December 20, 2006

RT0699
Communications 19 pages

Research Report

Early Capacity Testing of an Enterprise Service Bus

Ken Ueno, Michiaki Tatsubori

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

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

International Journal of Web Services Research , Vol.X, No.X, 200X

Early Capacity Testing of an Enterprise Service Bus

Ken Ueno Michiaki Tatsubori
Tokyo Research Laboratory, IBM Research
{kenueno, mich}@jp.ibm.com

ABSTRACT:

An enterprise service-oriented architecture is typically realized on a messaging infrastructure called an Enterprise Service Bus (ESB). An ESB is a bus which delivers messages from service requesters to service providers. Since it sits between the service requesters and providers, it is not appropriate to use any existing capacity planning methodology for servers, such as modeling, to estimate an ESB's capacity. There are programs that run on an ESB called mediation modules. Their functionalities vary and depend on how people use the ESB. This creates difficulties for capacity planning and performance evaluation. This paper proposes a performance evaluation methodology and techniques for ESBs. We actually run the ESB on a real machine while providing a pseudo-environment around it. In order to ease setting up the environment we provide ultra-light service requestors and service providers for the ESB under test. We show that the proposed mock environment can be set up with practical hardware resources available at the time of hardware resource assessment. Our experimental results showed that the testing results with our mock environment are equivalent to the results in the real environment.

KEY WORDS:
Capacity Testing, ESB, SOA, Web Service

Introduction

An enterprise service bus (ESB) is part of an infrastructure, a messaging bus based on Web Service standards [9,12,16]. It is a platform for Web Service intermediaries [5], and fills a core infrastructural role in a service-oriented architecture (SOA), which is a collection of services communicating with each other [1,15]. The first middleware providing ESB functionality was introduced in 2004. One of the unique features of an ESB is the use of component programs called mediation modules. These are programs that run on an ESB. There are various kinds, depending on how the ESB is used. Even though the technical concept of an ESB is not completely new, the increased adoption of SOA in industry raises new engineering challenges for research. Performance estimation of an ESB is one such challenge.

Predicting the performance of an ESB is different from predictions for traditional application servers, for which many studies have been done [2,3,10,14]. This is because an ESB plays not just a server role but also plays a client role for multiple service providers. That also means that the methods used for a J2EE server's performance evaluation [5] aren't suitable for an ESB. The mediation functionalities also cause some differences in how ESB performance is evaluated compared to evaluating simple intermediaries like TCP/IP network routers. Nevertheless, performance estimation of the ESB in the capacity planning phase, which happens at a very early stage in the project lifecycle, is critical to a successful project. If the capacity of an ESB used for a system is overestimated, we might need a significant change of the architecture, a large effort in system performance tuning during development or deployment, and/or extra funding for additional hardware. If the capacity is underestimated, we might overestimate costs or seek unneeded compromises from the system stakeholders.

The goal of the work presented in this paper is to provide a practical solution for IT architects assessing the capacity of an ESB. Though many researchers have addressed this issue through model-based approaches [4,11], they often require elemental performance measurements and sophisticated modeling of the entire system, which is usually not feasible for complex systems. In an ESB, modeling involves intermediate components called mediation modules that provide functions such as routing and protocol conversion, and configurations can vary for each system. This complexity makes it difficult to estimate the performance of an ESB with a model-based approach.

In this paper, we propose a capacity testing technique for ESBs during the phase of capacity planning, which is conducted very early in a project's lifetime. Our approach is to use a lightweight Web Service provider and a lightweight Web Service client for performance testing of the ESB. With the proposed technique, designers can evaluate the ESB system capacity with a small hardware environment consisting of Web Service requesters and providers. In contrast to most capacity planning techniques, the results of this capacity testing can reveal the actual maximum capacity of the ESB server on the specific platform.

For our technique, we designed and implemented a novel framework for a lightweight Web Service provider and a lightweight Web Service client. In the framework, the lightweight service provider is implemented based on our mock environment technologies, while the lightweight Web Service client is implemented based on a common HTTP load generator. The experimentation environment built with our framework allows us to measure the potential capacity of an ESB accurately with inexpensive hardware.

We built this kind of lightweight environment for a banking application example and evaluated the validity of our approach. As a result of our experiments, we observed that the measured results with our lightweight environment are almost identical to those with the real environment.
The rest of the paper is organized as follows: First, we introduce the ESB concept and point out the problems of an ordinary project using an ESB for a distributed system. Then we propose an ESB capacity testing approach for the capacity planning phase in a lightweight service provider environment. The fourth section describes the detailed implementation techniques for the proposed lightweight ESB environment and the fifth section shows the experimental results with our implementation. After discussing related work and several discussions on the applicability of our approach, we conclude the paper.

Capacity Planning of an ESB

In this section, we discuss the motivating background of our research. First, we introduce and explain the notion of an Enterprise Service Bus (ESB) and the components of typical ESBs. Then we highlight a problematic part of development in a scenario involving system development with an ESB.

Enterprise Service Bus

An ESB refers to a software architecture construct implemented by using technologies found in a category of middleware infrastructure products usually based on Web Service (WS) standards. It provides foundational services for more complex Service-Oriented Architectures (SOA) via an

XML-based messaging engine (the bus), and thus provides an abstraction layer on top of an enterprise messaging system that allows integration architects to exploit the messaging without writing code.

The technical concept of an ESB is not especially new, but it is coming to play a significant role in the enterprise computing world as SOA is becoming widely adopted. In addition to the standardization efforts for the technologies around SOA, a reason for its importance to practitioners is that it allows faster and cheaper accommodation of existing systems and also scales from point solutions to enterprise-wide deployments. Also, it provides increased flexibility in changing systems as requirements change.

Since an ESB is a bus, it is topologically located at an intermediary position between two types of SOA participants: service requestors and service providers. These are usually WS clients and WS servers, respectively, usually using SOAP over HTTP.

From the seven-layer stack perspective of OSI, the presentation layer is provided by SOAP headers and SOAP bodies embedded in SOAP envelopes. An ESB recognizes the layer for protocol and data conversions, and for transport-independent policies such as routing, caching and security.

Typical Components around an ESB

The typical topology of a distributed system involving an ESB consists of five tiers. Figure 1 depicts the topology and participating components in the distributed system. Though various software vendors provide various ESB products with different capabilities, the basic architectural form of a distributed system with an ESB is in this typical topology. This belief is based on a large number of real ESB customer experiences, and we believe that customers currently accept this scenario as typical.

Most customers have been using Web browsers as their user interfaces, which is the first tier of the five tiers in Figure 1, while the last tier is provided by databases. First of all, when using a Web browser, each end user accesses Web application servers that dynamically generate Web pages. A server running a Web application, the second tier, is typically implemented with a framework like JSP or servlets and is often built using an enterprise application container such as Apache Geronimo, JBoss Application Server, IBM WebSphere, or Microsoft ASP.NET. When the second tier is accessed, it invokes the Web Services by generating SOAP requests. The generated SOAP requests are typically sent using the HTTP. With an ESB, that request will be sent to the third-tier intermediary, which is an ESB. Then the ESB will perform some processing (if required) and forward the request to the target service provider, the fourth-tier of the topology. This tier is also provided by Web application servers. After the target service provider receives the request, it does any required operations, such as accessing an external database (in the fifth-tier) and processing any required business logic.

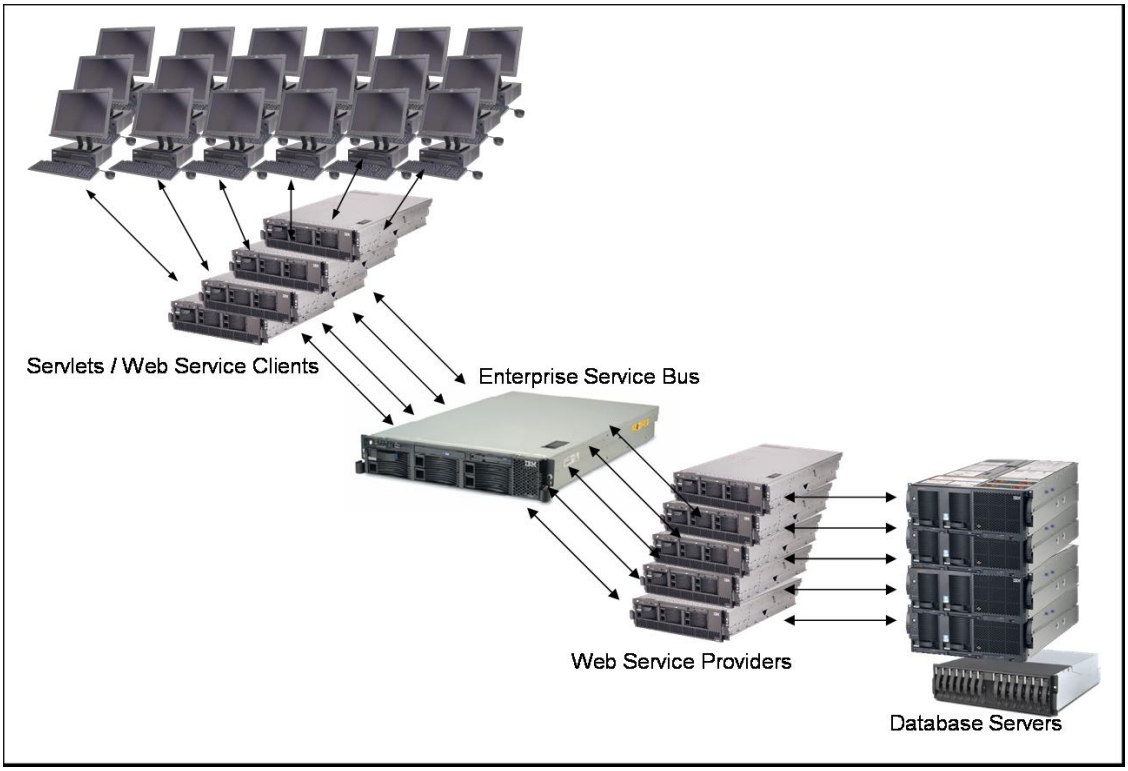

Figure 1. A typical topology of a distributed system with an ESB

Capacity Planning

Among the various problems in developing a distributed system with an ESB, the problem we address in this paper is capacity planning. The objective of capacity planning is to determine what hardware to buy to meet cost, performance, and scalability requirements. In a system development project, this happens at a very early stage of the project lifecycle [6] as shown in Figure 2. In many cases, we need to finalize the investment budget for a new project at that time. Usually this is in the abstract architecting phase of the project, just after the requirements analysis. When architecting a system, we need to evaluate the to-be-developed architecture to know whether or not it meets the requirements of the system stakeholders. Such an evaluation involves analyzing expected performance for the candidate hardware platforms. Based on the evaluated cost-performance of an architected system, an architect may need to negotiate with stakeholders for compromises on requirements or may proceed by refining the architecture to start detailed design and implementation as long as the design meets the requirements.

In a typical capacity-planning methodology [13], the simplest method for capacity planning is trend analysis. We collect data on current (and any past) system utilization and use it to estimate future utilization. The following are typical steps for capacity planning:
- Identify workload patterns
- Measure performance with the current configuration (infrastructure)
- Analyze trends and estimate growth trends and performance targets [17]
- Model the software and hardware infrastructure

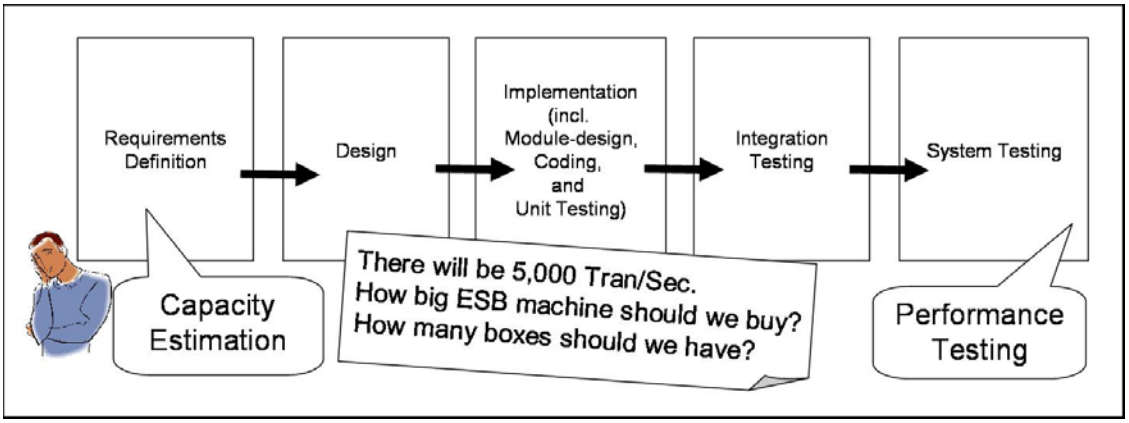

Figure 2. Capacity Estimation in Early Development Cycle

In ESB capacity planning, however, there is a problem at the second step, the step of measuring the performance on the current configuration [18]. Since an ESB is a relatively new infrastructure category, most of the projects that need capacity planning will not have any ESB in the existing configuration. Therefore, most new ESB systems will be purchased without prior experience. In addition, the configuration of an ESB varies for each project because of the intermediate components called mediation modules, which provide functionalities to customize the behavior of the bus, such as routing and protocol conversion.

Failure of capacity estimation can lead to disaster for a project. If the capacity of an ESB for a system is overestimated, it might cause a significant change of the architecture, increased work in system performance tuning during development or deployment, and/or the purchase of extra hardware. If the capacity is underestimated, we might overprice the project or request unneeded compromises from the system stakeholders.

Early Capacity Testing with a Lightweight Environment

In this section, we will discuss early capacity testing for an ESB. We will also discuss requirements for capacity testing by using a typical misconfiguration of an ESB in a testing environment. Then we will discuss our mock environment and finally describe the parameters that we can configure with our lightweight environment.

Early Capacity Testing

What we are proposing here is a method for early capacity testing. Though capacity testing is usually conducted at a very late phase of a project, we strongly suggest that it should be conducted during an early phase, specifically during the capacity planning phase of the project. Capacity testing only reveals the ultimate limits of the servers that are used for the capacity testing rather than the capabilities of the production system. Such a test determines the capacity of a specific system rather than the entire production system infrastructure. We should do proper capacity testing during capacity planning as part of making the investment decisions.

Requirements for ESB Capacity Testing

When we evaluate the ESB performance capacity, the most important factor is that the ESB has to be under high load. Then we can saturate the ESB system (or approach saturation) to determine its maximum performance. While evaluating the ESB performance, the Web Service clients need to send and receive enormous numbers of messages through the ESB. We need to have Web Service clients and service providers that can handle such heavy message traffic.

Based on our actual experiences with ESB customers, we realized that the typical topology of many ESB-based systems consists of five tiers: Web browsers, Servlets (Web Service requestors), the ESB, Web Service providers, and backend servers such as database servers. The main reason why many customers have this topology is that the roles of both the end user tier and the backend tier are the same as a typical Web application (J2EE) topology. The significant differences lie in the middle tiers.

We need to be careful about the differences between a real production system environment and a performance evaluation system in a lab. In a real production system environment, there are usually multiple large service providers as well as Web Service clients. Compared to a real production environment, a typical performance evaluation lab system has a relatively small machine, such as a single-CPU server with a very small cache. If either the Web Service client or the Web Service provider can only handle very low workloads, then it is unlikely that the ESB will become a bottleneck. This is the common misconfiguration that we observed in some performance verification labs as shown in Figure 3.

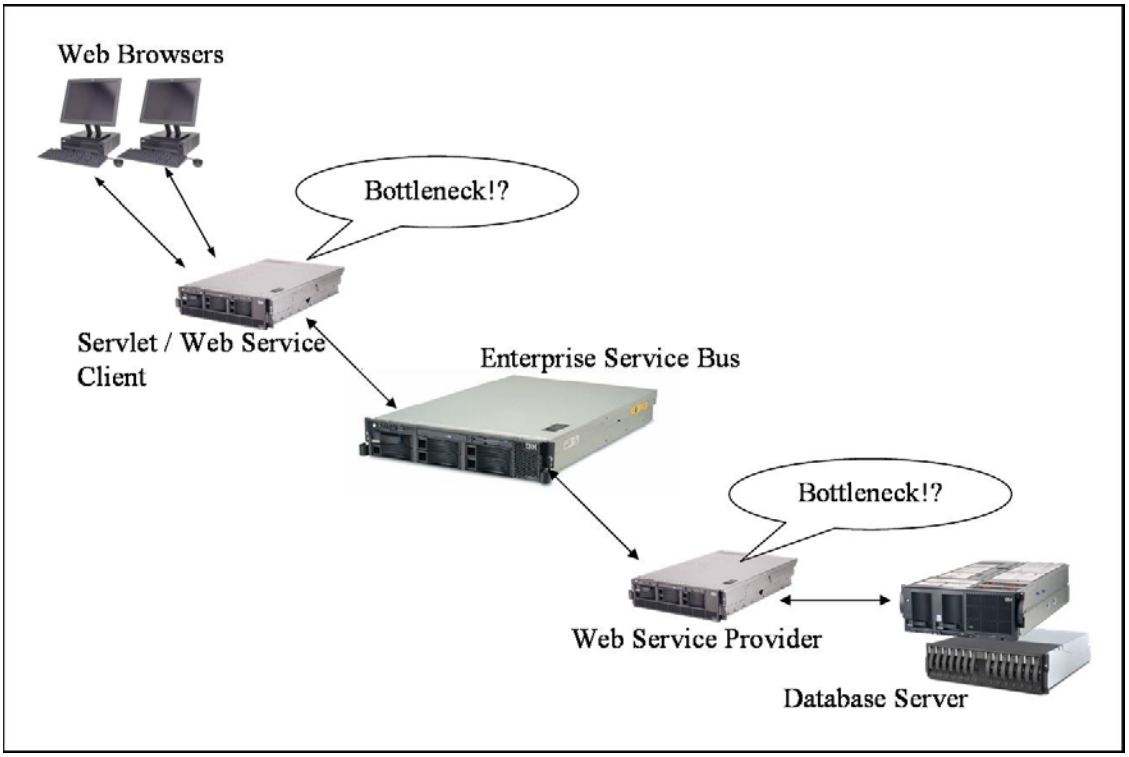

Figure 3. Bottlenecks in a misconfigured ESB test environment

Lightweight Mock Environment

With our approach we use an authentic ESB product and mediation modules on a realistic and large server. We configure the ESB middleware with a real WSDL [9]. Also, if necessary, we install mediation modules that can be used in the production system. Then we have an ESB server that is configured in a way very similar to the production system.

As we described before, in a real Web Service the clients are usually invoked by servlets instead of by the end users, who are typically using Web browsers. We need to remove any potential bottlenecks from the Web Service clients in the ESB performance evaluation environment. Therefore, we should use lightweight Web Service clients. With our approach, we use an HTTP workload simulator that emulates multiple virtual Web browsers [8]. Since the Web Service clients of this paper send SOAP messages using HTTP, an HTTP workload simulator is appropriate for creating the mock Web Service clients. The SOAP messages that the HTTP workload simulator sends should be the same as the messages that will be used in the production system. Then the ESB that we configure for the evaluation can handle those service requests as proper messages.

We also need to have a Web Service provider layer that can receive and respond to thousands of messages per second in order to load the ESB. We propose a method to create such an environment without setting up a real service provider environment such as a large server configuration. We call this a lightweight service provider. The lightweight service provider can receive messages from an ESB and respond with appropriate messages sent back to the ESB. Therefore the ESB cannot know whether or not the responses are from real service providers. The lightweight service provider is not a real Web Service provider. In the following section, we will discuss it in detail. The lightweight service provider may emulate delays such as those due to waiting for responses from backend database servers. Like the mock Web Service client we described earlier, the lightweight service provider should return the same SOAP messages that will be used in the production system. Therefore, in our ESB evaluation environment, the messages being passed around are the same as those the production system will use.

Configurable Parameters of a Lightweight Provider

Our lightweight mock provider supports the following configurable parameters in order to emulate the configurations of the production systems of service providers:
1). Multiple IP addresses with a single NIC
2). HTTP Keep Alive option
3). Number of threads
4). Response time within a mock provider

Since there will be multiple Web Services on multiple server nodes in a production environment, our mock provider should be able to handle several Web Services on a single system. The first parameter on the list will be used to support this.
The rest of the parameters will directly affect the results of the capacity tests, and the values set should be similar to those of a production environment. For example, the second parameter has several options, such as whether or not HTTP Keep Alive is allowed, the number of requests per connection using HTTP Keep Alive, and the maximum number of seconds to wait for the next request.

The third parameter controls the number of threads of the lightweight provider. With this option, we are able to control the minimum and maximum number of threads that can concurrently running on the provider to handle requests from an ESB. This will also affect the number of connections between the ESB and the mock providers.

The last option allows for emulating the response delay on the provider and/or the delays between the provider and the backend servers. In some cases, the response delays between the providers and backend database servers can affect the ESB performance, and it is possible to examine such situations with our approach.

Implementation of the Mock Environment

There are seven components that we needed to prepare for the implementation of an experimental proof of our approach: the WSDL, the ESB, Web Service clients, Web Service providers, the mediation modules, the request SOAP messages, and the response SOAP messages, as depicted in Figure 4.

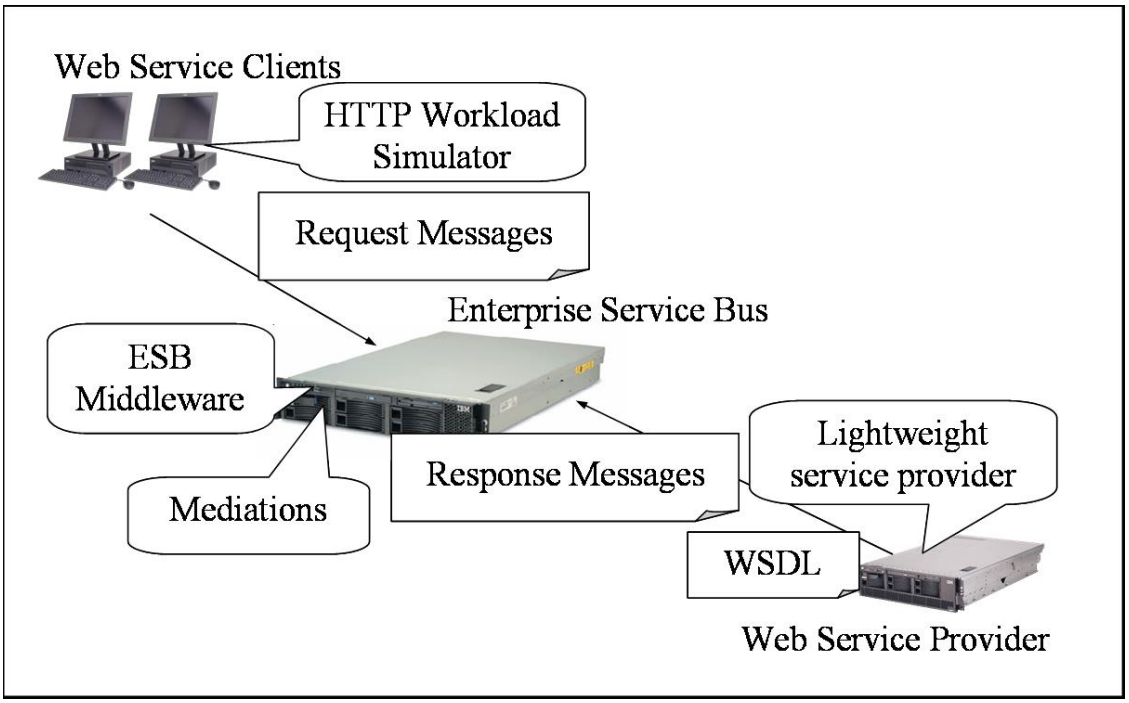

Figure 4. Components of mock environment for ESB capacity testing

The following paragraphs describe each component. In general, we need to import a WSDL file from the Web Service into the ESB to configure it properly. Therefore, we need to have an appropriate WSDL file for our experiment.

For the second component, the ESB itself, we prepared the hardware, middleware (ESB software), and the mediation modules that we will use in the production system to assess the performance capacity of the specific hardware used in the ESB system. Since we are doing capacity testing, it is necessary to have appropriate hardware for the ESB.

The third component, the Web Service clients, calls for preparing lightweight Web Service clients. With our approach, we use an HTTP workload simulator that emulates multiple virtual Web browsers. Since our Web Service clients send SOAP messages using HTTP, an HTTP workload simulator is appropriate for the mock Web Service clients. There are many commercial and public domain HTTP workload simulators. Since they have been used in many projects, they are quite mature and require very few resources. If needed, we could alternately use many small machines and easily avoid performance bottlenecks.

The next component is a lightweight Web Service provider. With our approach, we don't require a large server machine which hosts a lightweight provider. Instead, we use a small system. We also need to prepare a lightweight Web Service to emulate the actual service provider. There are several ways to implement this. The simplest one is to make a servlet that can receive SOAP requests from the ESB as shown in Figure 5. In the WSDL that we imported into the ESB, there is a line that points to a real Web Service endpoint, such as http://host:80/service/Banking. In this example, we simply deploy the lightweight Web Service servlet to link to the URI. In the lightweight Web Service servlet, we implement logic that responds to the SOAP response messages that were prepared in advance to be sent via the ESB. This can be implemented using the doPost() method. For instance, if you are using Apache Axis as your Web service engine, you can modify the org.apache.axis.transport.http.AxisServlet class to implement this idea. Of course we need to have a Web Service (J2EE) infrastructure to serve the lightweight Web Service servlet. The configurable parameters which we discussed earlier should be supported by either a Web Service infrastructure or a lightweight service provider.

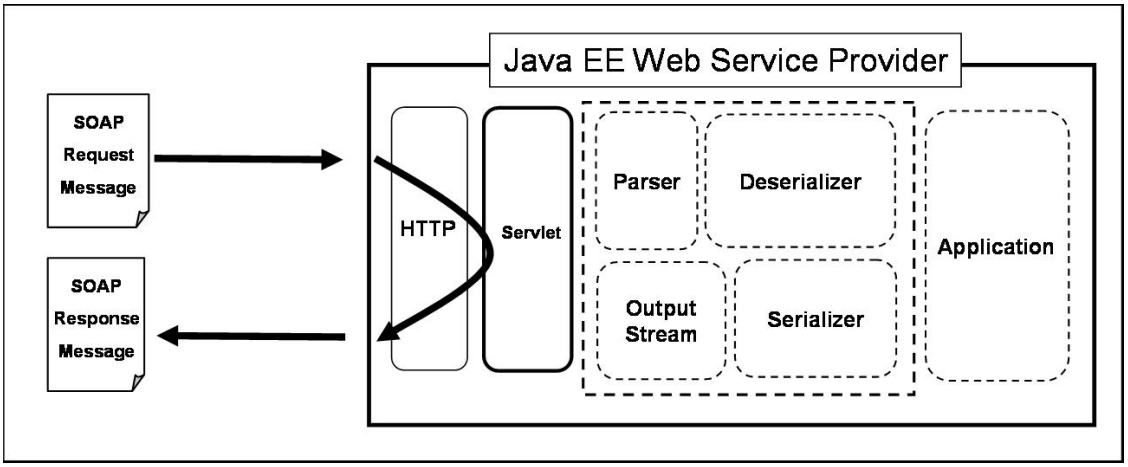

Figure 5. An Example of Lightweight Service Provider Implementation

The fifth type of components are request SOAP messages. The Web Service clients that form the HTTP workload simulator in our usage send these messages to the ESB using HTTP POST.

The last type of component consists of the response SOAP messages from the Web Service provider. In the lightweight Web Service servlet we described before, these messages are sent back to the ESB as response messages.

It is important to note that it should be very easy to set up this test environment and should be relatively simple to execute on a variety of different systems since we need to have this test environment in the very early phase of the project. To configure this system easy, we can use

several existing open source programs and/or commercial products for generating WSDL, SOAP request messages, and SOAP response message. With such software, you don't have to create Web Service client and server programs to generate SOAP messages. If you are already using these programs, you may simply capture their SOAP messages on the wire by using a message monitoring tool.

Experimental Results

We performed some experiments with the approach discussed in the last two sections. Based on the results, we determined that our approach is feasible. We explain our evaluation environment in the next few paragraphs.

Configuration

We used the following software and hardware for our experiments. For the Web Service client system, the ESB system, and the large Web Service provider system, an Intel Xeon MP 3.0 GHz 4-way server (4 MB L3 cache, 8 GB RAM) was used. For a small Web Service provider system, an Intel Pentium-M 1.7 GHz system (2 GB RAM) was used. We used gigabit Ethernet for a private network with a Cisco Catalyst 3750G 24T-E switch. For the authentic Web Service infrastructure and its ESB, IBM's middleware was used as the server software.

Through all the experiments, we used a realistic ESB system that is used in production environments. In this paper, we focus on evaluating the server-side mock environment. We used an HTTP workload simulator on a rather large system to emulate the Web Service clients. Thus there is no bottleneck in the Web Service client layer in our evaluation system. However, that does not mean that we needed a large machine for the client-side test environment. In fact, the client machine could have been replaced with a much smaller one if necessary.

We provide three ESB test scenarios:
- No ESB,
- ESB without mediation, and
- ESB with mediation.

The first scenario doesn't have any ESB. The second scenario uses an ESB with no mediation modules. Since there are no mediation modules, the ESB receives the messages from the Web Service clients and simply forwards them to the service providers. The third scenario uses the ESB with mediation modules. This mediation modules route requests to ports defined for the service destinations. The mediation modules will route requests to one of four destinations depending on a specific value in the SOAP header in the request message.

For each scenario, we tested three types of service provider environments (see Figure 6):
- Real Web Service providers on a small server (Naïve Test System)—representing a naively configured test environment,
- Lightweight Web Service providers on a small server (Lightweight Test System)—representing the lightweight test environment proposed in this paper, and
- Real Web Service providers on a large server (Production System)—representing the production environment.

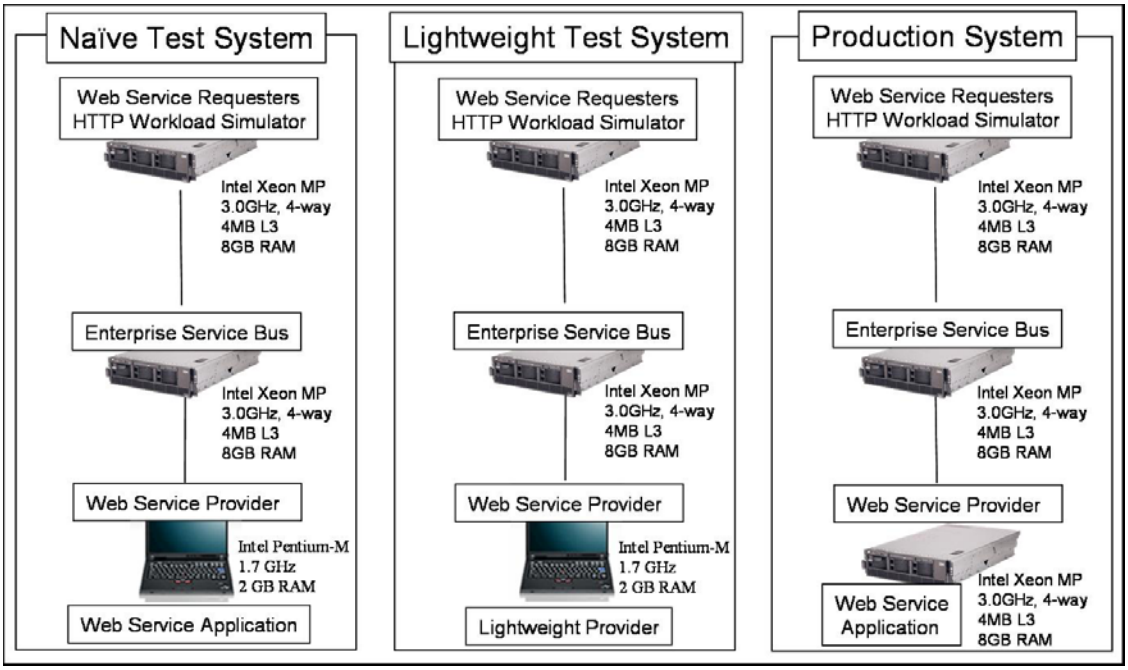

Figure 6. Performance Evaluation System Configuration

In the following Sections, we show the experimental results for each scenario: no ESB, ESB without mediation, and ESB with mediation, respectively.

Service Provider Performance Comparison

An HTTP workload simulator sent requests directly to these service providers and we used these numbers as our baseline shown in Figure 7. Note that in all cases, the server CPU usage is 100 percent and the client CPU usage is only 1 to 6 percent. The results of these measurements tell us the maximum capacity of the service providers.

The Production System served 4,901 requests per second. Since this transaction rate is high enough to emulate large Web Services environments in the real world, we decided to continue using a single SMP system as our high volume service provider.
Running as a Naïve Test System, it handled 513 requests per second, which is about 10% of the large server's throughput.

On the other hand, with our lightweight service provider, the small server recorded 2,028 requests per second. This shows that the lightweight service provider gave us a fourfold performance improvement.

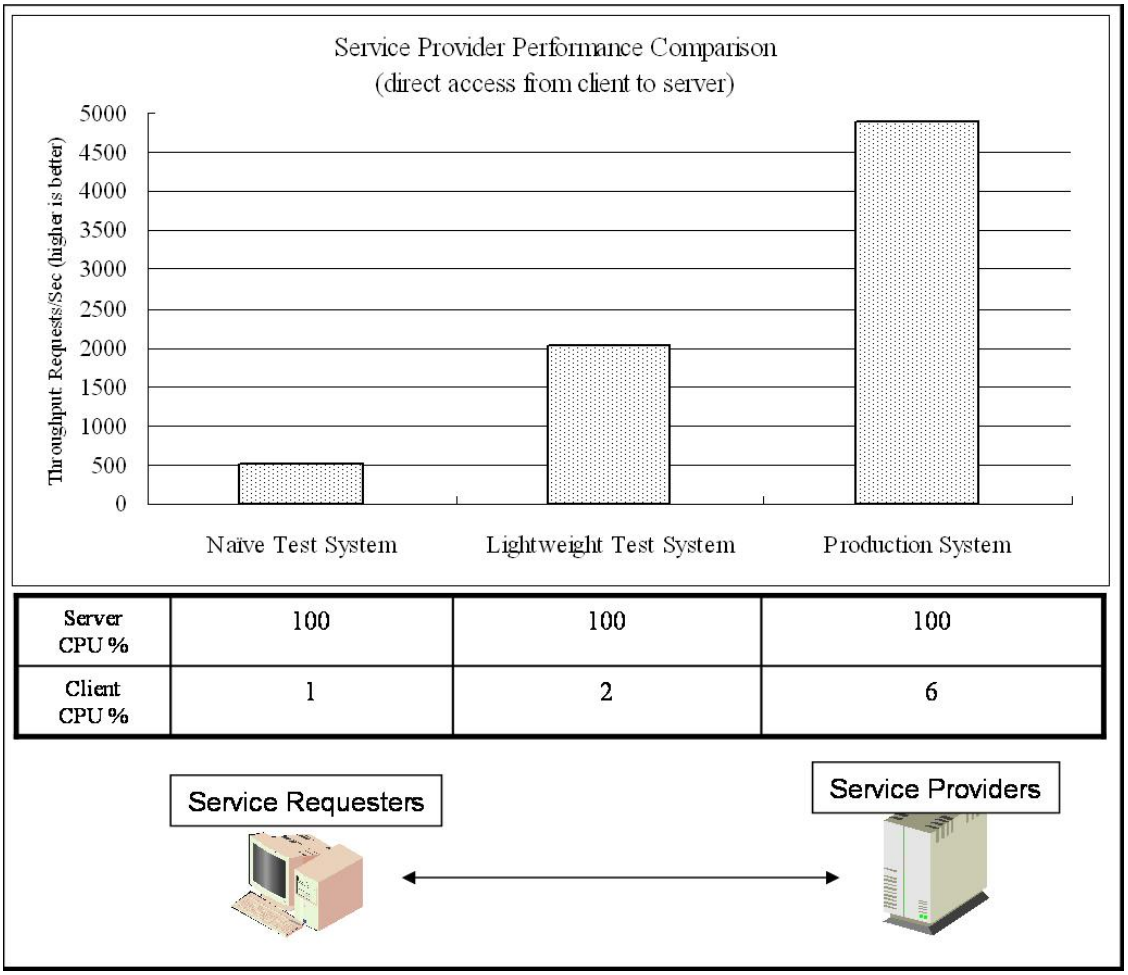

Figure 7. Service provider performance comparison

Note that in all three cases the client CPU usage was very low and it was not a bottleneck. Therefore, we will mention only ESB and Service Provider CPU usage in the following subsections.

ESB Performance Comparison

With the three different service provider environments, we measured the ESB performance without the mediation modules, as shown in Figure 8.

With the Naïve Test System, the maximum throughput was 475 requests per second. With the Lightweight Test System and with the Production System, the maximum throughputs are 1,260 and 1,272 requests per second, respectively. The performance difference between these two cases is less than one percent.

If we look at the CPU usage on the service providers, it shows that the Naïve Test System was completely overloaded, but the other two systems were not. In comparison, if you took a look at the ESB's CPU usage, the Naïve Test System was using 37% and the other two cases were saturating it at 100%.

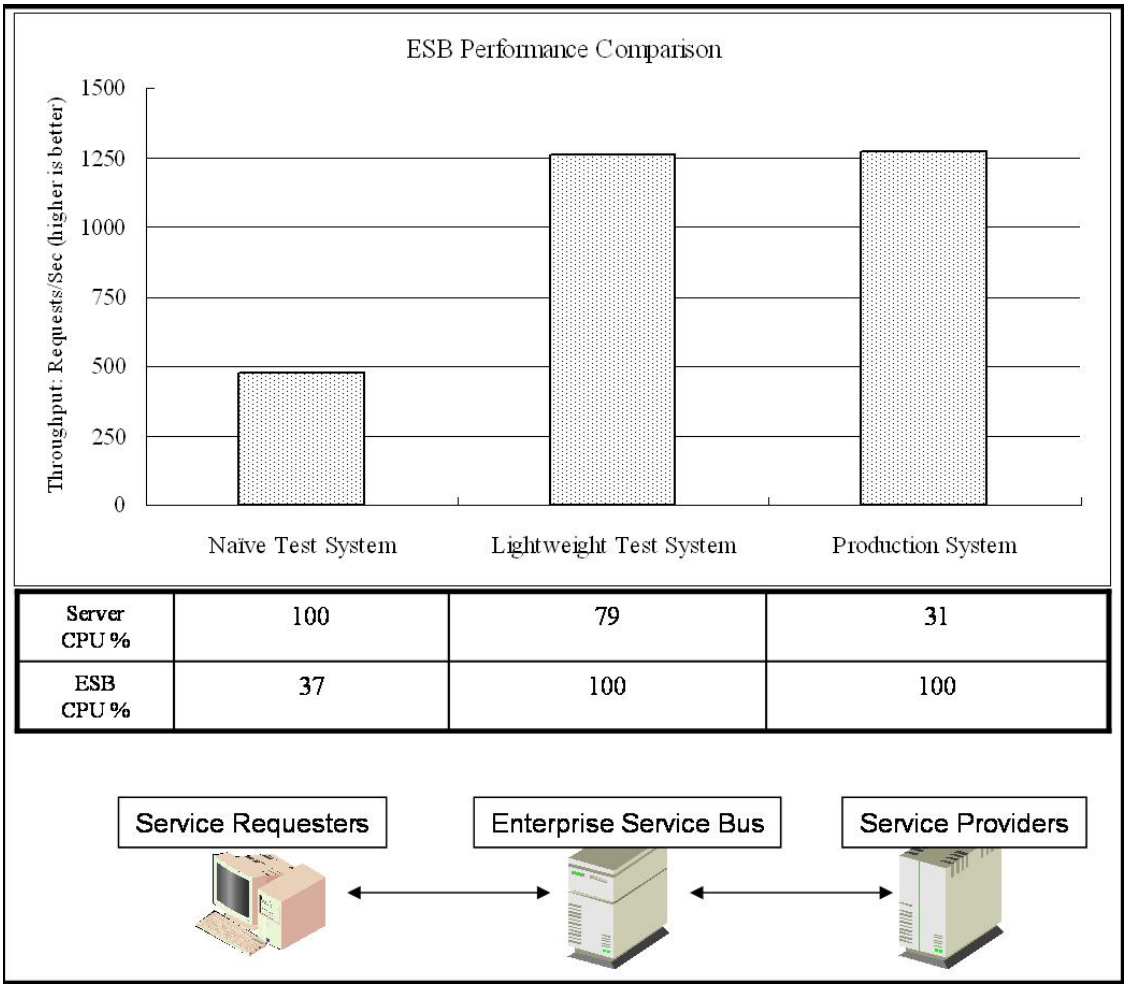

Figure 8. ESB performance comparison

ESB Performance Comparison with Mediations

With a port routing mediation module, we saw a similar trend as in the no-mediation scenario. With the Naïve Test System, the CPU usage of the server was 100% and the ESB's CPU usage was 63% which is not high enough to evaluate its maximum capacity. The throughput of this scenario was 397 requests per second. With the other two service provider scenarios, we completely maxed out the ESB system, as shown in Figure 9. With the Lightweight Test System, the small server's CPU usage was 51% and the large server's usage was 14% with the Production System.

The throughput of these cases was 601 requests per second and 610 requests per seconds, respectively. The performance difference of these two environments was only one percent.

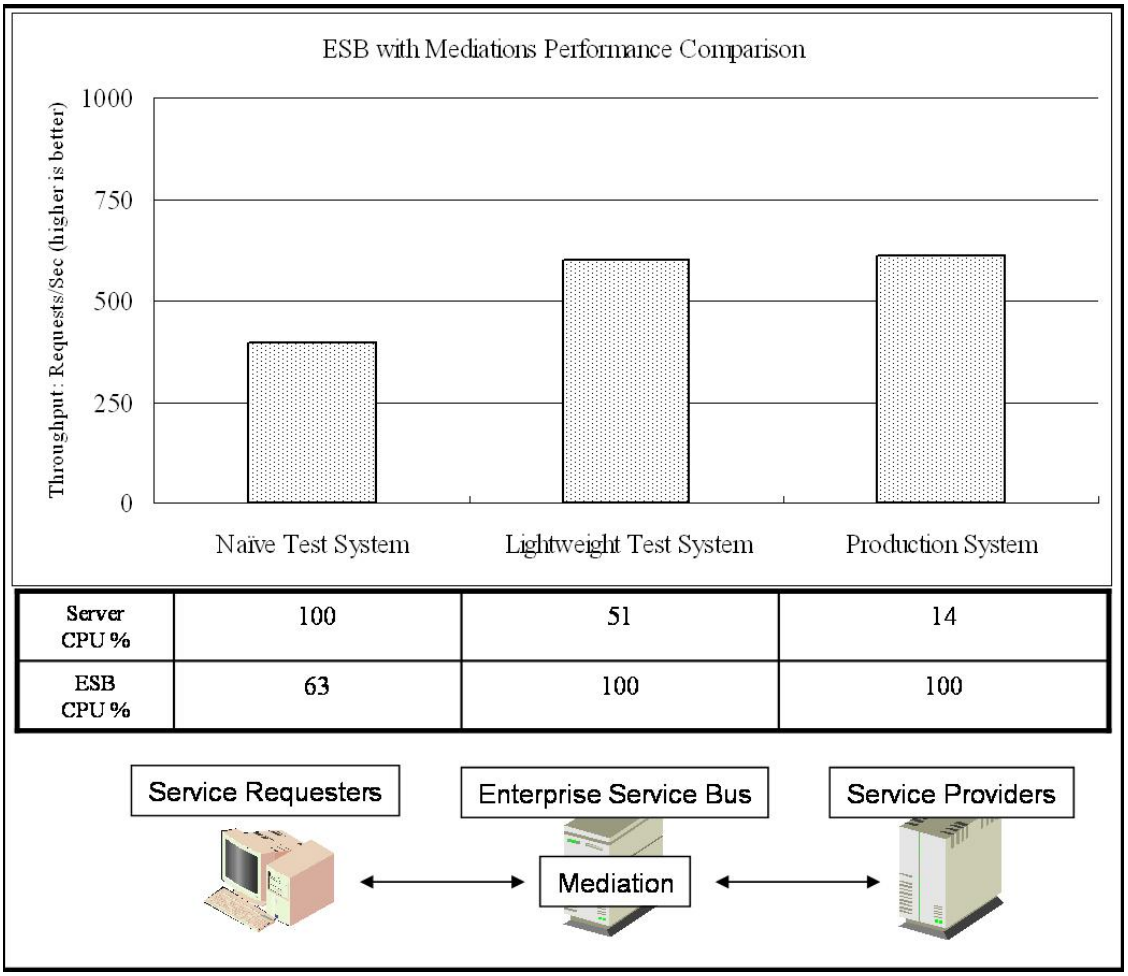

Figure 9. ESB with mediations performance comparison

Performance Analyses

In theory, the ESB CPU usage should be close to one hundred percent to find the maximum performance of the system. In addition, it is important that both the Web Service client layer and the Web Service provider layer should not be highly loaded.

With the small server (Naïve Test System), there is not much performance difference with and without the ESB (see Figures 7 and 8). These small-system results are misleading. Without further analysis of these results, a designer might believe that this ESB system would be large enough for a production system. That might be true if the service provider in the production system is relatively small, as in our single-processor system. In this scenario, the bottleneck is in the server due to the system resource limitation on computing power. Since there is a lack of CPU cycles in the service provider, the service provider cannot send messages back to the ESB quickly enough to saturate it.

With the large server (Production System), there is a huge performance gap between the cases with and without the ESB (see Figures 7 and 8). In this scenario, unlike the small server (Naïve

Test System) case, the ESB is a bottleneck. Since we have a powerful service provider system, the provider layer is no longer the bottleneck, but the ESB layer becomes the new chokepoint. With this scenario, we can measure the maximum capacity of this ESB system with a real Web server application on the service provider.

In the last test scenarios, which are using the lightweight provider on a small server (Lightweight Test System), we also see the performance gap between the tests with and without the ESB (see Figures 7 and 8). Due to the limited computing power of the small server, the throughput without the ESB is much smaller than for the large server with a real Web Service application scenario. This is understandable and an expected result. The key point is that the lightweight provider using the ESB can handle almost exactly the same throughput as the large server with a real application scenario as depicted in Figure 8 and 9. This indicates that our lightweight provider can effectively simulate the large server environment. Since both the real application on the large server using the ESB and the lightweight provider on the small server using the ESB can handle thousands of messages per second, the ESB was fully utilized and we determined the maximum throughput of the ESB.

Table 1. Response time comparison: Large server vs. Lightweight service provider

Scenario	Response Time (seconds/request)
Large server via ESB	0.039
Lightweight service provider via ESB	0.039
Large server via ESB with mediation	0.081
Lightweight service provider via ESB with mediation	0.082

We also captured response times during the capacity testing. As shown in Table 1, the response times with the lightweight service provider and the large server are very similar, and this provides additional evidence of the validity of our approach.

Related Work

In this section, we discuss the previous research on performance testing of distributed systems. As we have already discussed the alternative approaches for estimating capacity of ESBs in the introduction section, we here focus on actual testing with real machines for system under test. Specifically, we discuss the existing performance testing technologies, which can be used for capacity testing purposes. As there are variety of techniques and products for testing performance of distributed systems, we can discuss only some representatives.

In the Grid computing research area, several performance testing tools have been proposed since the performance is one of the most critical issues addressed in the area. In a Grid environment, the research issues arise from the situation of large numbers of variety of nodes in the environment. The issues addressed are, for example, clock synchronization for a large number of testing machines to control performance estimation accuracy, heterogeneity of WAN environments, and coordination of a large amount of resources [19]. However, researchers have not considered the lack of resources for testing as a problem in this area.

For the performance testing of web application servers, a lot of tools are available as products and are widely used by practitioners. For example, Mercury's Road Runner [20], Compuware's QALoad [21], Microsoft's Application Center Test [22], and IBM's Rational Performance Tester

[23] are the commercial products most widely used for Web application development. Their features are mainly differentiated by flexibility in customization of possible load scenarios and usability supported by the automation of testing. Unfortunately, their functionality is basically limited to load generation since their testing target is Web servers. In fact, their functionality can be regarded as a subset of the proposed testing environment in this paper.

Emulating the backend servers is a natural approach to setting up a testing environment. An example of an application of this approach can be found in the SPECweb2005 benchmark environment [24]. In the SPECweb2005 benchmark kit, a back-end simulator called BeSim is provided to set up the testing environment behind the Web server under test. BeSim is intended to emulate a back-end application server that the Web server must communicate with in order to retrieve specific information needed to complete an HTTP response (customer data, for example). BeSim exists in order to emulate this type of communication between a Web server and a back-end server.

The mock server used in the proposed testing approach is topologically the same as the BeSim server. However, since BeSim is designed for SPECweb2005, the messages to and from BeSim are predefined for this specific benchmark. Unlike the BeSim, our proposed mock server can receive and replay any messages that are required for your ESB capacity testing.

Discussion

As mentioned, there are several kinds of mediation modules that we can run on an ESB. In previous sections, we use routing mediation as an example. There are a few mediation modules that might not work with our approach. For example, if mediation requires external resources such as database access, we need to have another system that emulates the database.
We focused on Web Services, particularly SOAP over HTTP. For future work, we will study SOAP over JMS and other protocols.

In this paper, we focused on an ESB which is one of the most important components for the SOA infrastructure. However, our lightweight service provider works not only with an ESB but also with the BPEL engine which is another important component for the SOA infrastructure as depicted in Figure 10. The BPEL engine is a runtime which hosts business processes written in the Business Process Execution Language (BPEL) [25] and for BPEL business processes; in particular, the BPEL engine orchestrates the Web services. Since Service Requesters talk to the BPEL engine instead of Service Providers, Service Requesters doesn't know about the response messages from Service Providers to the BPEL engine. From this point of view, our approach for early capacity testing can be applied to the BPEL engine with our lightweight service provider.

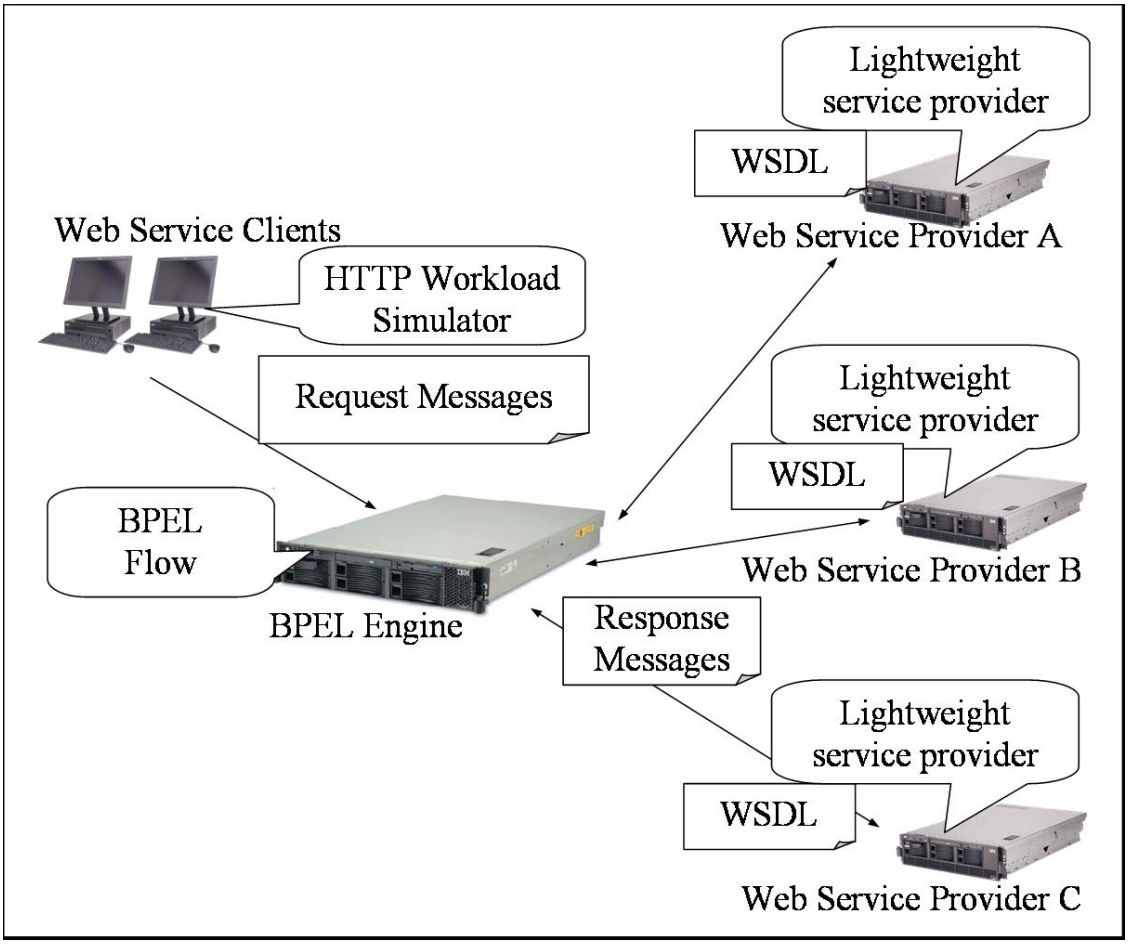

Figure 10. Mock environment for BPEL Engine capacity testing

Concluding Remarks

In this paper, we proposed a capacity testing method for an ESB that can be used during the phase of capacity planning. With our proposed capacity testing, designers can assess the ESB server capacity using a very small hardware environment by using lightweight service providers and an HTTP workload simulator as lightweight Web Service clients. Unlike most other capacity planning methods, the results of this capacity testing can reveal the actual maximum capacity of an ESB server on a specific platform.

ACKNOWLEGMENT

The authors would like to thank the member of the IBM Tokyo Research Laboratory including Yuichi Nakamura, Toshiro Takase and Naohiko Uramoto for their comments on an earlier draft of this paper.

REFERENCES

[1] Alonso, G. & Casati, F. (2005) Web Services and Service-Oriented Architectures. International Conference on Data Engineering, 1147.

[2] Avritzer, A. & Weyuker, E.J. (2004). The Role of Modeling in the Performance Testing of E-Commerce Applications. IEEE Trans. Software Eng., 30(12), 1072-1083.

[3] Avritzer, A., Kondek, J., Liu, D., & Weyuker, E.J. (2002). Software performance testing based on workload characterization. Workshop on Software and Performance, 17–24.

[4] Balsamo, S., Marco, A.D., Inverardi, P., & Simeoni, M. (2004). Model-Based Performance Prediction in Software Development: A survey. IEEE Trans. Software Eng., 30(5), 295-310.

[5] Cecchet, E., Marguerite, J., & Zwaenepoel, W. (2002). Performance and Scalability of EJB Applications. Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 246–261.

[6] Denaro, G., Polini, A., & Emmerich, W. (2004). Early Performance Testing of Distributed Software Applications. Proceedings of the Fourth International Workshop on Software and Performance, 94–103.

[7] Dikaiakos, M.D. (2004). Intermediary infrastructures for the World Wide Web. Computer Networks, 45(4), 421-447.

[8] Joines, S., Willenborg, R., & Hygh, K. (2002). Performance Analysis for Java Websites. Massachusetts: Addison-Wesley Longman Publishing Co., Inc.

[9] Frank Leymann. (2005). The (Service) Bus: Services Penetrate Everyday Life, ICSOC 2005, 12-20.

[10] Litoiu, M. (2004). Migrating to Web services: a performance engineering approach. Journal of Software Maintenance and Evolution: Research and Practice, 16(1-2), 51-70.

[11] Litoiu, M., Krishnamurthy, D., & Rolia, J.A. (2002). Performance Stress Vectors and Capacity Planning for E-Commerce Applications. International Journal on Digital Libraries, 3(4), Springer, 309-315.

[12] Luo, M., Goldshlager, B., & Zhang, L. (2005). Designing and implementing Enterprise Service Bus (ESB) and SOA solutions. Tutorial, IEEE International Conference on Services Computing, xiv.

[13] Menasce, D.A., & Almeida, V. (2001). Capacity Planning for Web Services: metrics, models, and methods. New Jersey: Prentice Hall PTR.

[14] Mos, A. & Murphy, J. (2002). Performance Management in Component-Oriented Systems Using a Model Driven Architecture Approach. Proceedings of the 6th International Enterprise Distributed Object Computing Conference, IEEE Computer Society, 227–237.

[15] Patrick, P. (2005). Impact of SOA on enterprise information architectures. Proceedings of the 2005 ACM SIGMOD international conference on Management of Data, 844–848.

[16] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., & Ferguson, D.F. (2005). Web Services Platform Architecture. New Jersey: Prentice Hall.

[17] Weyuker, E.J., & Avritzer, A. (2002). A metric for predicting the performance of an application. under a growing workload. IBM Systems Journal, 41(1), 45-54.

[18] Weyuker, E.J., & Vokolos, F.I. (2000). Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. IEEE Trans. Software Eng., 26(12), 1147-1156.

[19] Raicu, I, Dumitrescu, C., Ripeanu, M., Foster, I. (2006). The Design, Performance, and Use of DiPerF: An automated Distributed PERformance evaluation Framework, Journal of Grid Computing, 4(3), Springer, 287-309.

[20] Mercury Load Runner, http://www.mercury.com/

[21] Compuware QALoad, http://www.compuware.com/

[22] Microsoft Application Center Test, http://www.microsoft.com/

[23] IBM Rational Performance Tester, http://www.ibm.com/

[24] Standard Performance Evaluation Corporation (SPEC), http://www.spec.org/

[25] Andrews, T., Cubera, F., Dholakia, Hi., et al. (2003). Business Process Execution Language for Web Services version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

ABOUT THE AUTHOR

Ken Ueno is a member of Service Oriented Computing Group in IBM Tokyo Research Laboratory. He has been working on the area of J2EE server performance over 7 years. Before joining IBM Tokyo Research Laboratory, he worked for the WebSphere Performance Group in the WebSphere Development Team in the U.S. He has published several books including the WebSphere V3.5 Handbook... His recent research area is an SOA infrastructure performance.

Michiaki Tatsubori is a research staff member at IBM Research currently working in the Core Runtime Infrastructure Group of the Tokyo Research Laboratory. His research focuses primarily on the area of programming languages and software engineering with distributed computing and enterprise computing as its major applications. He led the Java Zone in the IBM developerWorks Japan and has published several books on Java, XML, and Aspect-Oriented Programming.