Enhancing Enterprise Service Bus Capability for Load Balancing

AIMRUDEE JONGTAVEESATAPORN, SHINGO TAKADA
School of Science for Open and Environmental Systems
Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa 223-8522
JAPAN
aimrudee@doi.ics.keio.ac.jp, michigan@doi.ics.keio.ac.jp

Abstract: - ESB is a core middleware technology which can support the integration of services according to the Service Oriented Architecture. A major responsibility of ESB is to route messages to heterogeneous services. However, conventional ESBs support only static routing, i.e. the service to which a message is sent must be fixed a priori. Thus, even if there are many services that can satisfy the same request, the request is always sent to the same service without considering the service status, e.g., load, at that time. This situation may lead to a low throughput performance on the service side and low satisfaction on the consumer side. This paper aims to enhance the ESB capability by supporting load balancing. Our approach focuses on balancing among a group of different services with the same function. We introduce the concept of service type and show the results of an experiment.

Key-Words: - ESB, Middleware Message Balancing, Web Services, Load Balancing, SOA

1 Introduction

Service Oriented Architecture (SOA) is an architectural design pattern in which the concept of a "service" is an abstraction of a function used by an application. SOA logically decouples the service requester from the service provider by isolating the service definition from a service implementation [1] [2]. One enabling technology for SOA is the enterprise service bus (ESB). ESB is an important middleware tool for integrating services based on various platforms. In Kambhampaty's proposed architecture for developing enterprise-wide SOA [3], ESB is used to enable a smooth communication between applications. In Panian's work [4], ESB is required to implement SOA, where the service implementations can plug in and out, and which supports multiple calling semantics (e.g. synchronous and asynchronous) and features (e.g., transformation and routing).

One of the main capabilities of ESB is the routing of messages among different services. Current ESB implementations support several message routing patterns, but can execute based only on static configuration [5]. When a client sends a message, that message will be dispatched to a specified service, or endpoint, regardless of the status of that service. At that moment, the service may be unavailable or busy with many messages and thus cannot immediately process the incoming request. In the case of the service being busy, the message is put into a queue and will be processed at a later time. In such a case, it will be troublesome if the request has a processing deadline, which cannot be met. Since the service is fixed, the service consumer cannot access other services that provide the same function. If the number of requests increases dramatically, it is better to distribute the requests to other existing services that can satisfy the same requests. For example, Mule [6] and ServiceMix [7] are both ESB implementations that support load balancing, but the target services (specifically, endpoints) must be set in a configuration file a priori and cannot be changed at runtime. We need a way to solve this issue of "too many" requests.

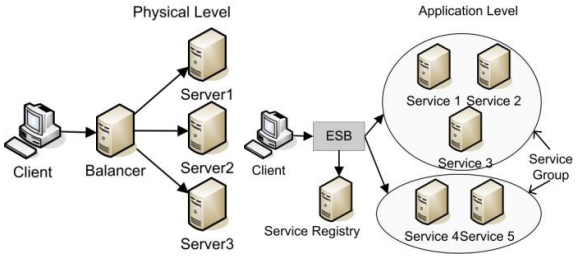
Many Web sites take a load balancing approach to handle this issue of "too many" requests. The simplest approach is for a hostname to have multiple IP addresses. This is the case with google.com, where the actual physical server one accesses will differ depending on the load at that time [8]. This same approach can also be taken in SOA if the service provider replicates the service onto multiple servers resulting in multiple physical services (Fig. 1 (a)). Thus, conventional load balancing approach can be taken to satisfy the issue of "too many" requests.

We take a different approach. Instead of replicating a service, we group different services having the same function. This is based on the premise that there are multiple similar services that

can be used to increase dependability [9]. In other words, there may be multiple services that can perform a given function that the user wants [10]. If an error or no response is received after a certain length of time, we can switch to an alternative service.

In order to enable grouping of services, we introduce the concept of service type. Services belonging to the same service type have the same function and same signature.

We also incorporate a load balancing feature into an ESB implementation, specifically Mule. Our load balancing mechanism dynamically selects the actual target service at runtime based on a specified service type (Fig. 1 (b)), using strategies such as random, round-robin, threshold, minimum, and least load [11]. What must be decided in advance is the service type and not the actual service. Thus, this has the added advantage to cope with the situation where services belonging to the same service type may change during runtime, i.e., a new service may be added to a service type.



(a) Conventional balancing (b) Our approach

Fig. 1 Load balancing

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes our concept of service type. Section 4 then presents our balancing ESB mechanism along with an overview of how it is implemented. Section 5 describes the experiment we conducted. Section 6 makes concluding remarks.

2 Related Work

Much research has been done recently on ESB for supporting SOA integration. We briefly describe three types: (1) dynamic service selection, (2) load balancing, and (3) service substitution.

2.1 Dynamic service selection

The ASB project [12] proposed an adaptable service bus that supports changes to business rules at runtime, thus avoiding costly shutdowns to applications. The service router component in ASB is responsible for selecting the service to be used at runtime. The target service selection is based on a ranking which uses information such as execution time and expected availability.

The DRESR project [5] allows the routing table to be changed at runtime. DRESR defines the Abstract Routing Path (ARP) using abstract service names, which are instantiated at runtime by replacing the abstract service names with the real URIs. DRESR supports the specification of service selection preferences such as response time.

B. Wu et al [13] proposed a method for dynamic reliable service routing. They add the context of application information related to the request message for use in service discovery. Their approach accepts a routing target list dynamically, so that the routing can change at run time. If the request does not respond within a suitable time, the ESB will resend the request to another service.

All three above approaches consider dynamic service selection, but they do not apply their approaches to message balancing.

2.2 Load balancing

The Cygnus and TAO projects [11, 14] incorporate balancing, which adapts to different load condition, to CORBA [15]. Their approach uses object groups each of which contains duplicates of a particular object. Thus, their approach cannot support heterogeneous services. Furthermore, some of their balancing strategies use load migration, but migration delay may cause problems.

Wang et al [16] and Roca et al [17] tried to avoid unnecessary load migration. Wang et al [16] proposed a load balancing middleware for service-oriented application. It collects a service group from resources that are registered in a service replica repository, and adds a load agent into the server side for providing load information. They balanced resource allocation among different services which is similar to our work but their approach uses machine-learning to predict loading.

Roca et al [17] used a local load balancing strategy, specifically nearest neighbor algorithm. Their technique is based on computing the average workload of nodes forming a neighborhood or domain. Migration is done when certain conditions are reached.

Karrio et al [18] focused on clusters of servers, and introduced a QoS aware load balancing algorithm (QoS-LB). The servers in each cluster have identical or nearly identical content, and need to be fixed a priori.

Fernandez et al [19] introduced Semantic Web technology to enable load balancing between multimedia servers. A client uses ontology information to determine which servers they can use. Once a session has started between a client and server, the QoS is monitored. If the QoS decreases too much, then it will be redirected to another server transparently. Our work can also switch to a new service implicitly when the requested message cannot reach the target endpoint.

2.3 Service substitution

Taher et al [20] proposed the concept of abstract Web service (A.WS) and concrete Web service (C.WS) for Web services substitution. Each A.WS is classified into a category and links to a list of similar concrete Web services. Our approach is similar to this structure but we added service type property such as QoS for advanced filtering of services.

Pianwattanaphon et al [21] used the service type ontology to describe the capability of Web services such as signature, behavior for invoking a substitute Web service in the case of invocation failure. However, we are not interested in semantic matching.

3 Service Type

Service information is normally stored in registries, such as UDDI [17]. In UDDI, business information (e.g., business name, contacts) and service information (e.g., service name, access point) are registered. These standard attributes in UDDI are not enough for our purpose. We thus propose "service type".

3.1 Service type

In order to enable the dynamic selection of Web services, we propose the concept of "service type" which is used to group Web services with the same function that can satisfy the same request.

Each service belongs to a service type. A type has the following information:

- Service Type Name: Each service type has a unique name.
- Service Signature: The signature consists of the input parameter(s) and return type.
- Service Property: A property is optional information, such as QoS attribute, which can be used when searching for a suitable service. Note that unlike the signature, this is optional, and services that do not provide property information may be included in the same type.

Table 1 shows examples of service types. For example, the MoviePreview type takes a string as an input parameter, and returns a MediaFile. There is one property "availability".

The basic idea of incorporating a service type is that services of the same service type may be substituted with each other. For example, in Fig. 2, Web Service #1 and Web Service #3 both have the same service type A. Since this means that they have the same functionality, they can be substituted with each other; if Web Service #1 is unavailable, then Web Service #3 can be called.

Table 1 Examples of service types.

Service Type Name	Service Signature		Service Type Property
	Parameter	Return Data Type	
Hotel Reservation	Location: String Room:Int	Boolean	Availability
Flight Information	Departure:String Arrive:String Date:Date	List	ExecutionTime
Restaurant Search	City:String	List	Accessibility
Calculator	1st Num: Double 2ndNum: Double Operation: String	Double	ExecutionTime
Money Exchange	1stCurrency:String 2ndCurrency:String Value:Double	Double	ExecutionTime
Document Printer	Document: File	Boolean	Availability
Photo Sharing	Picture:File	Boolean	Security
Online Radio	RadioName: String	MediaFile	Availability
Movie Preview	MovieName: String	MediaFile	Availability

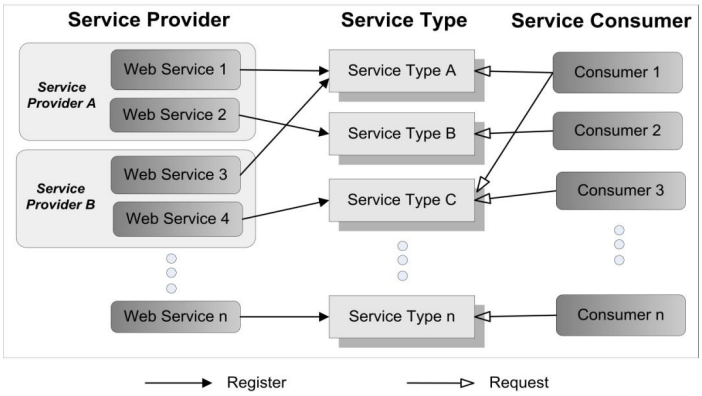


Fig. 2 Sharing service type

3.2 Incorporating service type

The service provider defines the service information, including the service type, and stores it into a service registry. Although the provider can use any name, it is recommended that when possible, a type that is already included in the repository should be used. In other words, when a service provider stores service information in the service registry, he/she first searches for a service type that matches the one they have built. If there are no matching service types, the provider can define a new service type.

If a property is defined for a service type, then all services belonging to that type must have that information. On the other hand, the actual services may specify property information that is not defined at the service type level.

Note that a service type with many services likely indicates that (1) the service type can be considered to be important because multiple service providers provide the basic functionality, and (2) the chance of load balancing increases.

4 Balancing mechanism

The basic idea of our mechanism is that, given a service type, we send a message to the most suitable service (belonging to the service type) based on the specified balancing strategy.

The rest of this section gives details of our balancing mechanism, which we have implemented using Mule ESB.

4.1 Mechanism components

We describe each component in our mechanism below (Fig. 3):

- *Inbound router* is provided by Mule, and it receives messages from a channel. We currently use JMS [22] channel. When a client sends a message, the message is stored in a request queue of ActiveMQ 5.2.0 [23] which is an open source JMS.
- *Message extractor* is a module for extracting the contents of the message and obtaining important values such as service type, service type property and request values. Note that this component is important, as the service type is included in the header. It is implemented using JDOM (Java Document Object Model) [24], which enables efficient manipulation of XML data in Java form.
- *Service group recognizer* receives the service type data from message extractor and then sends this data to the service registry for discovering the services belonging to this type. This results in a list of endpoints, which is sent to the balancing computing module. The service group recognizer also has the responsibility of filtering services if property information is available.
- *Service registry* is integrated with ESB for supporting dynamic service selection. Dynamic selection requires a list of services, each of which can satisfy the same request. Our current implementation is based on UDDI; we added an extra attribute for service type. Service type can be used to query a list of services that belong to that type.
- *Balance computing module* is the component for managing the sending of messages. The actual destination of a message is decided using a balancing strategy. The balancing strategy is set by the ESB administrator before running Mule. There are currently five strategies implemented (section 4.5). This module connects to the load monitor module for getting load information, and uses the obtained data to check which service application has the least amount of load at that time.

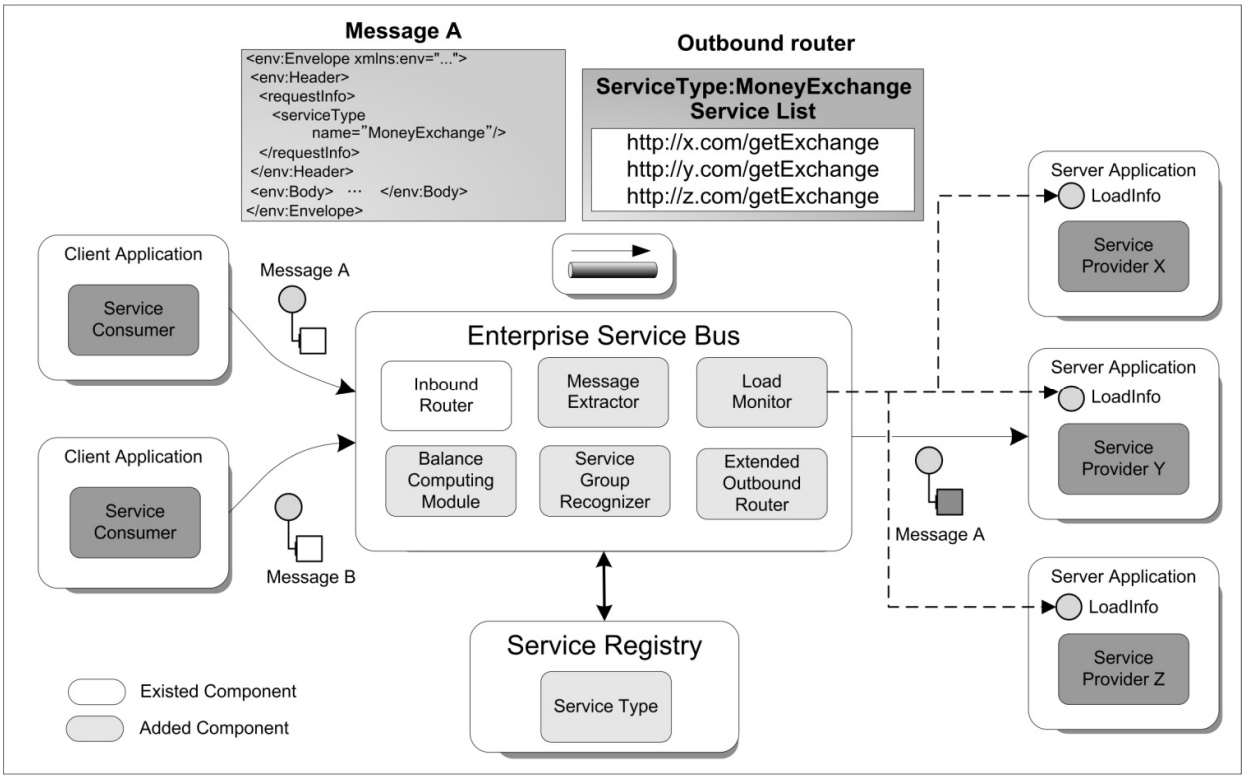


Fig. 3 ESB enhanced with balancing mechanism

- *Load info* is information concerning the load on the service. It should be updated by the service provider frequently. In our current implementation, it is calculated by the number of completed process messages subtracted from the number of incoming messages into the service.
- *Load monitor* is the module that connects to the service provider to obtain load info. We can configure a time interval for updating the load info data.
- *Extended outbound router* is a component that is extended from the standard Mule outbound router. The endpoint can be set at runtime for dispatching messages. If the outbound router catches an exception because the system cannot connect to the target endpoint as shown in Fig. 4, another service from the same service type is chosen. The ESB then resends the request to this service, and sends a signal to the service registry to temporarily block the broken endpoint. Meanwhile, the ESB will send a heartbeat to check if the service becomes "alive" again. If the service recovers, the ESB adds the endpoint back to the service list.

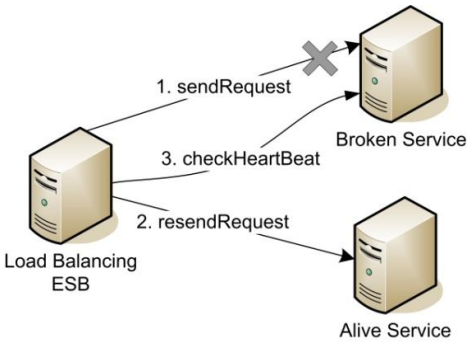


Fig. 4 Alive service checking

4.2 Balancing procedure

The steps in balancing are given below (Fig. 5):

1. A client sends a message to invoke a service. The actual service is not specified, rather the service type is specified in the header of the message. Service properties such as availability and execution time can also be attached in the header for use in filtering candidate services.
2. The inbound router of ESB catches the incoming request message, and forwards it to the message extractor component.

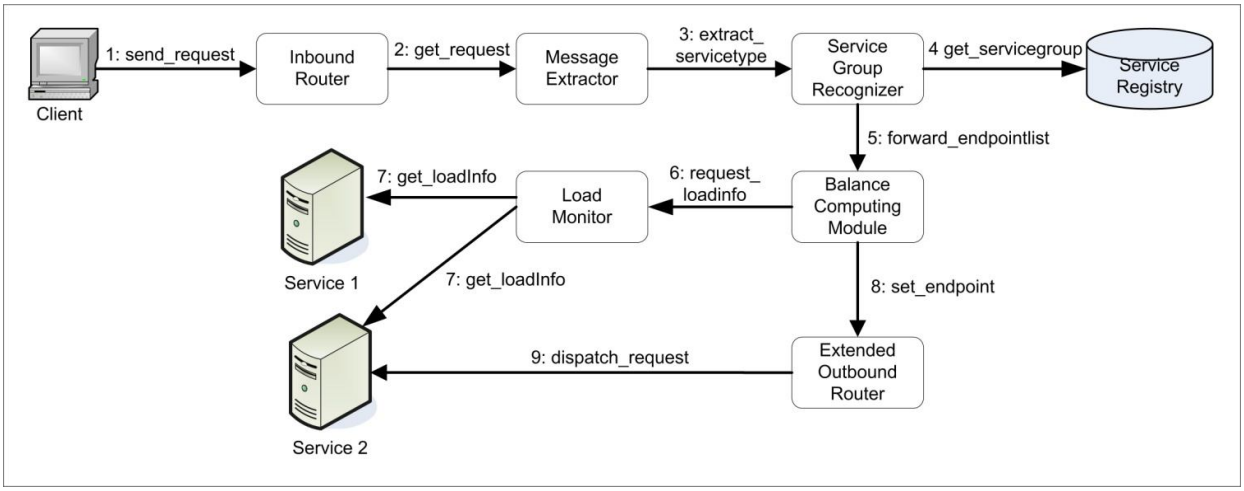


Fig. 5 Interactions between components

3. The message extractor extracts the service type value from the message header, and then sends it to the service group recognizer.
4. The service group recognizer queries the service registry using the service type value as a query parameter. The service registry returns a list of services belonging to the same service type to the service group recognizer.
5. If the service type in the message header contained properties, then when possible, the service group recognizer filters the list of services. Then, the service group recognizer obtains the endpoint of each service in the list, and sends them to the balancing module.
6. The balancing module requests the load information (loadInfo) from the load monitor module.
7. The load monitor module asks for the current loadInfo from the service providers. The service providers return the current load information to the load monitor. Then, the load monitor forwards the load information to the balancing module.
8. The balance computing module determines the target service using the balancing strategy, and forwards the endpoint to the extended outbound router.
9. The extended outbound router sends the message to the actual destination service.

4.3 Message header

In the previous section, we described how our mechanism works. The service type information is encoded in the header of a request message as shown in Fig. 6.

```
<env:Envelope xmlns:env="…">
 <!--Header part -->
 <env:Header>
    <requestInfo>
       <serviceType>
         <name>
            FlightInformation                    ... (1)
         </name>
         <property>
          <executionTime>
            <value>10ms</value>                  ... (2)
            <evaluation>LT</evaluation>
          </executionTime>
         </property>
       </serviceType>
     <requestInfo>
  </env:Header>
  <!--Body part -->
  <env:Body>
     <messageInfo>
       <departure>tokyo<departure>               ... (3)
       <arrive>boston</arrive>
       <date>27</date>
       <month>01</month>
       <year>2010</year>
     <messageInfo>
   </env:Body>
</ env:Envelope >
```

Fig. 6 Message example

Table 2. Evaluation attribute value in service property type

Value	Symbol	Meaning
EQ	=	equal
LT	<	less than
GT	>	greater than
LE	<=	less than or equal
GE	>=	greater than or equal
MIN	no symbol	minimum
MAX	no symbol	maximum

```
<mule xmlns-="…">
<jms:activemq-connector name="localhost"
       brokerURL="tcp://localhost:61616" /> … (1)
 <model>
  <service name="BalancingESBProject">
    <inbound>
      <jms:inbound-endpoint queue="request.queue"
                    synchronous="false"/>   … (2)
    </inbound>
    <component class="org.my.balancer"/>    … (3)
    <outbound >
      <custom-outbound-router class
            ="ExtendedOutboundRouter" >
      <outbound-endpoint address=" " />       … (4)
      <reply-to address="jms://receive.queue"/>
                                              …(5)
     </custom-outbound-router>
    </outbound>
   </service>
  </model>
<mule>
```

Fig 8. Mule configuration

In the header part, the serviceType name attribute is declared in the serviceType element, e.g. FlightInformation (1) in Fig. 6. A service consumer can add a serviceType property value (2) to filter candidate services after the message extractor component receives the service list from the service registry. This property is optional, so the service consumer can send the request with or without the property information. Only services that match the consumer specified properties will be returned. In Fig. 6, the service consumer requests the FlightInformation service and asks for an execution time of less than 10 milliseconds. Table 2 shows a list of evaluation value meaning.

In the body part of the message, the messageInfo element is the tag for collecting the parameter values to invoke the FlightInformation service such as departure place, arrival place and the travel date (3).

4.4 Mule configuration

Fig. 8 shows a basic template for the Mule configuration file using our approach. First of all, we must configure the destination of JMS connection (1). Then, we set the inbound router to connect to the JMS queue named "request.queue" for getting the incoming message (2). The incoming messages are processed by the balancer component (3). This corresponds to the balance computing module in Fig. 3. The target service is left blank because the ESB extracts "ServiceType" from the header of request messages and sets up the service endpoint later (4). Since this is an asynchronous message, we set the reply destination (5).

4.5 Balancing strategies

The selection of the actual destination service depends on the balancing strategy. The following strategies are available in our current implementation:

- Round-Robin: This strategy keeps an endpoint list of a given service type containing at least one endpoint, and selects an endpoint iteratively through the service list.
- Random: This strategy randomly chooses an endpoint from an endpoint list.
- Threshold: This strategy allows a service to continue receiving requests until a threshold value is reached. Once the threshold value is reached, subsequent requests are sent to another endpoint with the same service type. The next endpoint is chosen based on round-robin strategy. This next endpoint will be used until its threshold value is reached. Then the third endpoint is selected based on round-robin strategy, etc. If all services are over the threshold, then the round-robin strategy is employed for each message.
- Minimum: This strategy selects the service which has the least number of messages in the message queue.
- LeastLoad: This strategy is similar to threshold; it allows a service to continue receiving requests until a threshold value is reached. However it is different from threshold strategy in that once the threshold value is reached, subsequent requests are sent to the service with the least load. If all

services are over the threshold, then the system will send the message to the service server with the least load even though it has reached the threshold limit.

5 Evaluation

This section first describes an experiment that evaluates the performance, specifically response time, of our load balancing ESB. We then describe limitations to our approach.

5.1 Experiment environment

We used the open source Java-based ESB software Mule 2.2.1. Our load balancing ESB ran on Intel Core2Duo 2.4GHz PC with RAM 2 GB.

We deployed four services with the same service type and set up Apache server 2.2.11 [26] for publishing load information in VirtualBox V.2.2.4 on host CPU Intel Core2Duo 1.6 GHz, all running on Ubuntu 9.04 with RAM 128 MB. All PCs were connected over a 100 Mbps LAN as shown in Fig. 9.

The services that are used in our experiment all computer permutations of 10 elements. For example, given "abcdefghij", what are the possible ways that these characters can be ordered?

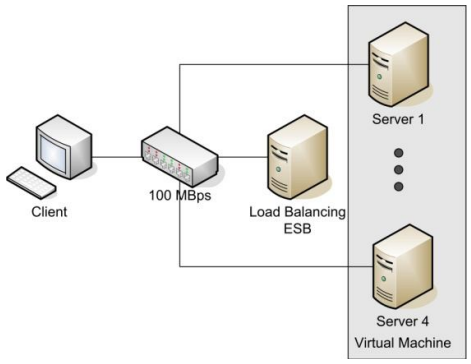


Fig. 9 Experiment testbed

5.2 Experiment method

In our experiment, a client sends a message every second to a service type. The client sends the messages to the ActiveMQ queue and Mule will retrieve the incoming request from the message pool. The time is recorded as StartTime. When the client receives a reply, the time (ReplyTime) is recorded, and finally the response time for that interaction is recorded as follows:

$$ResponseTime = ReplyTime - StartTime$$

- StartTime: Time when message is sent from the client side.
- ReplyTime: Time when the client receives a return message.
- ResponseTime: The amount of time between sending the request and receiving a response.

The frequency of message sending was determined so that the message pool would always be full. Thus, adding more clients would basically have had no effect on the outcome of our experiment.

Finally, for the threshold and leastLoad strategies, the threshold value was set at six messages.

5.3 Experiment result and discussion

Fig. 10 shows the result of the average response time for 100 messages for each of the balancing strategy. It shows that sequencing, i.e., when no load balancing was conducted, had the worst result. Thus, we can conclude that our load balancing approach was able to make better use of the available services.

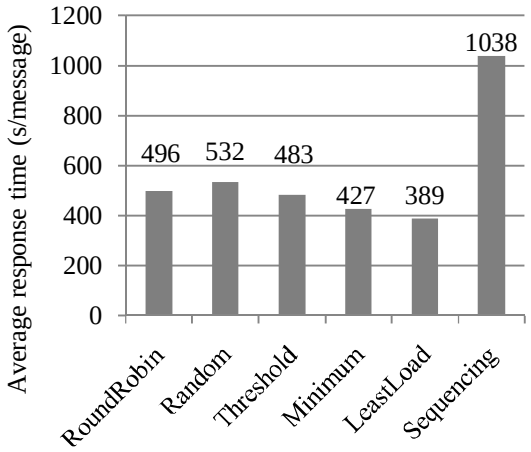


Fig. 10 Average response time

Fig. 10 also shows that the LeastLoad balancing strategy was the most effective strategy for balancing. On the surface it would seem that the Minimum balancing approach should have the best result. The reason for LeastLoad strategy being better is likely due to the extra processing that occurs when switching to (and/or finding) another service. When the Minimum balancing approach is taken, each time that a message is sent from the client, the "best" service (i.e., the service currently handling the least number of messages) is searched for and then chosen. For the LeastLoad strategy, the previous endpoint and its current message handling count (which have been cached) is first checked. If

the current message handling count is under the threshold, then the message is sent to that endpoint. However, if it is over the threshold, then the next service that the message should be sent to needs to be computed in the same way as the Minimum approach. This difference in the frequency of computing which service to send to is the likely reason why LeastLoad balancing performed better than the Minimum strategy.

In Fig. 11, we compare the average response times for the LeastLoad strategy, when the threshold takes a value between 4 and 12. The results show that the response time was best when the threshold was 6 messages. Excluding when the threshold was 4 messages, the results show that the response time was better when the threshold was lower. This is because if the threshold is set high, then the same service must handle more messages before a message is sent to another service. This means that there are services that are not doing anything. It is obviously better if the messages are distributed, and of course this is the point of load balancing.

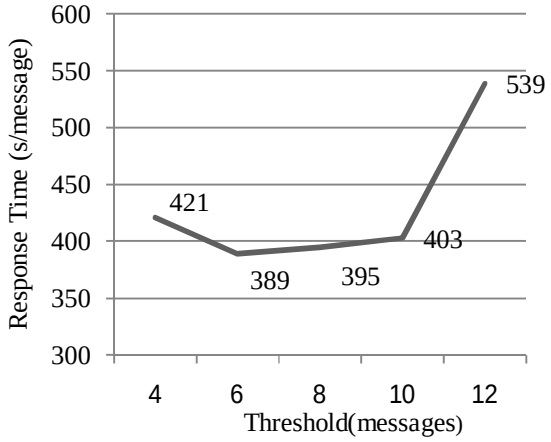


Fig. 11 Performance of the LeastLoad strategy under different threshold values

The exception is when the number of messages was four. The reason for this is the same as the difference between Minimum strategy and LeastLoad strategy. When the threshold is four, the switching occurs more frequently than when the threshold is six. This is where the overhead for computing which alternate service to send a message to can no longer be ignored. Thus, the response time for threshold value four was worse than when the threshold value was six.

Our proposed service type is based on the service property model of CORBA trader. The result of our experiment is similar to the result in [11]. The results in [11] also showed that the LeastLoad strategy was the most effective.

5.4 Limitations

There are several inherent limitations to our approach.

First, there must be multiple services of the same type for our approach to have any affect at all. Currently, we require the services to have the same parameter and return data type to belong to the same service type. We are considering if there are other ways to define a service type such as using ontology, so that the number of services belonging to the same type will increase leading to more candidates for load balancing.

Second, the granularity of services must be considered when registering, or else stateful services will become problematic. For example, if the client starts using a hotel reservation service, then all messages must be sent to the same service. A similar issue exists for services that require membership.

Third, providers must register their service according to a service type, i.e., although small, there is some extra work for the provider.

6 Conclusion

In this paper, we proposed a load balancing mechanism for ESB, which enables the dynamic selection of services which messages should be sent to. The key to this mechanism is the introduction of service types. Services belonging to the same service type have the same function and signature. Thus, messages can be sent to any service belonging to the specified service type. We implemented our approach, and conducted an experiment.

Our main contributions are as follows:

1. Our proposed load balancing is done between different services with the same function, not between replicated services.
2. Our "service type" enables the dynamic selection of the target service. The candidate services are not listed in an ESB configuration file a priori.
3. We compared and discussed the differences between balancing strategies

As for future work, we are considering how we can handle the first two issues that were given in section 5.4, specifically other ways to define a service type, and how stateful services can be handled. Furthermore, we plan on incorporating QoS in dynamic service selection. We also consider other information that can be sent, such as service consumer information for use with service property attribute to enable more powerful service selection.

References:

[1] ESB Interoperability Standards. Available from: http://www.ibm.com/developerworks/library/specification/ws-esb-interop/index.html

[2] F. Ismaili, B. Sisediev, Web Services Research Challenges, Limitations and Opportunities, *WSEAS Transactions on Information Science & Applications*, Volume 5, Issue 10, 2008.

[3] S. Kambhampaty, S. Chandra, Service Oriented Architecture for Enterprise Applications, *Proc. of 5th WSEAS Intl. Conf. on Software Engineering, Parallel and Distributed Systems*, 2006, pp. 48-54.

[4] Z. Panian, Requirements-driven Approach to Service-oriented Architecture Implementation, *Proc. of the 6th WSEAS Intl. Conf. on Multimedia, Internet & Video Technologies*, 2006, pp. 90-95.

[5] X. Bai, J. Xie, B. Chen, S. Xiao, DRESR: Dynamic Routing in Enterprise Service Bus, *Proc. of Intl. Conf. on e-Business Engineering*, 2007, pp. 528-531.

[6] Mule open source ESB. from: http://www.mulesoft.org/display/MULE2USER/Outbound+Routers#OutboundRouters-RoundRobin

[7] Apache ServiceMix. The agile open source ESB. from: http://servicemix.apache.org/how-do-i-configure-an-endpoint-resolverpolicy.html

[8] Google platform. from: http://en.wikipedia.org/wiki/ Google_platform

[9] A. Gorbenko, V. S. Kharchenko, A. Romanovsky, Using Inherent Service Redundancy and Diversity to Ensure Web Services Dependability, *LNCS 5454, Methods, Models and Tools for Fault Tolerance*, 2009, pp. 324-341.

[10] Y. Kono, S. Takada, N. Doi, A Framework for Multiple Service Discovery and Robustness, *Proc. of the 8th IASTED Intl. Conf. on Software Engineering and Applications*, 2004, pp.546-551.

[11] J. Balasubramanian, D. C. Schmidt, L. W. Dowdy, O. Othman, Evaluating the Performance of Middleware Load Balancing Strategies, *Proc. of 8th Intl.. Conf. on Enterprise Distributed Object Computing*, 2004, pp. 135-146.

[12] I.-Y. Chen, G.-K. Ni, C.-Y. Lin, A runtime-adaptable service bus design for telecom operations support systems, *IBM Systems Journal*, Vol.47, No.3, 2008, pp. 445-456.

[13] B. Wu, S. Liu, L. Wu, Dynamic Reliable Service Routing in Enterprise Service Bus, *Proc. of Asia-Pacific Service Computing Conf.*, 2008, pp. 349-354.

[14] O. Othman, C. O'Ryan, D. C. Schmidt, Designing an Adaptive CORBA Load Balancing Service Using TAO, *IEEE Distributed Systems Online 2(4)*, 2001.

[15] Object Management Group. CORBAservices: Common object specification. Version 1.0, May 10, 1996.

[16] J. Wang, Y. Ren, D. Zheng, Q. Wu, Agent Based Load Balancing Middleware for Service-Oriented Applications, *Proc. of the 7th Intl. Conf. on Computational Science Part2*, 2007, pp. 974-977.

[17] J. Roca, J. C. Ortega, J. Antonio Alvarez, J. Mateo, Data Neighboring in Local Load Balancing Operations, *Proc. of 9th WSEAS Intl. Conf. on COMPUTERS* , 2005, pp. 497-533.

[18] K. Kaario, T. Hämäläinen, P. Raatikainen, Adaptive Parameter Setting for QoS Aware Load Balancing Algorithm, *WSEAS Transactions on Communications*, Vol. 1, Issue 1, 2002,pp. 144-149.

[19] G. G. Fernandez, J. S. Carrion, L. J. Aguilar, I. M. Collado, A New Approach to Dynamic Load Balancing across Multimedia Servers, *WSEAS Transactions on Computers*, Vol. 5, Issue 11, 2006, pp.2758―2764

[20] Y. Taher, D. Benslimane , M. Fauvet, Z. Maamar, Toward an approach for web services substitution, *10th Database Engineering and Applications Symposium*, 2006, pp. 166-173.

[21] R. Pianwattanaphon, T. Senivongse, Compatibility by service type model for automatic web service substitution , *Proc. of 9th Intl. Conf. on Advanced Communication Technology*, 2007, pp. 76-81.

[22] uddi.org. UDDI. from: http://uddi.xml.org/

[23] Java Message Service from: http://java.sun.com/products/jms/overview.html

[24] The Apache software foundation. Apache Active MQ open source message broker, from: http://active mq.apache.org/

[25] JDOM Available from: http://www.jdom.org/

[26] Apache HTTP server project from: http://httpd.apache.org/