RC24252 (W0705-012) May 1, 2007
Computer Science

IBM Research Report

Improving the Performance of Web Services Using Deployment-Time Binding Selection

SangJeong Lee
Computer Science Division
KAIST
Yuseong-gu, Daejeon 305-701
Korea

Kyung Dong Ryu, Kang-Won Lee, Jong-Deok Choi
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Improving the Performance of Web Services Using Deployment-Time Binding Selection

SangJeong Lee[*] Kyung Dong Ryu[+] Kang-Won Lee[+] Jong-Deok Choi[+]
[*]Computer Science Div., KAIST
Yuseong-gu, Daejeon 305-701, Korea
peterlee@kaist.ac.kr
[+]IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.
{kryu, kangwon, jdchoi}@us.ibm.com

ABSTRACT

In this paper, we present a novel deployment-time binding selection framework for Web services to improve the performance. Using the information about target environments, we determine the best binding based on the availability and the accessibility of a service, and the performance characteristics of the bindings in a target environment. We have implemented the proposed mechanism as part of Eclipse-based development tools. We present an extensive performance evaluation of our methodology using benchmarks that we have created following public Web service interfaces, and emulating several e-business applications including a large scale legacy transaction processing system that runs on a mainframe.

1. Introduction

For seamless operation of business applications, distributed modules communicate using various binding technologies, such as RMI (Remote Method Invocation) and SOAP (Simple Object Access Protocol). One of the main goals of these technologies is to hide the complexity of underlying systems, and to provide communication among distributed modules regardless of their platform and language differences. In this way, software developers can focus on designing core algorithms while minimizing the effort in communication and data exchange. In the emerging Service Oriented Architecture (SOA), distributed modules will be implemented as Web services communicating via SOAP. However, a naïve adoption of such technologies can result in suboptimal performance due to heavyweight message processing.

Traditionally, the performance problem in middleware layers has been addressed by optimizing the performance of each layer. For example, related work has proposed to improve the SOAP performance by caching SOAP invocation results [1][2] or by accelerating XML processing [3][4][5]. On the other hand, manual optimizations based on best practices are common in real life software development practices. For example, a skilled software development team can design an application, which can adapt to changing run-time environments, and can select most appropriate access mechanisms based on the run-time performance of the application [6][7]. However, these approaches increase software development cost and prolongs the development and test cycles. Recently, more systematic approaches have been proposed to restructure distributed applications using the information about target operation environments at deployment time [8][9]. In [8], we introduced a general deployment time optimization framework called, *Blue Pencil*. In [9], Zhang et al. proposed a method to reduce the memory footprint during deployment.

This paper explores the performance gain achievable through careful binding selection during the deployment of Web service applications. In particular, we study the problem of automatically selecting the best performant binding among multiple alternatives such as SOAP, RMI or direct invocation. We find this problem to be nontrivial because the accessibility and performance characteristics of a binding changes depending on the configuration, e.g. application server implementations and firewalls. Also one of the old rule of thumbs, say RMI is much faster than SOAP [10][11][12], does not hold any more due to recent advances in SOAP/XML processing technologies. In this paper, we report that the performance gap has narrowed to a point that SOAP outperforms RMI under certain workload in a latest enterprise-grade application server.

The main contribution of this paper is twofold: (a) the design of a binding selection mechanism in the Blue Pencil framework, and (b) an extensive performance evaluation using several micro-benchmarks and a couple of business application benchmarks.

The proposed binding selection mechanism is based on a set of rules, and a novel performance prediction algorithm presented in the paper. These mechanisms are encoded in a lightweight interface module called, *binding proxy*. To facilitate the development of Web service applications, we have implemented a plugin to automatically generate the binding proxy in two different Eclipse-based integrated development environments (IDEs), namely Rational Application Developer (RAD) and WebSphere Developer for System z (WDz).

To evaluate the efficacy of the proposed approach, we have measured the performance of various Web service benchmarks in two different environments. First, in a regular application server environment, we used several benchmarks that we have created following popular Web services (e.g. Google, Amazon, MapPoint). We have also measured the performance using an internal benchmark emulating a financial transaction system. Secondly, we have tested the proposed mechanism in a legacy transaction processing system, called the Customer Information Control System (CICS) that runs on a mainframe. In this environment, we used a COBOL-based complex stock trading benchmark with Web interfaces. Overall, we discover that careful selection of bindings can reduce the average response time of a Web service by 34 – 80%. We also report the performance improvement in terms of throughput and CPU load. The latter metric is especially important for mainframes due to their typical high utilization.

The remainder of the paper is organized as follows: Section 2 motivates the main problem of this paper; Section 3 presents an overview of the Blue Pencil framework; Section 4 presents a special case for binding selection algorithm based on message size estimation; Sections 5 and 6 present performance result from the benchmarks in regular Web service and mainframe environments, respectively; and finally Section 7 concludes the paper with summary.

2. Binding Selection Problem

The efficiency of bindings between distributed components can significantly impact the end-to-end performance of a Web service. In the literature, the SOAP/XML message processing delay has been reported nontrivial compared to the network delay, the processing delay at servers and clients. Over the years, this trend has not changed fundamentally in one part due to the abundance of network infrastructure and computing resources at the backend, and in the other part due to the introduction of new optimization techniques (e.g. DB query caching).

To understand the breakdown of the end-to-end performance of a well-provisioned Web service, we performed a simple test on the Google SOAP Search API using the popular keywords listed on Zeitgeist. We sent 1,000 search requests separated by 60 seconds, and measured the end-to-end response time, the network latency, and the search time returned by the server.

Average end-to-end response time	347.17 ms
Average search time	63.35 ms
Average round trip time (RTT)	29 ms

From these numbers, we can compute that the average processing time of SOAP binding takes up to 254.82 ms, which is equivalent to 73.4 % of the overall service invocation time. Although this is an upper bound in some sense because we used popular keywords, whose search time will be minimal, we observe that binding processing latency constitutes a significant part in the end-to-end response time, and thus improving its efficiency is important.

As introduced in the previous section, we approach the issue of binding performance from the view point of selecting the most performant binding among possible alternatives. We assume this binding selection to happen during the deployment of an application since at that time we can discover the configuration of the target system and make an informed decision [8].

When selecting a binding, we consider a set of alternatives that satisfy the following properties:

- (P1) *Availability*: the binding mechanisms, which can be handled by the service and offered in the service description.
- (P2) *Accessibility*: among the bindings that satisfy P1, the bindings that can access the service from the relative location in a given configuration.
- (P3) *Performance*: among the bindings that satisfy P2, the binding that performs better than other bindings for a given service in a given operation environment.

These properties can be illustrated using the following example. Consider an application which calls a remote service. The service publishes that it can handle SOAP, RMI, and direct invocation (P1). The location of the service may be anywhere in the network, e.g. outside a corporate firewall, within the same subnet, within the same machine, or even in the same address space. If the service is within the same subnet, one can use either one of SOAP or RMI (P2). Now suppose the service is running in a certain application server, and the performance of RMI binding is better than SOAP. Then, one can use RMI for performance benefit (P3).

Typically, application developers do not have this information about the target operation environment, and may opt to use SOAP, which is guaranteed to work in all cases at the expense of potential performance hit. In the next section, we present how Blue Pencil supports this selection process.

3. Blue Pencil Framework

This section provides an overview of the Blue Pencil framework with an emphasis on the modules related to the binding selection problem. The Blue Pencil framework consists of five main components: (a) a component to support publishing available bindings of a service, (b) a component to generate a *binding proxy*, which is an intermediary to assist application developer, (c) a component to store various policy rules and a performance estimation logic, (d) a component to discover the configuration parameters of a target environment, and (e) a component to transform applications using program analysis techniques. In this paper, we do not consider code transformation, and refer interested readers to [8].

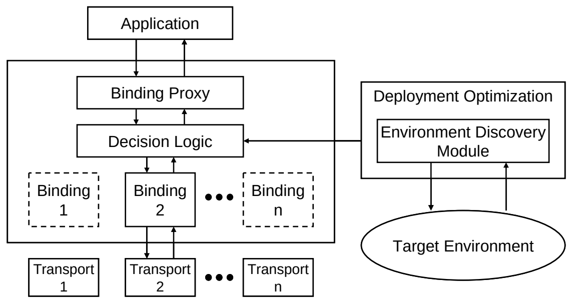


Figure 1. Structure of a Binding Proxy

3.1 Service Publication and Proxy Generation

To enable binding selection, the service provider must publish multiple bindings for a service. This can be done via WSDL (Web Services Description Language). The standard WSDL only supports SOAP and HTTP bindings, but we need to be able to publish additional binding types, e.g., RMI, JCA, JMS, and direct invocation. Blue Pencil employs Web Service Invocation Framework (WSIF) [7] for this purpose. The resulting service descriptor will specify which bindings are available at the server. To support the service publisher, we provide a development tool to auto-generate this extended service description [8].

The application programmer can create a client application for a service using the service descriptor. In Blue Pencil, we provide a concept of a *binding proxy*, which encapsulates multiple bindings to be used for remote service invocation. Essentially, this proxy is a lightweight intermediary that provides a uniform interface to hide the differences of various binding mechanisms. Figure 1 presents an abstract view of a binding proxy containing *n* bindings. Each binding type may interface with a different transport layer. For example, binding 1 = RMI and transport 1 = IIOP, binding 2 = SOAP and transport 2 = HTTP, and so on.

We note that a proxy contains a decision logic that selects a binding that has been derived from a set of policy rules. These policy rules may have been compiled by domain experts or published by middleware manufacturers or a third party. Alternatively, the decision logic may contain a simple algorithm to predict the performance of certain bindings in a particular configuration. Depending on the operation environment, static rules may be sufficient to make a decision, or algorithmic performance estimation may be required. In the next section, we present a performance prediction algorithm that is based on the message size estimation.

During the generation of a binding proxy, the generator module contacts a policy repository and collects the relevant rules. While we do not restrict how the policy rules should be specified or stored, our current design is inspired by the autonomic computing policy language (ACPL) [13]. At minimum ACPL can specify if-then rules that read "if the *condition* is true, then execute the *action*." In our context, conditions are typically complex Boolean expressions defined on the value space of configuration parameters, such as string (e.g. the manufacturer of an application server), boolean (e.g. the existence of a firewall), number (e.g. the version number of a middleware component). The use of a policy-based design provides simplicity, structure, and flexibility. For example, updating the decision logic for new types of application server is as easy as updating the rules.

3.2 Environment Discovery

When the user deploys a client application using the Blue Pencil deployment module, the module invokes an environment discovery function to discover the configuration parameters of the target environment. The parameters that we want to discover include information about the application package, information about the application server, relative location of the service, and other constraints between the application and the service.

Blue Pencil employs the following three approaches to discover these parameters. First, we discover information from a well known location in the system such as configuration files (e.g. deployment descriptor) or schema files specified by URLs. A similar technique is to examine the type of package files. We detect if an application runs on a Java platform by checking the type of application packages, e.g. checking if the package is an EAR (enterprise application archive), WAR (Web application archive), or JAR (Java archive) file.

Second, the discovery module uses a programmatic approach to query the features of the environment. In [8], we presented a scenario where the environment discovery module queries the capabilities implemented by a database management system (DBMS) via the Database Metadata interface. Using this mechanism, we can discover if a correlated subquery is supported by calling the supportsCorrelatedSubqueries method, for instance. This approach relies on a well-defined interface to examine the target environment.

Third, it employs various target-specific techniques. For instance, the discovery module finds out if the client runs on the same object request broker (ORB) as the service, by calling the javax.rmi.CORBA.Util.isLocal method. Discovering this information is useful since in this case, the client can invoke the service via a local Java call, which will be much faster than remote invocations. Another example is the case when the discovery module checks if the client is located in the same security domain as the server by querying the database that contains the network configuration information. Finally, the module checks if firewalls exist between the client and the server by sending probe packets (e.g. telnet connection request) to a particular port, and see if a response is returned.

The collected information in this step will be fed to the decision logic in a binding proxy as input, and the proxy will configure itself with the most appropriate binding. We note that this configuration discovery and binding selection process may be used during a major reconfiguration of the application not just during the initial deployment.

4. Performance Predictor Design

On the contrary to the traditional belief that SOAP involves much communication overhead[1], we observed that the binding performance of SOAP is sometimes better than that of RMI for certain workload in the latest enterprise-grade application server implementation, namely WebSphere Application Server v6 that employs various SOAP/XML acceleration technologies such as pre-compiled serializers and deserializers based on schema-specific XML parsing technologies [3][4]. Therefore, some binding decisions cannot be made from static rules. In this section, we first explore the performance inversion issue happening in WAS, then we design a novel performance predictor based on message size estimation for WAS.

4.1 Performance Variance

In this section, we present the performance of two different types of Web services to motivate that performance prediction is necessary. For the simplicity of exposition, we consider only SOAP and RMI-IIOP bindings in WAS.

For the workload, we use a benchmark modeling the Google search service and a benchmark for a financial transaction service. Detailed description of these workloads and the testbed environment is presented in Section 5.1. For our discussion in this section, we only need to know that: (a) both the client and the server were running in a local testbed environment, (b) the control parameter for search is the number of search results, and (c) the control parameter for the financial benchmark is the size of request and response messages. Table 1 summarizes the result.

For Google services, RMI performs about 2.5 times better both in terms of response time and throughput. However, with the financial benchmark, performance of SOAP is sometimes better than RMI under certain workload, while for other workload it is the other way around. These results show that the performance of binding varies depending on two parameters: the remote service type (Google vs. financial), and the message type (1KB vs. 100KB).

[1] We also have tested the binding performance of SOAP and RMI using JBoss and WebLogic. The results show that RMI always outperforms SOAP with our benchmarks.

Table 1. SOAP and RMI Performance with WAS

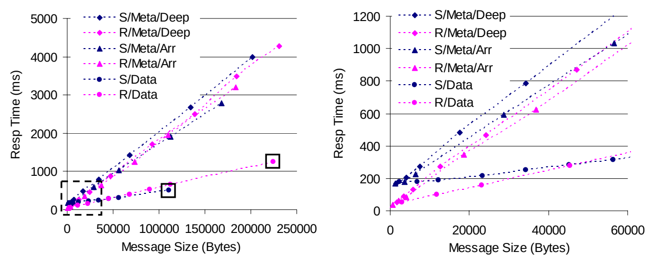
	Parameter	Response Time (msec)			Throughput (reqs/sec)		
		SOAP	RMI	SOAP/RMI	SOAP	RMI	RMI/SOAP
Google	10 results	235.7	82.6	**2.85x**	212.1	605.3	**2.85x**
	100 results	1252.7	515.8	**2.43x**	39.9	96.9	**2.43x**
financial	1KB	183.3	137.6	**1.33x**	272.8	363.4	**1.33x**
	10KB	370.8	553.5	**0.67x**	134.8	90.3	**0.67x**
	100KB	3075.1	4495.5	**0.68x**	16.3	11.1	**0.68x**

To understand these results better, we analyzed the messages exchanged between the client and the server. From this study, we discovered that the slow performance of RMI in financial benchmark is related to an extensive use of Java String objects and Java Calendar objects. More precisely, IIOP uses UTF-16 encoding, consuming two bytes for each character, whereas SOAP uses UTF-8, which requires only one byte for each character. Furthermore, Calendar objects become extremely bloated in RMI-IIOP, adding about 1,000 bytes to describe the Calendar type, consisting 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". From this study, we infer that the data types contained in the messages have some impact on the binding performance.

4.2 Message Format and Binding Performance

To gain further insight, we investigate the correlation between the message format and the binding performance using synthetic workload with controllable parameters. For this study, we have generated three types of synthetic messages: (1) metadata-oriented messages with simple string/integer data and deep class nesting up to 10 levels (denoted by *Meta/Deep*); (2) metadata-oriented messages with simple data and arrays containing 100 – 3,000 elements (denoted by *Meta/Array*); (3) data-oriented messages with many integer and string fields (denoted by *Data*).

Figure 2 presents the correlation between the response time and the message size for different types of objects with SOAP and RMI-IIOP. From the chart, we make the following observations. First, the message size for the same object differs significantly for different bindings, e.g. the rightmost points marked by squares in Figure 2(a). Second, the initial response time (i.e. the y-intercept) of each curve shown in the magnified chart in Figure 2(b) is much higher for SOAP (about 160 ms) than RMI (about 20 ms). This is due to the heavy initial cost of extra HTTP routing via a Web container to deliver requests to an EJB container. Thus, RMI offers better performance for small



(a) Overall graph (b) Magnified graph

Figure 2. Relationship of Message Size and Response Time

messages (< 10KB). Finally, the slopes for RMI and SOAP plots are almost equal for the *same type* of objects (Figure 2(a)). In particular, the slope of metadata-oriented messages and that of data-oriented messages are different but both protocols show similar slopes.

The first observation is interesting, but does not help us in predicting the performance. The second observation can be easily encoded as a static rule, thus we do not consider in performance prediction. The third observation is potentially valuable for designing a performance predictor for WAS. If this trend holds in general, then we can predict the binding performance from the size of a message – because the size-performance slope is the same for the same message (and different bindings). When we consider the ongoing trend in optimizing binding performance this observation makes an intuitive sense. For example, as the SOAP/XML processing modules become optimized, the parser and the (de)serializer will process messages without the inefficiencies of unnecessary data conversion and data copy between layers, or suboptimal schema validation. In the end, it will make only one pass for parsing and (de)serialization, whose processing time will be proportional to the message sizes, and it may take similar time to process different types of bindings.[2]

4.3 Message Size Estimation

To build a performance predictor for WAS based on the intuition from the previous section, we proceed to develop a simple model to estimate the average message size. As a proof of concept we just consider SOAP and RMI-IIOP protocols, but it is easy to generalize the idea to include other protocols. The main challenge in this step is to estimate the message size using only the static information available at deployment time.

4.3.1 SOAP message structure

A SOAP message is structured with three main parts: HTTP header, SOAP metadata, and payload. Empirically, we find that the size of HTTP headers varies in a small range. Thus, in our model, we treat them as constant. The SOAP metadata can be further divided into two parts: message information (e.g. header information and name space) and payload tags. The size of these fields can also be either treated as constants, or obtained from a WSDL file. Finally, the size of payload data is a sum of all data length. This is a variable part in the message, which cannot be determined by the static information in WSDL.

4.3.2 RMI-IIOP message structure

The structure of an RMI-IIOP message consists of two main parts: IIOP header and payload. An IIOP header again consists of message ID, version, and the variable part depending on the message type (e.g., request, reply, cancel request). The size of IIOP headers can be treated as constants for our purpose since their size is typically small with a small variance.

The size of payload can be divided into three parts: GIOP primitive type data (e.g. octet, short, long), OMG IDL constructed types, and null data. 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. The size of fixed data types can be calculated from WSDL. However, we need actual data value to calculate the size for variable length data, such as string.

Constructed data defined by the OMG IDL supports complex data types (e.g. struct, array, etc.). When a constructed data type object first appears in a message, several fields about type definition must be inserted such as codebase URL, and type name, which are mostly constant. When the same type of constructed data objects appear more than once in a message, the later occurrence refers to the original type definition. Finally, each null data consumes fixed 4 bytes in an IIOP message.

4.3.3 Performance Predictor Design

In general, both SOAP and RMI message consists of three parts: (1) constant part, (2) variable part that can be calculated from the static information, and (3) variable part that cannot be calculated from the static information, such as payload data. The first two parts can be determined from the interface definition. However, the third one must be determined in some other way, and thus requires further discussion.

In this paper, we consider two ways to estimate the size of the variable parts. The first one is a grey box approach that uses general hints for a given service type. For

[2] This statement assumes that the message size counts only the bytes that are important for parsing and (de)serialization. If some part of messages will be skipped by the parsers, they should not be counted.

example, when we deploy a client for a bank application, we can use a general parameter space that has been derived from some other bank services. While the new application may exhibit different workload characteristics, this approach may be useful in practice. The second approach tries to estimate the size using some default value range that is large enough to cover most probable cases. This approach is useful for primitive types (e.g. numbers) since they have clear bounds. For complex types, e.g. arrays and strings, we can place an upper bound from empirical values or from a reasonable limit on the maximum message size. This approach is effective since most business applications exchange relatively small message size. An exception to this rule of thumb is scientific applications in a Grid environment; validating our design in such a scenario is a future research topic.

Once we have a total value space for the variable data types, we can design an estimation algorithm for average message size. This estimation algorithm will effectively predict the performance of the bindings (from the third observation in Section 4.2). In this paper, we use a simple algorithm that takes a majority vote on all the message size estimates on the entire parameter space. For example, if the number of size estimate that RMI message will be smallest is *f*(*RMI*), that of SOAP will be smaller is *f*(*SOAP*), and that of JCA will be smaller is *f*(*JCA*), then the predictor simply chooses the binding with the argmax *f*(*x*). We note that this performance prediction result will be combined with the static rule (such as the one from the second observation in Section 4.2), before making the final decision. We validate the performance of this predictor design in the next section.

5. Performance Evaluation in WAS

This section provides performance evaluation of the proposed prediction algorithm. Then it presents the results from various Web services in the WebSphere Application Server environment.

5.1 Workload

We have created benchmarks modeling various real-world applications with different characteristics: **Amazon** search, **Google** search, Microsoft **MapPoint**, and **Financial** Transaction. The Google service provides keyword-based search, whose messages are heavily string-based and relatively small. The Amazon service provides operation to inquire products, customers, and sellers, modeling online shopping site. The Microsoft MapPoint provides Internet map services such as find-address, calculate-driving-directions, and retrieve-maps. The find-address operation consists of simple string messages, while calculating route and map operations involve heterogeneous data types including binary data. The Financial Transaction benchmark models money transfer operations between banks. Thus its message structure is complex, involving arrays, mixed data types; and the message size can be large, up to a few hundred Kbytes.

Table 2. Performance Prediction
(hint A: max message size, B: strlen, C: null%)

Service	Vote Percentage of Majority			Decisio n
	No hint	Hint: A	Hint: A, B, C	
Google - *doGoogleSearch*	91.7 %	91.7 %	88.9 %	RMI
Amazon - *itemSearch*	62.5 %	62.5 %	100 %	RMI
MapPoint - *findAddress*	84.2 %	100 %	100 %	RMI
MapPoint - *calculateRoute*	81.6 %	81.6 %	100 %	RMI
MapPoint - *getMap*	95.2 %	95.2 %	100 %	RMI
Financial - *handleAddShort1*	97.1 %	97.1 %	100 %	RMI
Financial - *handleAddShort10*	77.1 %	77.1 %	100 %	SOAP
Financial - *handleAddShort100*	55.2 %	55.2 %	100 %	SOAP
TRS - *getTravelPlan*	100 %	100%	100 %	RMI
TRS - *getFlightInfo*	62.0 %	62.0 %	100 %	SOAP

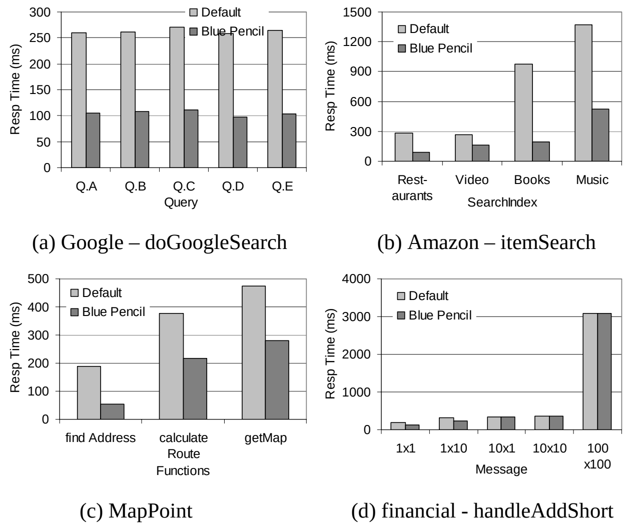
As key parameters that characterize the workload, we consider average string length, binary object size, null probability, average array size. The null probability represents how many parameters are unspecified when a service is invoked. Generic Web services typically have high null probabilities (e.g. over 80% for Amazon) because a single service interface is used for different purposes based on the parameters (e.g. book or restaurant search). Except the financial benchmark (up to 264), we have discovered that the average array size is relatively small (less than 7) in our workload.

5.2 Efficacy of Performance Prediction

In this section, we evaluate the effectiveness of binding selection based upon the proposed performance estimation algorithm. As we discussed earlier, we explore two different approaches: (1) estimation without hint, and (2) estimation with hint at various levels. For the case without hint, we set the maximum message size to 300 KB and the other parameters to have values in the following space:

- String length: 5 – 75
- Binary data size: 1.5KB – 45KB
- Null probability: 0% - 90%
- Array size: 2, 5, 10

There are three types of hint that we consider. The first one is the maximum message size: we bound simple messages (e.g. text search) with 100KB, and complex messages (e.g. financial data) with 300KB (Hint A). The second hint is the average size of string from the measurement (Hint B). The third hint is a tighter range of null probability, ± 10% of the measured value for each service (Hint C). Table 2 presents the accuracy (or confidence level) of the performance predictor. For each benchmark, it shows the percentage of votes for the final decision with different levels of hint. For instance, the predictor selected RMI for Google service based on the votes for RMI – 91.7%, 91.7%, 88.9% for the three levels of hints, respectively. The resulting binding of this voting



(a) Google – doGoogleSearch (b) Amazon – itemSearch

(c) MapPoint (d) financial - handleAddShort

Figure 3. Average Response Time with Benchmarks

is finally selected. We use these results for the evaluation in the subsequent sections.

5.3 Performance Impact

We now present the performance benefits from the proposed binding selection mechanism in our local testbed. The testbed consists of a Pentium 4, 2 GHz machine with 1.5GB memory (the server), and a Pentium 4, 3 GHz machine with 3.0 GB memory (the client) both running Windows XP. In all our experiment, we assume that the client and the server are on the same network to demonstrate the selection between RMI and SOAP. On the client side, we have created multiple threads, each simulating the behavior of an individual client to keep the server busy. We assume the default binding (without Blue Pencil) to be SOAP binding since it is widely used in Web services for general connectivity. In all cases, we present the average response time measured from 5,000 queries. Figure 3 presents the results.

Figure 3(a) shows the results from the Google service with different search keywords (Q.A – Q.E). From the figure, we observe that the response time improves by 59 – 62% with Blue Pencil, which employs deployment-time binding selection with message size estimation. Figure 3(b) presents the results from Amazon service with different types of search (restaurant, video, books, music). In this case also, we observe performance improvement with Blue Pencil by 34 – 80%. Figure 3(c) presents the results from MapPoint service with different types of queries (find-address, calculate-route, get-map), which differ significantly in terms of message format and size. Overall, we observe performance improvement of 41 – 72%. Finally, Figure 3(d) shows the results from financial benchmark with different request/response sizes (1KB x 1KB – 100KB x 100KB). From the figure, we observe that Blue Pencil gives a performance benefit for small

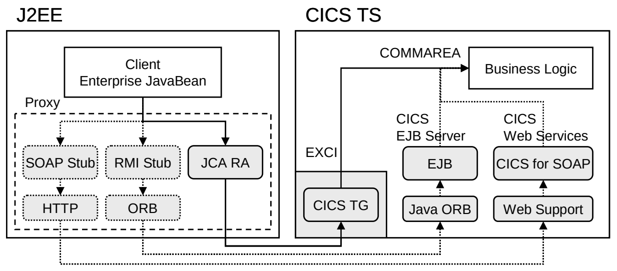


Figure 4. CICS binding proxy and three CICS connection mechanisms

messages, but for large messages it performs the same as the default configuration. This is due to the messages that are heavy on String and Calendar objects, which result in poor RMI performance. In this case, Blue Pencil selects SOAP, which is the correct choice.

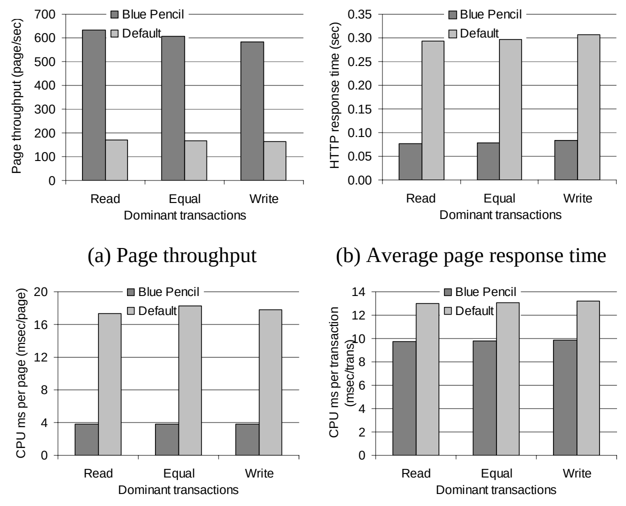
6. Optimizing Web Services of Legacy Information System

The wide popularity of Web services has affected not only the regular server systems, but also the legacy information processing systems. For example, the Customer Information Control System (CICS) of IBM now provides SOAP binding interfaces so that a Web services client can communicate with its applications.

Originally, CICS was used for terminal based operation. Later it added a mechanism for information access by other applications programmatically through shared memory area, called the Communications Area (COMMAREA). It now offers remote access mechanisms for Web Service requester, Java Servlets or EJBs running on J2EE, .NET applications, and Web browsers. The box denoted by label “CICS TS” in Figure 4 illustrates three main access mechanisms: (a) J2EE Connector Architecture (JCA), (b) Web Services using SOAP, and (c) Enterprise JavaBean.

For binding selection in CICS, service providers must publish multiple CICS connectors. To denote JCA, we use *cics* namespace defined by WSIF. We then need a proxy generation support for CICS application developers. For this we have extended the WebSphere Developer for System z (WDz), an Eclipse-based IDE for mainframes, to include WSDL extension and proxy generation features. The structure of generated binding proxy for CICS is illustrated in the box denoted by label “J2EE” in Figure 4.

We use a benchmark application to evaluate the benefit of binding selection, called *Trader*, which emulates stock trading activities through Web applications. Trader consists of two parts: the CICS part that manages persistent information about companies and customers in COBOL, and the J2EE part that handles the presentation of retrieved information through servlets and accesses CICS through J2EE EJBs. It mimics the state transition involved in stock trading, and the benchmark is driven by



(a) Page throughput (b) Average page response time

(c) CPU msec per page of WAS (d) CPU msec per transaction of CICS

Figure 5. Performance result with Trader application

a script that automatically navigates the pages based on predefined transition probabilities.

For evaluation, we have set up CICS Transaction Server v3.1 and WebSphere Application Server v6.1. In particular, to represent the current server consolidation trend in system z, we installed the client, CICS TS and WAS on the same mainframe machine but on different logical partitions (LPARs). However, CICS TG was run on the same LPAR as CICS TS.

For simplicity we discuss the binding selection between SOAP and JCA. Existing reports on the performance of SOAP connector teaches us that it is much slower (6 to 7 times) than that of JCA with CICS TG. Thus we have encoded a simple rule that when JCA is available and accessible, use JCA over other bindings. We are now ready to quantify how much performance benefit such a simple transformation can bring.

Now we examine the performance gains from Trader. For this, we consider three application modes: “Read”, which primarily models reading data from the CICS server (by weighing the probability of transition to the ‘Quotes’ state *k* times higher than update operations); “Write” which models updating data (by assigning *k* times more weights to the update probability); and “Equal”, which has same rate of read and write operations. Figure 5 presents the result for Trader when $k = 3$. We examine throughput, response time, and CPU ms/trans measured at WAS and CICS. From the figure, we observe significant performance gains in terms of response time, throughput, and the processing overhead by selecting JCA over SOAP binding – up to 73.7 % reduction in response time, 275 % increase in throughput, 78.9 % reduction of CPU load of WAS, and 25.3 % reduction in CPU load of CICS.

7. Conclusion

This paper presents the Blue Pencil framework that is designed to improve the performance of Web services. The framework automates the process of checking the availability, accessibility and performance properties of multiple bindings in the target environment, and selecting the best performing binding at the deployment time. With the extensive performance evaluation, we present that the deployment-time binding selection results in the significant performance improvement of J2EE Web services – up to 82 % reduction in response time. The result with CICS services provides more promising performance gain – up to 74 % reduction in response time, 275 % increase in throughput, 78.9 % and 25.3% reduction in CPU load of WAS and CICS, respectively. It is important to note that Blue Pencil provides these performance gains without burdening developers with new programming models or libraries; Blue Pencil features are enabled transparently and easily through plugins in RAD and WDz.

REFERENCES

[1] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju, “Differential Serialization for Optimized SOAP Performance,” in *Proc. of HPDC-13*, June 2004.

[2] D. Andresen, D. Sexton, K. Devaram, and V. Ranganath, “LYE: a high-performance caching SOAP implementation,” in *Proc of ICPP-04*, Aug. 2004.

[3] M. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, and A. Heifets, “XML screamer: an integrated approach to high performance XML parsing, validation and deserialization,” in *Proc of WWW 2006*, May 2006.

[4] R. A. van Engelen, “Constructing Finite State Automata for High-Performance XML Web Services,” in *Proc. of ISWS’04*, June 2004.

[5] R. A. van Engelen and K. A. Gallivan, “The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks,” in *Proc. of CCGrid2002*, May 2002.

[6] N. K. Mukhi, R. Konuru, and F. Curbera, “Cooperative Middleware Specialization for Service Oriented Architectures,” in *Proc. of WWW2004*, May 2004.

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

[8] S. Lee, K.-W. Lee, K. D. Ryu, J.-D. Choi, and D. Verma, “Deployment Time Performance Optimization of Internet Services,” in *Proc of Globecom 2006*, Nov. 2006.

[9] O. Demir, P. Devanbu, E. Wohlstadter, and S. Tai, “Optimizing Layered Middleware,” in *Proc of SEM 2005*, Sep. 2005.

[10] M. Govidaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon, “Requirements for and Evaluation of RMI Protocols for Scientific Computing,” in *Proc. of IEEE/ACM SC2000 Conference (SC’00)*, Nov. 2000.

[11] C. Kohlhoff and R. Steele, “Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems,” in *Proc. of WWW2003*, May 2003.

[12] D. Davis and M. Parashar, “Latency performance of SOAP implementations,” in *Proc. of CCGrid2002*, May 2002.

[13] Policy Management for Autonomic Computing, on-line document available at http://www.alphaworks.ibm.com/tech/pmac, March 2005.