Throughput Evaluation of different Enterprise Service Bus Approaches

Stein Desmet, Bruno Volckaert, Steven Van Assche, Dietrich Van Der Weken,
Bart Dhoedt, Filip De Turck
Department of Information Technology (INTEC) – IBCN
Ghent University – IBBT
Gaston Crommenlaan 8, bus 201, B-9050 Gent, Belgium
stein.desmet@intec.ugent.be

Abstract*— *Recently, the notion of Service Oriented Architecture (SOA) has become increasingly attractive to companies. Especially the concept of an Enterprise Service Bus (ESB) is becoming a popular way to implement SOA. However, there is no universal definition for an ESB, and a wide variety of different approaches to the concept exists. This article aims to provide some clarity by presenting an evaluation of four different ESBs, two commercial and two open-source, each with their own implementation of the concept of ESB. This is done by comparing their underlying architecture, provided functionality, usability and detailing performance results.

Keywords: SOA, ESB, comparison, architecture, functionality, performance

I. Introduction

In the last few years, the concept of a Service Oriented Architecture - or SOA - has grown significantly. Consequently, it becomes increasingly attractive to companies, as it promises a high degree of flexibility, reusability, and cost savings.

An SOA is an architectural model in which a loosely coupled collection of services communicate in some way with each other. This can be some simple form of exchanging data, but could also involve several services coordinating to get some complex activity done. Note that service doesn't necessarily mean Web Service.

One of the several possible ways to realize SOA, is the architectural pattern Enterprise Service Bus (Fig. 1). One could think of an Enterprise Service Bus as the glue to bind services together. However, giving an exact definition of an ESB is no simple task, since definitions vary from source to source, and some even claim an ESB to be a product, and not a pattern. Nonetheless, most definitions agree on several characteristics, which can be used to construct a somewhat general definition.

The ESB is based on standards, and it is, as its name suggests, a messaging system that acts as a backbone for services. Communication between services themselves, and between clients and services goes through the bus, instead of linking them with point-to-point connections to each other. Essential is that the Bus supports a wide range of transport methods, and the ability to perform transformations on the data passed within the bus. This ensures there is no coupling

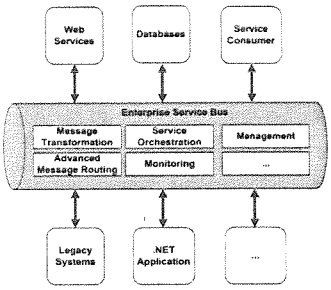


Fig. 1: The Enterprise Service Bus.

between calling a service, and the transport protocol and message format used. Generally, data inside the bus will be in XML format, and transformations are done through XSLT. Other essential features include content-based routing of the data, support for standards such as Web Services and usually the possibility to orchestrate services. The term service orchestration means coordinating several services to perform a more complex, usually long-running, task.

Three important standards for implementing SOA are Java Business Integration (JBI) [1], Service Component Architecture (SCA) [2] and Windows Communication Foundation (WCF) [3]. Main supporters for JBI include Sun Microsystems and Tibco Software, while SCA's primary vendors are BEA and IBM.

These standards also provide a possible foundation for the Service Bus itself. However, not all vendors take this approach, and most choose to implement ESB in their own way, with their own visions in mind. This means that, unless both support the same standard, implementing the same use case on ESBs from two different vendors, will usually require an entirely different approach. Which ESB is the best choice for a specific situation, depends on requirements such as performance, tooling, supplied services and transport protocols.

For this article, four ESBs were selected, two commercial, and two open source. The chosen commercial ESBs are respectively *Aqualogic Service Bus v2.5* from BEA and *Websphere ESB v6.0.1* from IBM. For the open-source camp we have selected *ServiceMix v3.1*, an Apache product, and MuleSource's *Mule v1.4*.

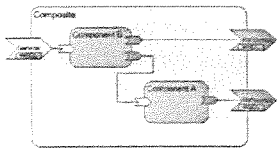


Fig. 2: The basic components of SCA.

This article's structure is as follows: Section II discusses related work, Section III provides a review on the selected ESBs, and IV provides performance results. A conclusion is drawn in Section V.

II. Related work

An interesting comparison of serveral ESBs is presented in [4]. This article generally focuses on commercial ESBs and the functionality they provide. An additional article is also available, which reviews each Service Bus separately [5].

Another comparison between products, including individual reviews, is provided by Forrester [6], although not all of these reports are free of charge.

[7] describes performance benchmarking and capacity testing of an Enterprise Service Bus in detail.

III. Functional Comparison

First, Section III-A gives a short description on both the SCA and JBI specifications. WCF is not included, as it is not used by any of the reviewed ESBs.

Secondly, Section III-B presents a review on every selected ESB separately, where we take a look at a number of different characteristics for each of these products. Points of interest include:

- underlying architecture and implementation. Is it based on a standard, or is it a proprietary architecture?
- provided out-of-the-box features and supported standards. This includes both available services, such as service orchestration, and transport mechanisms, such as HTTP or JMS.
- general usability. How easy is it to set up the product, what kind of tooling is available, what kind of work is involved to actually implement a use case.

Note that this is necessarily a global review as going into every single aspect of the products would take quite a few pages.

Eventually, Section III-C shows a comparison matrix of the four products.

A. Standards

Service Component Architecture

The basic element in SCA is the *Component* (Fig. 2): it exposes services, called *Services*, and may depend on other services, referred to as *References*, both of which are defined by WSDL interfaces. A Component can have any implementation, such as Java, C++ or BPEL, and is configured by XML descriptors.

Multiple Components are then wired together to form a *Composite*. Just like a Component, it exposes Services and References, and may in fact be used as the implementation for another Component.

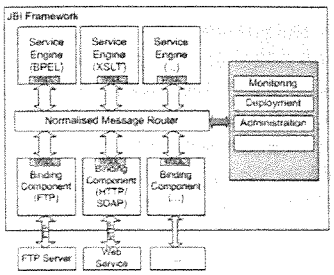


Fig. 3: The basic structure of JBI.

Services and References use *Bindings*, which define respectively how References are accessed and how Services can be accessed. Examples of possible Bindings are HTTP/SOAP or JMS. These allow for a decoupling between a Component's specific implementation and the way it communicates.

Java Business Integration

In JBI, components do not communicate directly with each other, but are instead plugged into a framework, called *Normalised Message Router*, which delivers messages between the various components. A component may provide services which are described by WSDL interfaces. Two different types of components are distinguished: *Service Engines* and *Binding Components* (Fig. 3). Service Engines provide services to other components, and may consume other services as well. Examples of provided services could be data transformation or routing capabilities. Binding Components on the other hand, provide connectivity to external systems or services through a particular transport protocol, such as HTTP, JMS or JDBC. Binding Components too can act either as service provider or service consumer.

Deploying projects in JBI is done by defining *Service Units*: these are project-specific XML descriptors for Service Engines and Binding Components. These Service Units are packaged into a *Service Assembly* to form the project.

B. Overview of Existing Approaches

IBM Websphere ESB

IBM offers several products in the ESB area: the *Websphere ESB* (WESB), the *Websphere Process Server* (WPS) and the *Websphere Message Broker* (WMB). The Process Server and the Websphere ESB are closely related: the former is a superset of the latter and adds functionality such as BPEL and human interaction. Message Broker is more related to Websphere MQ and offers integrations for more heterogeneous environments, while Websphere ESB is more service oriented. Unfortunately, only Websphere ESB arrived in time to be included in this review.

The installation is divided into two separate packages: a two-disc installer for WESB itself, and a four-disc installer for the development environment,

or *Websphere Integration Developer*(WID). Installation can be done in either graphical or silent mode.

The architecture of the service bus is completely based on the SCA specification - although with a few custom extensions here and there - and is implemented on top of IBM's Websphere Application Server. This means that everything inside the bus is based on services composed out of SCA components, that are wired together to form a flow. The amount of provided services is not all that extensive and sometimes gives the impression of promoting the use of Process Server.

External systems that aren't service-based, such as databases or legacy systems, are accessed through adapters. There are two different kinds of adapters available: WBI Adapters and Websphere Adapters. WBI adapters act as a service facade for the external system, but are now considered obsolete, and are being replaced by Websphere Adapters. This type is based on the JCA (*J2EE Connector Architecture*) specification, and as such, resides in the WESB itself. They access the external systems directly (e.g. through JDBC), but are exposed as SCA components internally. This eliminates the sometimes substantial overhead of a facade, but on the other hand does increase the required resources on the WESB server.

The development environment is based on Eclipse, and provides a graphical representation for SCA. Everything is done from within WID: from defining or importing SCA components and wiring them together to form the flow - which is simply done by drawing lines connecting the components - to deploying the entire application on the service bus. At the time of writing, WID suffered from some minor issues here and there, but a 3 GB update was recently released, bringing WID from version 6.0.1 to 6.0.2, which remedies these issues. Apart from WID, a web-based console is also available to administrate and monitor the service bus.

Both the service bus and the development environment offer such an amount of options that documentation really is required to get started. Luckily, there is a multitude of comprehensible tutorials and RedBooks available on IBM's website.

BEA AquaLogic Service Bus

BEA also offers a wide range of ESB related products: the foundation is the *AquaLogic Service Bus (ALSB)*, and it can be enhanced with additional products, such as Business Process Management or human interaction.

Installing the Bea Aqualogic Service Bus is quite easy: simply run the one-disc installer - available in graphical, console and silent flavour - which contains both ESB and development environment.

The Aqualogic Service Bus is not based on a standard like SCA or JBI, but uses a proprietary architecture, implemented on top of the WebLogic Application Server. ALSB divides services into two categories: *proxy services* and *business services*. Proxy services reside inside the Bus and communicate with external clients or other proxy services. Business services on the other hand, are services external to the Bus, and only communicate with proxy services.

Theoretically speaking, the concept of adapters does not exist in AquaLogic, as the focus seems to lie more on providing integration in mostly service-enabled environments. This means that when access to for example an external database or legacy system is needed, a service facade must be put in front of it, and this needs to be imported as a business service. Naturally, a facade will usually have a higher overhead than an adapter based approach. However, BEA is working on this issue, as direct JDBC functionality was added to version 2.6, which was released at the time of writing. Another available option would be to obtain the BEA WebLogic Integration package, which can be combined with ALSB to add additional communication possibilities.

Tooling for the Aqualogic ESB differs from other products in that the development environment is completely web-based: once the Service Bus has started, two different web interfaces are available: one for building integrations and one for administrating the server. This concept works surprisingly well, since the interfaces are completely dynamic, and use HTTP session capabilities for maintaining state. One benefit is that this poses little requirements on the developer's machine: only a JavaScript capable browser is needed.

Defining the flow uses a proprietary graphical notation, which is a fairly easy process, not limited by the web-based IDE. A wide range of routing and transformation options are available to define the flow.

As with the Websphere ESB, documentation is necessary to get started, which is readily available from the website, in the form of both tutorials and manuals.

ServiceMix

ServiceMix is an open-source ESB project currently residing in the Apache Incubator. Installation merely consists of unzipping the archive from the website. At the moment of writing, the only tooling included is provided by means of Apache Maven: it creates a project skeleton, which then needs to be modified. The result is compiled, packaged and deployed with Maven, all of which is done from the command line. However, an alternative is available: LogicBlaze offers *Fuse*, which can be viewed as a distro of ServiceMix. It bundles ServiceMix itself and several additional components such as an UDDI registry, JMS server and more importantly: an Eclipse-based development environment. This comes in the form of a simple installer, requiring no dependencies.

ServiceMix's internal architecture is completely based on the JBI specification, and is thus completely service based. The implementation itself is built from the ground up, not based on any existing application server, making it more light-weight. A wide range of both Service Engines and Binding Components are

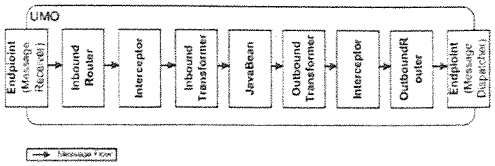


Fig. 4: The basic make-up of an UMO.

available to get started with, allowing amongst other things advanced routing patterns. Work is also being done on a container that allows integration with SCA.

Accessing both service-oriented and non service-oriented external systems can be done through a Binding Component, as explained in Section III-A on JBI.

As stated before, development is largely Maven-based: projects can be either developed by using Maven directly from the command line, or by using LogicBlaze's IDE, which actually still utilizes Maven under the covers. Unfortunately, Maven itself and its derived Eclipse plug-in are currently not entirely issue-free yet, which can potentially cause some minor problems from time to time.

At the moment, even with the Fuse IDE, some XML knowledge is required, since a graphical way to configure components is still under development. This means that it is still necessary to modify the Maven-generated XML descriptors by hand, in order to link the endpoints together to define a flow. This may sound complicated, but is actually offset by JBI's relative simplicity.

One shortcoming of ServiceMix though, is that the documentation process was not complete yet at the time of writing: a few parts are either blank or not detailed enough.

Mule

Mule is an open-source service bus from MuleSource, a company specifically founded for the development of the Mule platform. Mule's philosophy seems to deviate from that of the average ESB: it's based on a proprietary architecture, and is actually more of a lightweight messaging platform with full ESB capabilities.

Setting up Mule is, just like ServiceMix, rather straightforward: unzip the archive and set an environment variable. The entire installation only takes up a mere 30 MB.

The architecture of Mule is based on the concept of *Universal Message Objects*, which are hosted within a container, called *Mule Model*. The UMOs supply the business logic and communicate with each other and external applications through endpoints. The Model manages the UMOs: it handles message flow between them, as well as component lifecycling, threading, pooling, etc.

Figure 4 illustrates the basic make-up of an UMO: each has at least one incoming and one outgoing endpoint, bound to a specific transport protocol. An interesting feature of these endpoints is that they support streaming data. At the core of the UMO resides a JavaBean which can perform arbitrary tasks when a message is received. Surrounding the JavaBean on both sides are the routers, interceptors and transformers. Routers and transformers are self-explanatory, and interceptors can be used to intercept message flow, e.g. for logging or authorization purposes.

A wide variety of transport protocols, routers and transformers are supplied with the distribution, even some that are rarely found in other ESBs, such as TCP and UDP transport protocols. This illustrates that more low-level activity is also possible with Mule.

At the moment, one downside of Mule is that project deployment is still under development: both hot-deployment and project discovery are not yet available. In other words, Mule requires a restart when a new application needs to be added, and has to be supplied with its location.

Another aspect which is currently under heavy development, is the tooling: the available development environment - an Eclipse plug-in - is still very basic, requiring manual creation and modification of the XML descriptors. Luckily, Mule's architecture is not very complex, so this is a rather simple process. Nevertheless, both deployment and tooling share a very high priority on MuleSource's to-do list.

Just like ServiceMix, the documentation process is not entirely up-to-date yet, causing some incomplete parts here and there in the documentation.

C. Comparison Matrix

Table I displays a comparison matrix, which by no means claims to be complete, but rather attempts to highlight the more important or interesting features. Note that components which must be obtained separately are not included in the table.

IV. Performance Evaluation

Due to large differences in architecture, implementation and provided features, making a fair performance comparison is rather difficult. Still, a few simple test cases can provide some insight into the scalability of the respective service buses. Also note that different use cases might yield different results: one ESB might perform better in one use case, while it may perform comparably worse in another. Take for example two use cases, where one needs to perform databases access, and the other does not. A random ESB might perform best in the latter, but might do worse in the former, because it uses a facade based approach, while its relatives make use of direct JDBC calls.

A. Test Scenario

For this paper, performance testing is focused on two tests: the first one consists of clients making calls on a Web Service proxy hosted in the service bus, which in turn routes the requests to an external Web service. For the second test an additional XSLT transformation is performed on the messages, before passing them to the external Web Service. In both tests, round-trip times and throughput are measured.

TABLE I: Comparison matrix between ESBs. Y=included, N=not included, C=container available, (1)=read access only, (2)=separately available, (3)=available in WPS

		WESB	ALSB	SM	Mule
Features					
	XSLT	Y	Y	Y	Y
	Pluggable XSLT engine	Y	Y	Y	Y
	Custom transformations	Y	Y	Y	Y
	Scripting	N	N	Y	Y
	Rules engine	N (3)	N (2)	Y	Y
	BPM	N (3)	N (2)	Y	Y
	Scheduling	N	N (2)	Y	Y
	Built-in datastorage	Y	Y	Y	N
	Java calls	Y	Y	Y	Y
	Monitoring	Y	Y	N	N
	Call tracing	Y	Y	N	N
	Built-in test client	Y	Y	N	N
Transports					
	HTTP/S	Y	Y	Y	Y
	JMS	Y	Y	Y	Y
	FTP	Y	Y	Y	Y
	E-mail	Y	Y	Y	Y
	File	Y	Y	Y	Y
	JDBC	Y	Y (1)	N	Y
	REST	N	N	N	Y
	TCP	N	N	N	Y
	UDP	N	N	N	Y
	VFS	N	N	Y	Y
	XMPP	N	N	Y	Y
	RSS	N	N	Y	N
	EJB	Y	Y	Y	Y
	JCA	Y	Y	Y	N
Standards					
	SOAP	Y	Y	Y	Y
	WS-security	Y	Y	Y	Y
	SCA	Y	N	C	N
	JBI	N	N	Y	C
	BPEL	N	N	Y	Y
Other					
	Commercial support	Y	Y	Y	Y

B. Test Setup

The setup consisted of three machines, connected by a dedicated 100Mbit switch. Each contained two dual-core AMD Opteron 2200 cpus, and four GB RAM. One machine hosted the Web Service, the other the ESB, and one ran the clients. Running testclients on such a high level machine might seem strange at first, but this allows simulating a larger number of clients more accurately, as opposed to simple single core machines.

The Web Service in question simply performs a string echo operation and is hosted using Apache Axis 1.4. Its SOAP style is RPC/encoded [8]. While nowadays the document/literal style is more recommended, rpc/literal was chosen to see how well it was supported by the ESBs, as it used to be more widely utilised. One downside though, is that rpc/encoded has a lower performance than doc/literal [9].

The XSLT transformation was kept equally simple: it merely copies the SOAP message, while converting the input string argument to upper case. Care was taken to insure that each ESB used the exact same XSLT file, instead of relying on the built-in editors to construct the transformation. The motivation is that an XSLT transformation's performance may vary a great deal depending upon its specific implementation.

To simulate the clients and collect data, the load-testing tool Apache JMeter was used, in combination with JAX-RPC to make the web service calls themselves. The Axis client library was not used, because it has a certain disadvantage: HTTP connections are not pooled, resulting in a continuous creating and discarding of connections.

TABLE II: Results for a simple Web Service proxy, with sixty threads simulating clients. 'Axis' shows the results when directly calling the Web Service without an ESB.

Name	Avg. RTT (ms.)	50% (ms.)	90% (ms.)	Throughput (req./s.)	Avg. clients simulated
Axis	13.12579	2	36	4353.20	57.14
Mule	23.35965	16	39	2468.00	57.65
Bea	40.06326	26	55	1486.50	59.55
IBM	103.5194	84	206	576.10	59.64
ServiceMix	31.24897	25	46	1900.90	59.4

Clients make requests as fast as they can, each 8000 requests per run, discarding the first 4000 to allow the Virtual Machines to reach a steady-state.

Services generally won't exchange very large quantities of data, so messages usually tend to be of average size. Therefore, the size of both request and response SOAP messages was set to 1KB.

The tested ESB versions were ALSB v2.5, WESB v6.0.1, ServiceMix v3.1 and Mule v1.4.

The employed Java Virtual Machine was the Sun Hotspot Virtual Machine 1.6.0-b105. Exceptions are the Bea and IBM Service Bus, which run on their own JVM: respectively IBM JVM 1.4.2 and JRockit 1.5.

During initial testing stages, it became clear that both ServiceMix and Mule would need modifications to their default pooling settings, before they would actually be capable of running the tests. The default thread pool settings in ServiceMix for example, only allowed for a maximum of 32 concurrent HTTP connections. Similarly, Mule's default XsltTransformer's pool settings caused Transformer objects to be continuously created and subsequently discarded, a very costly process - it takes more time than the actual XSLT transformation itself - causing threads to queue up on the Transformer pool. Other tell-tale signs of inadequate thread pool settings were general slow performance, while cpus only carried an average load of 25%, problems which disappeared after setting more appropriate values. It should also be noted that WESB and ALSB showed none of these signs.

C. Results

Figures 5 and 6 show the measured throughput and round-trip times when making calls on the proxies, in function of the number of concurrent clients. For reference, the graphs also display throughput and round-trip times obtained when calling the Web Service directly. Table II presents some more detailed data for a test run with sixty concurrent clients. Showing data for each performed test run in this table would be impractical, as this would fill several extra pages.

Figures 7 and 8, and Table III show the gathered results for the second test.

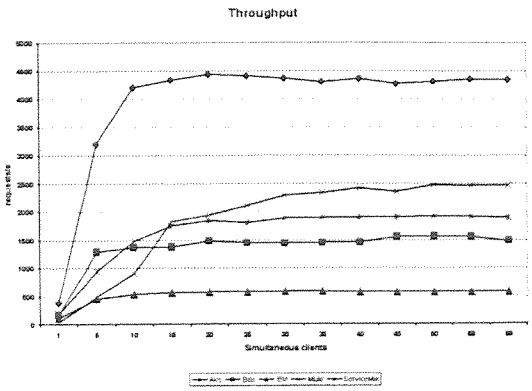


Fig. 5: Throughput for a simple Web Service proxy. Axis refers to the results obtained by directly calling the Web Service.

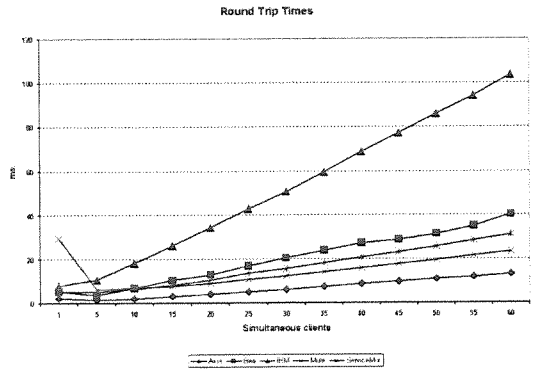


Fig. 6: Round trip times for a simple Web Service proxy. Axis refers to the results obtained by directly calling the Web Service.

D. Verification of results

An important factor when performing benchmarks is making sure the test framework doesn't interfere with the results [10]. This is especially a potential pitfall when simulating multiple concurrent clients on a single machine. This necessarily uses multi-threading, and when there are too many threads, it's possible that they start throttling, which results in two things: firstly, the requested amount of clients is not simulated, and secondly, the response time measurements are skewed, as they include time from when the thread was queued for processor resources.

Luckily, there's a rather simple way to verify whether or not the correct amount of clients was simulated. This uses *Little's Law*, which states that the average number of clients Q in any system is equal to their arrival rate λ, multiplied by their average time spent in the system, or response time R. Additionally, with sufficient measurement duration, it's justified to assume that the arrival rate is equal to the completion

TABLE III: Results for a Web Service proxy with an XSLT added. Sixty threads are simulating clients.

Name	Avg. RTT (ms.)	50% (ms.)	90% (ms.)	Throughput (req./s.)	Avg. clients simulated
Mule	29.85709	23	49	1958.00	58.46
Bea	105.7129	72	146	566.00	59.84
IBM	140.5799	124	250	424.40	59.66
ServiceMix	40.5135	35	60	1470.50	59.54

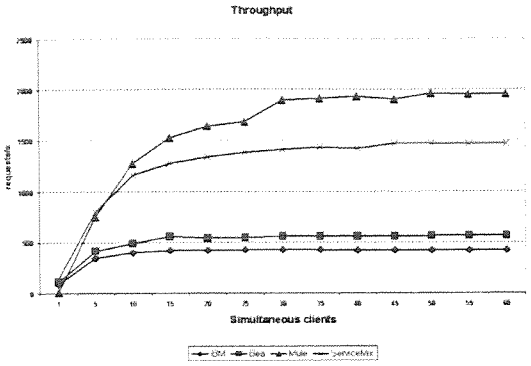


Fig. 7: Throughput for a simple Web Service proxy with XSLT transformation.

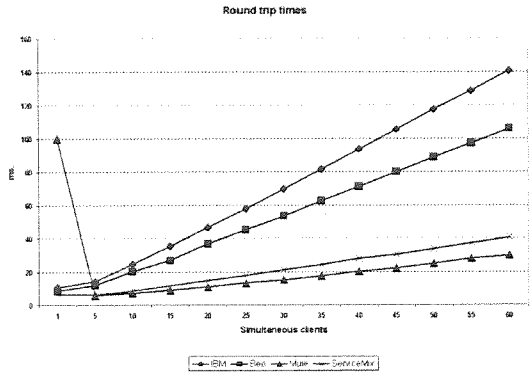


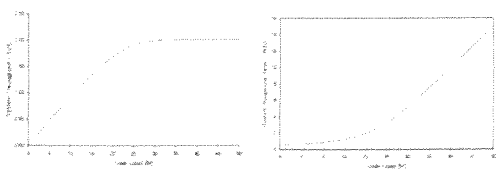
Fig. 8: Round trip times for a simple Web Service proxy with XSLT transformation.

rate, or throughput X. Thus:

$$Q = \lambda R = XR$$

For example, the column 'Avg. clients simulated' in Tables II and III shows some of the values calculated with Little's Law. These prove that there is no problem reaching the specified number of concurrent clients. Similar results were obtained for the other test runs. Remark that we used the measured round-trip time here, which is the response time of the entire system: composed of ESB, network and web server. However, there's no harm in this, as a system's behaviour is determined by its bottleneck stage, in this case the ESB.

Another interesting topic covered in [10] explains what the shape of the throughput and response-time graphs should look like (see Fig. 9). Note that the response-time graph may seem exponential at first, but it in fact exhibits linear behaviour after the knee. This linearity should follow $R = ND_b$, with N the number of clients and D_b the time to service a single



(a) Throughput (b) Response time

Fig. 9: Typical graph shapes in performance benchmarking.

TABLE IV: Round Trip Time increase when adding an XSLT transformation. Measurements were taken for sixty clients.

Name	RTT increase
Mule	127.81%
Bea	263.86%
IBM	135.80%
ServiceMix	129.65%

request in the bottleneck stage of the tested system. As maximum throughput entirely depends on throughput in the bottleneck stage, D_b can be calculated with: $D_b = 1/X_{max}$, where X_{max} is the entire system's maximum possible throughput. When graphs exhibit different behaviour, then something must be wrong in either tested or test system, e.g. excessive paging, throttling threads or resource contention.

When looking at the obtained throughput and response-time graphs, it's clear that they exhibit a behaviour similar to the aforementioned typical shapes, albeit with some minor deviations here and there. This indicates that there are no major problems with the ESBs. Note that the servers might appear to saturate rapidly, but this is only normal when taking into account that clients make requests as fast as they can.

The round-trip time graphs do not diverge significantly from the expected linear behaviour, with one strange exception: Mule performs much slower when it experiences low loads - between 1 and 10 clients - which is visible from both throughput and round-trip time graphs. Repeated tests show the same behaviour each time around, so some one-time event such as the JVM increasing its memory can be ruled out. This odd behaviour can most likely be attributed to thread and object pools being ill-tuned for low client loads.

Table IV shows that response times increase about equally when adding an XSLT transformation to the proxy, except for Bea's Service Bus, where response times more than double. The reason for this is that ALSB embeds the SOAP messages inside another XML structure, thus increasing the time needed for a transformation compared to the other ESBs when applying the same XSLT. When the transformation is defined with the built-in editor, ALSB performs almost two times better, as shown in Table V.

The reason for the open-source products' better performance is that they are not built on top of an application server. On the other hand, this added overhead is compensated by other advantages of an application server foundation, such as easier integration with J2EE-based applications, or the ability to use pre-existing J2EE adapters.

Lastly, it should be noted that WESB as a product is not optimized for performance, but focuses on different aspects. When looking for performance, one should rather consider IBM's Message Broker, which is more geared towards speed.

V. Conclusion and Future Work

In this paper, an overview of different existing approaches to ESB was presented, examining their underlying architecture, capabilities and usability. Additionally, these implementations were subjected to performance testing, in order to get performance scalability results.

TABLE V: ALSB's performance when using the pre-defined XSLT and the transformation created with the built-in editor.

Name	Avg. (ms.)	50% (ms.)	90% (ms.)	Throughput (req./s.)
Xslt (imported)	105.7129	72	146	566
Xslt (editor)	60.6322667	39	91	964

It was clear that each implementation provides the basic required functionality, but that both open-source solutions include certain additional features by default, while these must be obtained separately for the commercial products.

Since they are built on top of an application server, the commercial implementations do not perform as well as the open-source ones, but this approach offers other benefits, such as easier integration with existing J2EE technology.

At the time of writing, the open-source products lag somewhat behind on their commercial counterparts when it comes to tooling, documentation or monitoring, although these are under heavy development.

Nonetheless, each reviewed product certainly delivers a valid way to implement SOA, so choosing the type of ESB will depend on the specific situation and its requirements.

Possibilities for future work include performing additional benchmarks, for example on the routing capabilities and the ability to handle asynchronous requests - another important aspect for ESBs. Evaluating a more complex use case where such things as databases and JMS are required might also yield somewhat more real-world performance results. Another point which certainly merits attention is the use of Business Process Management in combination with ESBs.

Acknowledgement

Part of this work is funded trough the IBBT Geisha project. Filip De Turck acknowledges the FWOV for their support through a postdoctoral fellowship.

References

[1] Sun, "JBI," http://jcp.org/en/jsr/detail?id=208, January 2007.
[2] OSOA, "SCA," http://www.osoa.org/, January 2007.
[3] Microsoft, "WCF," http://wcf.netfx3.com, January 2007.
[4] L. Macvittie, "Make way for the esb," *Network Computing*, vol. 17.05, pp. 41–58, 2006.
[5] ——, "Reviews: Esb technology," *Network Computing*, vol. 17.05, 2006.
[6] K. Vollmer and M. Gilpin, "The forrester wave: Enterprise service bus, q2 2006," Forrester, Tech. Rep., 2006.
[7] K. Ueno and M. Tatsubori, "Early capacity testing of an enterprise service bus," in *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006)*, September 2006, pp. 709–716.
[8] R. Butek, "Which style of wsdl should i use?" *IBM developerWorks*, 24 May 2005.
[9] F. Cohen, "Discover soap encoding's impact on web service performance," *IBM developerWorks*, 01 Mar 2003.
[10] N. J. Gunther, *Analyzing Computer System Performance with Perl::PDQ*. Springer, 2005.

PROCEEDINGS OF
THE 2007 INTERNATIONAL CONFERENCE ON
SOFTWARE ENGINEERING RESEARCH & PRACTICE

SERP'07

Volume II

Editors

Hamid R. Arabnia
Hassan Reza

Associate Editors

Lawrence Chung, Jose Luis Garrido
Emanuel Grant, Vince Schmidt
Ashu M. G. Solo
Nary Subramanian

WORLDCOMP'07
June 25-28, 2007
Las Vegas Nevada, USA
www.world-academy-of-science.org

©CSREA Press

Copyright © 2007 CSREA Press
ISBN: 1-60132-033-7, 1-60132-034-5 (1-60132-035-3)
Printed in the United States of America