Dynamic Service Selection Capability for Load Balancing in Enterprise Service Bus

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: - The concept of SOA and its enabling technologies such as ESB are becoming more and more popular. This enables services that have been built on various platforms to be used. Although the number of available services is growing, most ESB implementations only allow static routing, i.e. the service that messages are sent to is pre-determined and fixed. Thus, even though there may be multiple services that have the same function, only one or two may be used. We thus propose to incorporate load balancing feature into an ESB. Unlike conventional load balancing, our approach does not balance among replicated services; we conduct balancing among services that may be provided by different providers. In order to realize this, we introduce the concept of service type. We also show the results of an experiment.

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

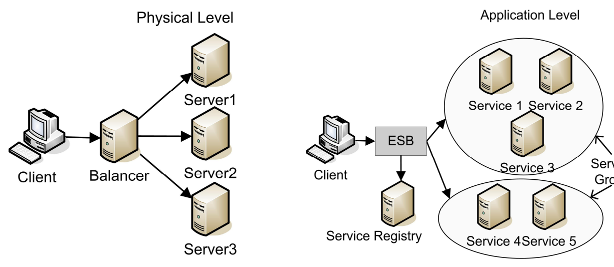
1 Introduction

A Service Oriented Architecture (SOA) is a design pattern which decouples service providers from service consumers. One enabling technology for SOA is the enterprise service bus (ESB). ESB is an important middleware tool for integrating services based on various platforms. Its main focus is to route messages among different services.

Current ESB implementations support several message routing patterns, but can execute based only on static configuration [1]. If the requests increase dramatically, it is better to distribute the requests to existing multiple services which can satisfy the same requests. For example, Mule [2] and Service Mix [3] 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.

Many Web sites take a load balancing approach to handle the 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 server one accesses will differ depending on the load at that time [4]. 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 also be taken to satisfy the issue of “too many” requests.

We take a different approach. Instead of replicating a service, we group services with the same function. To this end, 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 switches between services of the same type (Fig. 1 (b)), using strategies such as random, round-robin, threshold, minimum, and least load [5]. Furthermore, services belonging to the same service type may change during runtime, so there is no need to pre-determine concrete candidate services.



(a) Conventional balancing (b) Our approach

Fig. 1 Load balancing

Our approach assumes the following:

- A registry exists which contains information on the services.
- Client programs can attach the service type in the header of a request message.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes our service type model. Section 4 then presents our balancing ESB framework 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

In the ASB project [6], a target service is chosen at run time. The target service selection is based on a ranking which uses information such as execution time and expected availability. Their approach focuses only on an adaptable service bus for dynamic service composition only. The DRESR project [1] defines the Abstract Routing Path using abstract service names, which are instantiated at runtime by replacing the abstract service names with the real URIs. B. Wu et al [7] proposed a method for dynamic reliable service routing. The context of application information related to the request message is added for use in service discovery. If the request does not respond within a suitable time, 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

Wang et al [8] proposed a load balancing middleware for service-oriented applications. 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. Their approach uses machine-learning to predict loading. Note that their approach is based on having replicas of the same service to achieve load balancing. The Cygnus and TAO projects [5, 9] apply adaptive balancing in CORBA [10] by adding load condition dynamically. However, this approach uses fixed and static replica management and load migration to relieve overload. Migration delay may cause problems, and it cannot support heterogeneous services.

2.3 Service substitution

Taher et al [11] proposed a 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 filter services. Pianwattanaphon et al [12] 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 Model

Services are normally declared in a service registry such as UDDI [13]. 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 ESB to dynamically determine which service to send a message to. We propose to group Web services with the same function that can satisfy the same request by adding an attribute.

3.1 Service type

We introduce the concept of "service type". Each service in our registry 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 Service type examples

ServiceType Name	ServiceSignature		ServiceType Property
	Parameter	Return DataType	
HotelReservation	Location: String Room:Int	Boolean	Availability
FlightInformation	Departure:String Arrive:String Date:Date	List	ExecutionTime
CarRent	CarType:String Hour:Int	List	Accessibility
ResturantSerching	City:String	List	Accessibility
DocumentPrinter	Document: File	Boolean	Availability
PhotoSharing	Picture:File	Boolean	Security
MoviePreview	MovieName: String	MediaFile	Availability

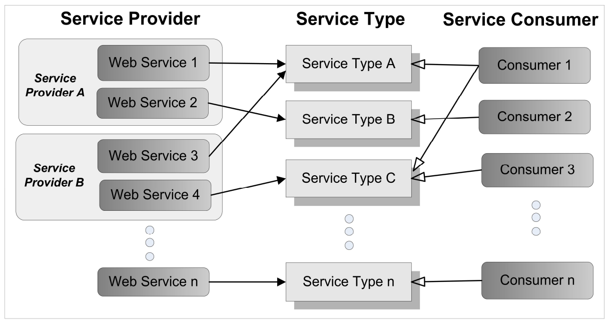


Fig. 2 Sharing service type

3.2 Incorporating service type

The service provider defines the service information, including the service type. 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.

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.

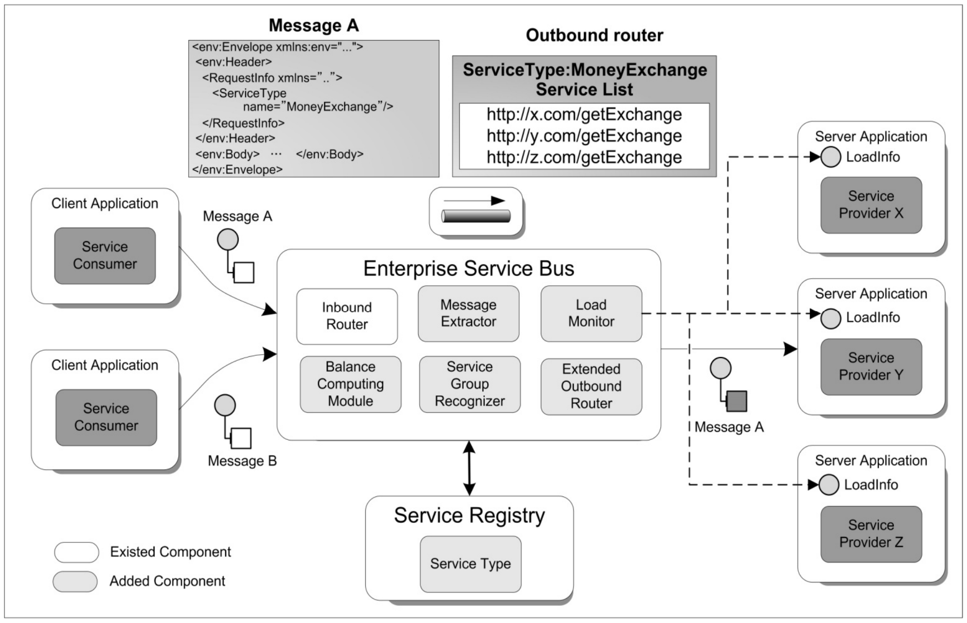
4 Balancing Mechanism

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 mechanism.

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 [14] channel. When a client sends a message, the message is stored in a request queue of ActiveMQ 5.2.0 [15] which is an open source JMS.
- **Message extractor** is a module for extracting the header and body from the message. The service type is an essential header attribute in our approach.
- **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 main component for managing the sending of messages. The actual destination of a message is decided using a balancing



```
<env:Envelope xmlns:env="...">
 <env:Header>
  <RequestInfo xmlns="..">
    <ServiceType
          name="MoneyExchange"/>
  </RequestInfo>
 </env:Header>
 <env:Body>   ...   </env:Body>
</env:Envelope>
```

Fig. 3 ESB enhanced with balancing mechanism

strategy. This module connects to the load monitor module for getting load information, and uses the obtained data to check which service application has a the least amount of load at that time.

- **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 period for updating the load info data.

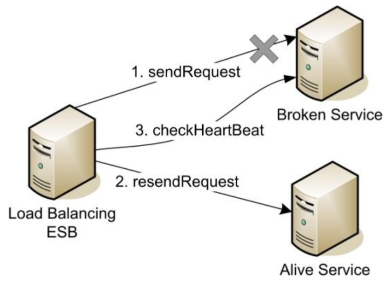


Fig. 4 Alive service checking

- **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.

4.2 Balancing procedure

The steps in balancing are given below:

1. A client sends a message to invoke a service. The actual service is not specified; instead the service type is specified in the message header. Service properties such as availability 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.
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.
5. The service registry returns a list of services belonging to the same service type to the service group recognizer.
6. 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.
7. The balancing module requests the load information (loadInfo) from the load monitor module.
8. The load monitor module asks for the current loadInfo from the service providers.
9. The service provider returns the current load information to the load monitor.
10. The load monitor forwards the load information to the balancing module.
11. The balance computing module chooses the target service based on the balancing strategy, and forwards the endpoint to the extended outbound router.
12. The extended outbound router sends the message to the actual destination service.

4.3 Balancing Strategy

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 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 4 services with the same service type and set up Apache server 2.2.11 [16] 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.

5.2 Experiment method

The client sends a message (StartTime) every second to a service type. When the client receives a reply, the ReplyTime is recorded, and then the response time for that interaction is recorded. The response time at client side is as follows:

$$ResponseTime = ReplyTime - StartTime$$

- **Response time:** The amount of time between sending the request and receiving a response [17].
- **StartTime:** Time when message is sent from the client side.
- **ReplyTime:** Time when the client receives a return message.

For the threshold and leastLoad strategies, the threshold value was set at 6 messages.

5.3 Experiment result and discussion

Fig. 5 shows the result of the average response time for 100 messages for each of the balancing strategy. It shows that the sequencing, i.e., when no load balancing was conducted, had the worst result. Thus, we can conclude that our approach was able to balance the load between similar services.

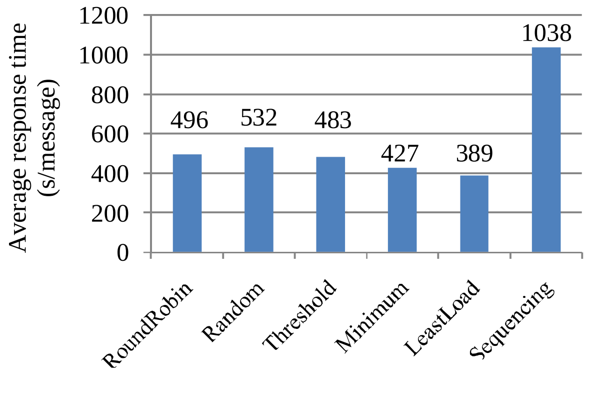


Fig. 5 Average response time

Fig. 5 also shows that the leastLoad balancing strategy is 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 this is likely due to the extra processing that occurs when switching to (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 always computing which service to send to, and only sometimes computing is the likely reason why leastLoad balancing was better than the minimum strategy.

In Fig. 6, we compare the average response values 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.

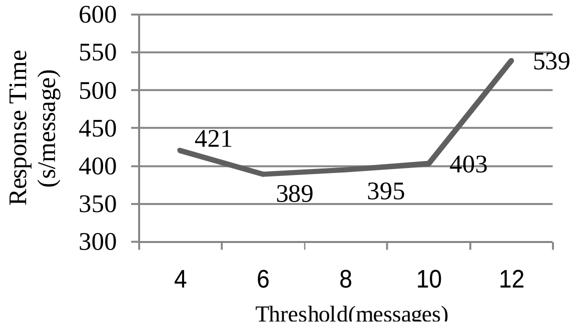


Fig. 6 Performance of the leastLoad strategy under different threshold values

The exception is when the number of messages was 4. The reason for this is the same as the difference between minimum strategy and leastLoad strategy. When the threshold is 4, the number of switching is more than when the threshold is 6. 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 4 was worse than when the threshold value was 6.

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 [5]. The results in [5] 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, providers must register their service according to a service type, i.e., although small, there is some extra work for the provider.

Third, 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.

6 Conclusion

In this paper, we proposed a balancing mechanism that dynamically routes messages based on the specified balancing strategy. We first proposed a service type model, where each service belongs to a type. Messages can then be sent to any service that belongs to the same type. We implemented our approach, and conducted an experiment.

Our main contributions are as follows:

1. Our 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 plan to consider QoS in dynamic service selection of ESB and study the ranking of matched service results. 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] 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

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

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

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

[5] 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

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

[7] 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

[8] 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

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

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

[11] 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

[12] 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

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

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

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

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

[17] S. Kalepu, S. Krishnaswamy, S. W. Loke, Verity: A QoS Metric for Selecting Web Services and Providers, *Proc. of 8th Intl. Conf. on Web Information Systems Engineering Workshops*, 2003, pp. 131 - 139