amazon and WSBench (handleAddShort100) are relatively low. For Amazon, the workload exhibits very high null probability (> 84%) as shown in Table 3. Thus it becomes difficult to estimate the performance without such hints. In the WSBench case, the actual operation involves only large messages; however, the parameter space ranges from very small to very large messages.

When we have a simple hint about the max message size ("Hint: A" column), we only see improvement in MapPoint (findAddress), and the accuracy of all the other services remains the same. Thus we can conclude that having a hint on max message size is not very effective. Finally, when we have hints about average string length and null probability as well as message size ("Hint: A, B, C" column), we see the message size estimation algorithm selects the correct binding with a very high accuracy. In this case, because the regions of these parameters are more specific, the number of data points is less than the previous cases; however, they still provide enough reference points for decision. The only exception is Google, whose accuracy has decreased by a small margin. We find this is due to the high concentration of string data in its messages, which favors SOAP. Nevertheless, we could select the right binding with a high confidence in this case also.

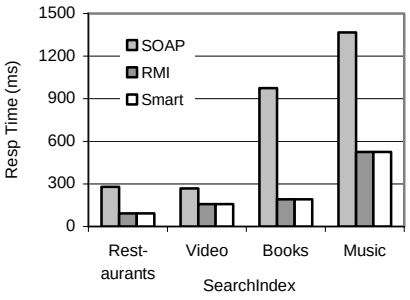
Overall, the proposed estimation algorithm proves effective even when we do not have much information about specific services. In particular, the proposed message size estimation algorithm selected the right binding in all cases that we have considered.

6.3 Performance Benefits of Smart Stub

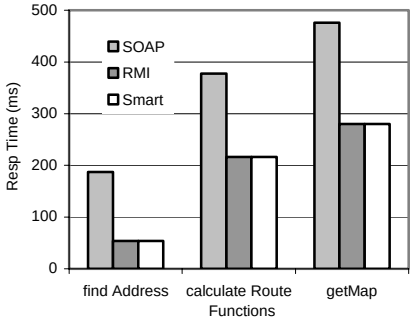
We evaluate the performance improvement of our smart stub in the local testbed. We used a Pentium 4, 2.0 GHz machine with 1.5 GB memory, as a server, and used a Pentium 4, 3.0 GHz machine with 3.0 GB memory as a client both running Windows XP. In all our experiment, we assume a configuration where both SOAP and RMI can be used, e.g., client and server are on the same network. On the client side we have created 50 threads, each of which simulates the behavior of an individual client, to keep the server busy. Figure 6 shows the average response time results with smart stub along with SOAP and RMI bindings.



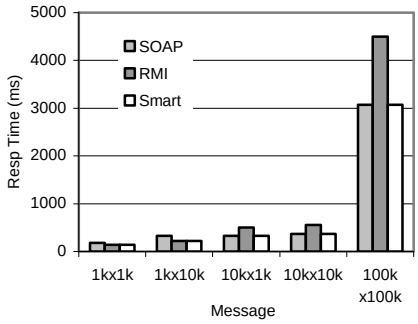
(a) Google - doGoogleSearch



(b) Amazon – itemSearch



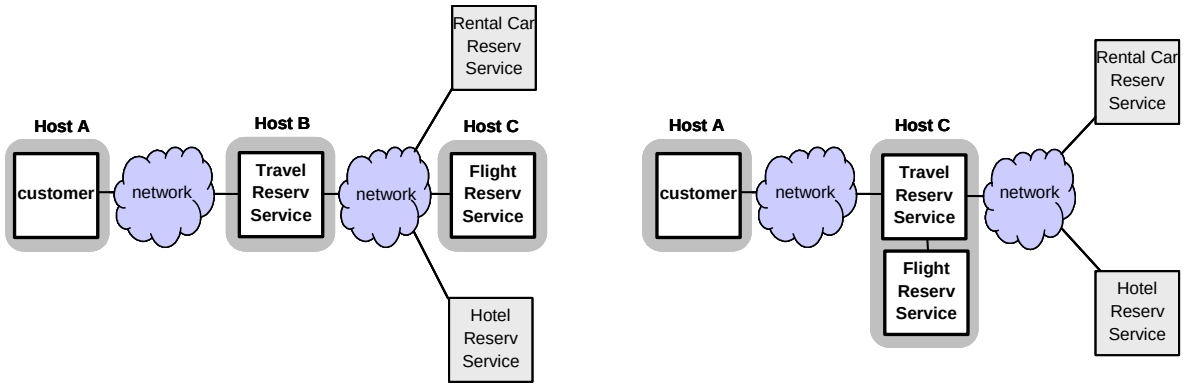
(c) MapPoint



(d) WSBench - handleAddShort

Figure 6. Average Response Time with Different Bindings

From the figure, we observe that smart stub always provides the same level of average response time as the better binding between the two in all cases. Compared to the default SOAP binding, the smart stub reduces the response time of Google Web service by about 60%. Similarly, the performance of Amazon has improved by 34% – 80% and that of MapPoint has improved by 41% – 72%. In the WSBench case, sometimes RMI is better and other times SOAP is better. In all cases, the smart stub finds the right binding and provides performance improvement by up to 32%. Overall, we observe that the response time performance could be improved by 32% – 80% even when the target environment implements a highly optimized SOAP processing. We further evaluate the efficacy of the proposed mechanism in a multi-tier end-to-end configuration.



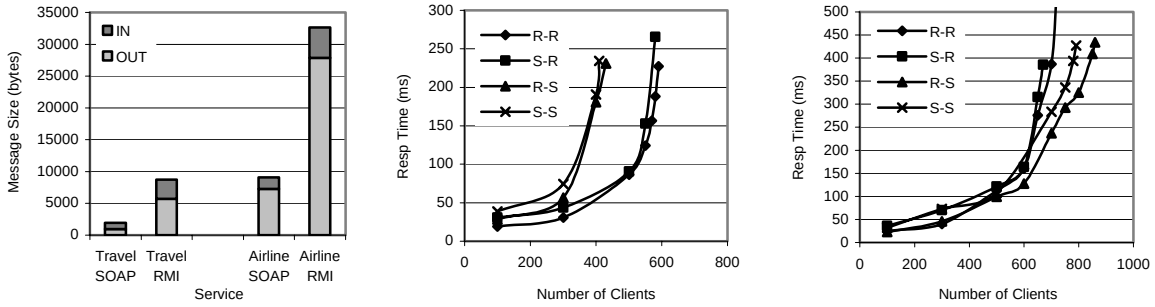
(a) Two services hosted on separate servers (3-tier) (b) Two services on the same server (2-tier)

Figure 7. Two Different Tiered Hosting Configurations for Travel Reservation Services

6.4 Case Study in Multi-tier Configuration

In the next experiment, we evaluate the effectiveness of our smart stub for different deployment configurations. To this end, we have developed a multi-tier Web service configuration based upon typical travel reservation services. In this scenario, a customer sends a request for possible travel plans to the main travel reservation service, which then request information for relevant flight schedules, rental car and hotel information from corresponding back-end services. The returned information from the back-end services is gathered by the Travel Reservation Service and the most relevant plan is sent back to the customer. For simplicity, we focus on the interaction between customer, Travel Reservation Service (*TRS*), and Flight Reservation Services (*FRS*).

Figure 7 illustrates this multi-tier services scenario with two different hosting configurations. Figure 7 (a) represents a 3-tier configuration where FRS is offered by a 3rd party and hosted on a separate machine (Host C). Figure 7 (b) represents a 2-tier case where FRS is hosted on the same server as TRS (Host B) where both services are run by the same company. For both cases, we assume that both RMI and SOAP are available. In our experiment setup, Hosts A and B are machines with a Pentium 4 3.0 GHz CPU and 3 GB memory. Host C has a Pentium 4 2.0 GHz CPU and 1.5 GB memory. All the hosts run Windows XP and connected via fast Ethernet. For application servers, we use WebSphere Application Server.



(a) Message size (b) 3-tier configuration (c) 2-tier configuration

Figure 8. Performance of Multi-tier Travel Reservation Services

Figure 8 presents the results for the our Travel Reservation Services scenario. Figure 8 (a) shows the messages sizes of each service. In this bar chart, for both TRS and FRS, the RMI messages are larger than

the SOAP messages. However, RMI message sizes of TRS are all less than 10KB, where RMI always wins over SOAP regardless the size comparison. For the 3-tier configuration, the smart stub for the customer on Host A picks an RMI using our binding selection algorithm. On the other hand, the smart stub of TRS picks a SOAP binding to connect to FRS. Figure 8 (b) provides the average response times when the number of clients are increased. Each selection pair is marked by X-Y, where X is a binding between the customer and TRS and Y is a binding between TRS and FRS. Our algorithm selects the binding pair 'R-S', which offers the best performance (the lowest response time and the highest throughput) over other binding pairs. For the 2-tier configuration, the smart stub of the customer picks an RMI binding the same way as in the 3-tier configuration. However, the smart stub of TRS picks an RMI to bind to FRS since both services are running on the same machine. Thus, the smart stubs select 'R-R' binding combination, which provides the best performance as shown in Figure 8 (c). This experiment demonstrates that our binding selection algorithm performs well even for multi-tier services with various hosting configurations.

7. Related Work

Conventional optimization techniques focus on individual components in an isolated manner. For example, techniques have been proposed to improve the performance of communication layers between distributed components. Such techniques include high-performance XML parsing, efficient implementation of SOAP protocols. Other techniques are data caching at network layer (e.g. SOAP result caching, and DB query caching), availing of multiple ways to communicate between the components (Web Service Invocation Framework or WSIF), and manual tuning during code inspection time.

However, existing optimization techniques have the following drawbacks. First they do not utilize the information available in the target environment where the software or service is deployed. Second, in many cases, those optimization techniques are applied independently with each other to improve the performance of a specific function or layer. Third, although such techniques may improve the performance of individual components, there is inherent overhead from multiple layers of abstraction. Finally, manual optimization techniques do not scale and cannot be applied in various domains.

There have been extensive research activities which have studied the performance of SOAP in various environments. Govindaraju et al. compared Java RMI and SOAP in scientific computing [1]. Kohlhoff et al. evaluated SOAP for business application such as real-time trading systems [2] and Elfwing et al. compared SOAP to CORBA in Web service environment [4]. In addition, Davis et al. examined the latency performance of several SOAP implementations as well as other protocols such as Java RMI and CORBA [3]. They all reported that the performance of SOAP usually is worse than Java RMI or CORBA in an order of magnitude, since SOAP uses text-based data representation.

Juric et al. have studied various approaches to tunnel RMI messages through firewalls, such as RMI over open ports, HTTP-to-port, HTTP-to-CGI, HTTP-to-servlet, and compared them with the SOAP-based approach in terms of functional and performance differences [6]. Although the RMI tunneling approaches provide clever mechanisms to reach services behind firewalls, such techniques are neither standardized nor recommended. Moreover, these tunneling-based approaches suffer from much poorer performance than that of native Java RMI or SOAP. Thus in our paper, we have excluded tunneling-based techniques for the binding selection problem.

Several works have looked at optimizing XML processing. Sundaresan et al. have proposed a compression technique that considers the structural part and the text of an XML document separately [7]. They have also developed a technique to quantify the characteristics of a document based on the schema and the document itself and determine the most appropriate compression algorithms for the document.

Such compression techniques can reduce the document size significant so that they can be easily stored on small devices and easily transferred.

In [18], Löwe et al. have studied the generation of schema-specific parsers for DTDs and restricted schemas. They found that the parsers show quite good performances and can be faster than non-validating parsers. Engelen et al. also utilized the XML schema information and proposed a compiler-based SOAP/XML engine, gSOAP, to improve the performance of SOAP/XML processing [19]. Based on XML schemas, schema compilers generate a parser which is specialized to process the valid instances of the source schemas. The resulting parser can achieve nice performance improvement while it rejects all other XML documents. For optimized generation of the schema-specific parsers, Engelen [21] and Chiu et al. [20] have utilized automata as an intermediate representation.

Caching is also used to improve the performance of SOAP processing, especially generating SOAP messages. Andresen et al. have proposed client-side and server-side caching in [16][17]. They developed three caching mechanisms: complete caching, body caching, and envelope and body caching. In [15], Abu-Ghazaleh et al. have proposed differential serialization, which helps in bypassing the serialization step for messages similar to previous ones. For certain workloads, these caching mechanisms can achieve nice speedups by a factor of two to ten.

In [8], Mukhi et al. presented an extension to the SOA architecture so that the service provider and the requestor can negotiate and dynamically change the configuration based on the run-time characteristics of the application. Web Service Invocation Framework (WSIF) [14] tries to enable dynamic binding selection by providing an XML schema extension and a set of Java libraries to process them. However, the intelligence to select the right binding for a certain run time environment is essentially a burden of application developers. In contrast, Blue Pencil tries to provide a programming environment where the task of the application programmer remains the same and actually selection occurs during the deployment time by the smart deployment module.

8. Summary and Future Work

This paper presented the motivations, the mechanisms, and the effectiveness of the deployment-time binding selection for distributed J2EE applications. Using the latest implementations of two popular application servers, JBoss and WebSphere, we have examined the performance of RMI and SOAP in the modern J2EE environment, and discovered that under certain workloads, SOAP can perform better than RMI. This led us to carefully investigate the binding selection problem. In particular, we faced the challenge to estimate the performance of SOAP and RMI using only static information such as the service interface definition, the location of the service, and the hint about the application server. To address this problem, we proposed a simple algorithm to predict the performance based on the message size estimation of the different protocols. From extensive performance evaluation using various real world workloads and internal performance benchmarks, we showed that our algorithm is simple yet effective. Overall, the proposed binding selection algorithm could improve the performance by 32% – 80% even with highly optimized SOAP implementations. For other application server environments, such as JBoss, the performance benefit of the proposed technique is expected to be even greater.

To extend the applicability of the proposed binding selection mechanism, we need to effectively handle legacy applications or third party applications that have been created outside the Blue Pencil framework. To handle such cases, we can develop a code transformation technique to generate a smart stub that corresponds to the client module by changing the existing stub code. We can investigate various compiler technologies in this space, including program slicing using forward and backward chaining, code pattern matching, and dead code elimination. Basically, the idea is to identify a code block to be removed (old

stub) and replace it with a new code block (a smart stub). Transforming a chunk of code to an equivalent code is a very difficult problem in general. However, we can leverage the fact that stub codes are typically machine generated and thus will likely to follow deterministic patterns.

References

[1] M. Govidaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon, "Requirements for and Evaluation of RMI Protocols for Scientific Computing," in *Proceedings of the IEEE/ACM SC2000 Conference (SC'00)*, Nov. 2000.

[2] C. Kohlhoff and R. Steele, "Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems," in *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, May 2003.

[3] D. Davis and M. Parashar, "Latency performance of SOAP implementations," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, May 2002.

[4] R. Elfwing, U. Paulsson, and L. Lundberg, "Performance of SOAP in Web Service Environment Compared to CORBA," in *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, Dec. 2002.

[5] S. Lee, K.-W. Lee, K. Ryu, J.-D. Choi, D. Verma, "Deployment Time Optimization of Distributed Applications," IBM Research Report, on-line document available at http://www.research.ibm.com/people/k/kangwon/publications/deployment_time_optimization.pdf, Nov. 2004.

[6] M. B. Juric, B. Kezmah, M. Hericko, I. Rozman, I. Vezocnik, "Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis," *ACM SIGPLAN Notices, vol. 39, no. 5, pp. 58-65*, May 2004.

[7] N. Sundaresan, R. Moussa, "Algorithms and Programming Models for Efficient Representation of XML for Internet Applications," in *Proceedings of the 10th International World Wide Web Conference (WWW10)*, May 2001.

[8] N. K. Mukhi, R. Konuru, F. Curbera, "Cooperative Middleware Specialization for Service Oriented Architectures," in *Proceedings of the 13th International World Wide Web Conference (WWW2004)*, May 2004.

[9] Microsoft MapPoint, http://www.microsoft.com/mappoint/products/webservice/default.mspx

[10] MapPoint demo application, Java Driving Directions, http://demo.mappoint.net/

[11] Amazon Web Services, http://www.amazon.com/gp/aws/landing.html

[12] Object Management Group, Inc., Common Object Request Broker Architecture: Core Specification, Version 3.0.3, on-line document at http://www.omg.org/cgi-bin/doc?formal/04-03-12, March 2004.

[13] Object Management Group, Inc., Java to IDL Language Mapping Specification, Version 1.2, on-line document at http://www.omg.org/cgi-bin/doc?formal/02-08-06, Aug. 2002.

[14] Web Service Invocation Framework, http://ws.apache.org/wsif/

[15] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju, "Differential Serialization for Optimized SOAP Performance," in *Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC-13)*, June 2004.

[16] K. Devaram and D. Andresen, "SOAP optimization via parameterized client-side caching," in *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, Nov. 2003.

[17] D. Andresen, D. Sexton, K. Devaram, and V. Ranganath, "LYE: a high-performance caching SOAP implementation," in *Proceedings of the 2004 International Conference on Parallel Processing (ICPP-04)*, Aug. 2004.

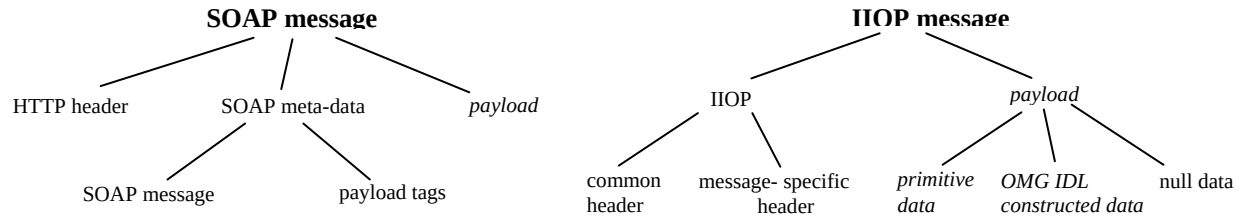
[18] W. Löwe, M. L. Noga, and T. S. Gaul, "Foundations of Fast Communication via XML," *Annals of Software Engineering, vol. 13, no. 1–4, pp. 357–379*, June 2002.

[19] R. A. van Engelen and K. A. Gallivan, "The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, May 2002.

[20] K. Chiu and W. Lu, "A Compiler-Based Approach to Schema-Specific XML Parsing," in *Proceedings of the First International Workshop on High Performance XML Parsing*, May 2004.
[21] R. A. van Engelen, "Constructing Finite State Automata for High-Performance XML Web Services," in *Proceedings of the 2004 International Symposium on Web Services and Application (ISWS'04)*, June 2004.

Appendix - Models for Message Size Estimation

To help the discussion about message size estimation, we present a diagram to show the structures of a SOAP message and an IIOP message in Figure 9. In the figure, variable size components whose size cannot be determined by WSDL are shown in italics.



(a) Structure of a SOAP message (b) Structure of an IIOP message

Figure 9. Structure of SOAP and IIOP Messages

A.1 SOAP message size

The structure of a SOAP message consists of three main parts: (a) HTTP header, (b) SOAP metadata, and (c) payload. Figure 9 (a) presents the structure of a SOAP message. Our empirical study shows that the size of HTTP headers varies in a small range.[6] So in our model, we treat them as constant.

The SOAP metadata can be classified into two parts: message information and payload tags. The SOAP message information includes envelope tag, header information, body tag, and name space. We find that the size of these fields can also be treated as constants,[7] and the length of target namespace can be obtained from a WSDL file. The size of payload tag is a sum of all payload parameter tags. Thus it can be written as:

$$S_{SOAP\_payload\_tag} = \sum T(e_i) \times N(e_i),$$

where e_i is an element of a set, E, which contains all elements that appear in the WSDL of a service, $T(e_i)$ is the tag size of e_i, which is almost fixed for each tag type, and $N(e_i)$ is the number of occurrences of e_i in a message. Thus $S_{SOAP\_payload\_tag}$ can be determined from a WSDL file.

Finally, the size of payload data is a sum of all data length.

$$S_{SOAP\_payload} = \sum L(d_i),$$

where d_i is an element of a set, D, which contains all payload data that appear in a message, and $L(d_i)$ is the length of a payload data in terms of byte. For example, L("Paul Smith") = 10, L(3.1415) = 6, and

[6] In all our experiments, the header size range was 352 – 356 bytes for a request message, and 184 – 196 bytes for a response message.
[7] In particular, envelope tags consume about 250 bytes, headers consume about 17 bytes, and body tags consume about 30 bytes.

L(2005-12-25) = 10. This is a variable part in the message estimation model, which cannot be determined by the static information in WSDL.

To summarize, the SOAP message size model consists of three parts: (1) constant part (e.g. HTTP header, envelope tag, etc.); (2) variable part that can be calculated from WSDL (e.g., namespace, payload tags); and (3) variable part that cannot be calculated from WSDL (i.e., payload data). Thus, in order to estimate average SOAP message size, we need to have an idea about the actual data values in the message.

A.2 RMI-IIOP message size

The structure of an RMI-IIOP message consists of two main parts: IIOP header and payload. Figure 9 (b) presents the structure of an IIOP message. An IIOP header consists of two parts: one that is common to all messages and the other that is specific to each message type e.g., request, reply, cancel request, etc. The common header contains: (1) identifier, (2) version, (3) byte order flag (which indicates little endian or big endian), (4) message type, and (5) message size [12]. The message specific headers are different from one message type to another. For example, a request header contains six fields including request id, target address, and operation name; whereas a reply header contains only three fields: request id, status, and service context. Since they follow well-defined patterns, they can be treated as constants for our purpose.[8]

The size of payload can be broken down into three parts: primitive type data, OMG IDL constructed types, and null data. Here the primitive type data are the ones defined in GIOP primitive types, not the primitive types of a specific language. GIOP defines thirteen primitive types including wchar (wide character), octet, short, long, float, and boolean. The mapping between GIOP primitive types and Java primitive types are defined in [13]. The size of primitive type data can be simply computed by counting how many data instances occur in an RMI message. Thus, we get:

$$S_{primitive\_type} = \sum [N(type_i) \times S(type_i)],$$

where $N(type_i)$ is the number of occurrences of $type_i$ data and $S(type_i)$ is the size of a $type_i$ data value. For fixed data types, this value can be easily calculated from the interface definitions in WSDL. However, we need actual data to calculate the size for variable length data, such as string.

Constructed data defined by the OMG IDL support complex data types such as struct, union, array, string, etc. A constructed data object consists of type definition part and the actual value part. When a constructed data type object first appears in an IIOP message, several fields about type definition must be inserted such as value tag, codebase URL, and type name. A value tag is a four byte flag (0x7fffff00 – 0x7fffffff) that contains information about the data type (e.g., IDL supported type or user-defined type). A codebase URL indicates the location of a codebase that is used for user-defined encoding. Type names contain fully qualified name of a data type. When the same type of constructed data objects appear more than once in a message, the later occurrence refers to the type definition information occurred in the first place. The length of the actual data can be calculated in the same was as in the primitive data case. Finally, each null parameter consumes fixed 4 bytes in an IIOP message.

As in the SOAP model, the RMI-IIOP message size model also consists of three parts: (1) constant part (e.g., common header, message-specific header), (2) variable part that can be calculated from WSDL (e.g., fixed primitive data types, fixed data types in a constructed data types), (3) variable part that cannot be calculated from WSDL (e.g., variable length primitive data types, variable length data types in a constructed data types).

[8] Our empirical data shows that request messages are about 180 bytes, and reply messages are about 40 bytes.