An Assessment of Middleware Platforms for Accessing Remote Services

Young-Woo Kwon and Eli Tilevich
Dept. of Computer Science
Virginia Tech
{ywkwon,tilevich}@cs.vt.edu

William R. Cook
Dept. of Computer Sciences
The University of Texas at Austin
wcook@cs.utexas.edu

Abstract

Due to the shift from software-as-a-product (SaaP) to software-as-a-service (SaaS), software components that were developed to run in a single address space must increasingly be accessed remotely across the network. Distribution middleware is frequently used to facilitate this transition. Yet a range of middleware platforms exist, and there are few existing guidelines to help the programmer choose an appropriate middleware platform to achieve desired goals for performance, expressiveness, and reliability. To address this limitation, in this paper we describe a case study of transitioning an Open Service Gateway Initiative (OSGi) service from local to remote access. Our case study compares five remote versions of this service, constructed using different distribution middleware platforms. These platforms are implemented by widely-used commercial technologies or have been proposed as improvements on the state of the art. In particular, we implemented a service-oriented version of our own Remote Batch Invocation abstraction. We compare and contrast these implementations in terms of their respective performance, expressiveness, and reliability. Our results can help remote service programmers make informed decisions when choosing middleware platforms for their applications.

1 Introduction

The next couple of years will see a fundamental shift in how the average user takes advantage of computing resources. Traditional shrink-wrapped software applications will move in the direction of a computation model dominated by cloud computing [4, 19]. In this shift, the provisioning of software will evolve from software-as-a-product (SaaP) to software-as-a-service (SaaS). For example, a desktop application could be modified so that much of its execution takes place at a remote server in the cloud, with only the GUI rendered locally. The GUI part is likely to run on a mobile device, for example a smart phone.

Two levels of infrastructure are needed to realize this vision of software services. Firstly, component models are needed to define services and their interfaces. The Open Service Gateway Initiative (OSGi) [10] provides a platform for defining and managing components that can be used as services. It is used by developers to package features as components for separate deployment, and by end users to select components they need. Secondly, middleware infrastructure is needed to allow services to be accessed remotely. There are several different kinds of middleware, and each has different performance, expressiveness, and reliability characteristics. Middleware can be based on messaging, remote procedure calls, or remote evaluation, with the option of asynchronous processing. The trade-offs between these approaches have not been properly examined and, as a result, are poorly understood.

To address this lack of understanding, in this paper we describe a case study we have conducted to examine the trade-offs of using different middleware platforms of accessing services remotely. For the case study, we chose a realistic OSGi service that has been integrated into several commercial applications. This service is the Lucene search engine library [1] that provides functionality to index and search text files in Java. For the case study, we implemented a simple dictionary application that can search and return definitions, find synonyms, as well as suggest corrections for misspelled or partially-specified words.

We have implemented three Lucene-based services using five different middleware platforms: TCP sockets, synchronous and asynchronous remote calls in R-OSGi [12], Message Oriented Middleware (MOM) [1], and Remote Batch Invocation (RBI) [5]. For each implementation, we measured: (1) the total number of lines of uncommented code and its cyclomatic complexity, (2) the aggregate latency of invoking remote service methods, and (3) the degree of reliability of remote service methods in the presence of network volatility. The lines of code and its cyclomatic complexity are commonly used to assess the complexity and quality of a software artifact. The aggregate latency of in-

[1] http://lucene.apache.org/

voking a service indicates how long it takes for the clients to derive the service's expected benefits. This metrics comprehensively assesses the Quality of Service (QoS) from the end user's perspective. Finally, the ability of a remote service to cope with network volatility is critical to maintaining the required QoS in the majority of realistic network environments.

One of the evaluated platforms is our own Remote Batch Invocation (RBI), a middleware abstraction we have recently introduced [5]. In RBI, a batch is a collection of method calls, conditional statements, and loops that is transfered in bulk to the server, which executes the collection and returns the results to be assigned to local variables. Although RBI clients resemble traditional RPC clients, they have a fundamentally different, service-oriented execution model. As such, our implementation of OSGi in RBI is the first non-RPC implementation of the OSGi R4.2 specification, which codifies how OSGi bundles should be accessed remotely.

Based on the results of our case study, the technical contributions of this paper are as follows:

- The first non-RPC remote implementation of the OSGi R4.2 specification.
- A comprehensive evaluation of the trade-offs between the performance, expressiveness, and reliability of middleware platforms for accessing services remotely.
- A systematic analysis of the evaluation that can help inform a working programmer about which middleware platform should be used to access services remotely.

The rest of this paper is structured as follows. Section 2 introduces the concepts and technologies used in this work. Section 3 describes the implementation of OSGi in RBI. Section 4 describes our case study and its results. Section 5 discusses related work, and Section 6 presents future research directions and concluding remarks.

2 Background

In the following discussion we describe Service Oriented Architecture (SOA), OSGi, and middleware platforms, including R-OSGi and Message Oriented Middleware.

2.1 Service Oriented Architecture

Service Oriented Architectures (SOA) has been recently employed as a means of providing uniform access to a variety of computing resources across multiple application domains. In SOA, software components are provided as services, self-encapsulated units of functionality accessed through a public interface [11]. Essential characteristics of service-orientation are platform independence and support for stateless communication models.

Services can access each other only via each other's public interfaces. Loosely coupled services may be collocated in the same address space or be geographically dispersed across the network. Among the software engineering advantages of SOA are strong encapsulation, loose coupling, ease of reusability, and standardized discovery.

OSGi

The Open Service Gateway Initiative (OSGi) provides a platform for implementing services [10]. It allows any Java class to be used as a service by publishing it as a service bundle. OSGi manages published bundles, allowing them to use each other's services. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete stages) and allows it to be added and removed at runtime.

OSGi is a mature software component platform. It has been widely adopted by multiple industry and research stakeholders, organized into the OSGi Alliance. OSGi is used in large commercial projects, including the Spring framework [2] and Eclipse [3], which use this platform to update and manage plug-ins. The OSGi standard is currently implemented by several open-source projects, including Apache Felix [4], Knopflerfish [5], and Eclipse Equinox [6].

2.2 Distribution Middleware

Distribution Middleware provides mechanisms for software on one system to invoke operations on a remote system. Middleware eliminates the need for low-level network programming and offers convenient building blocks for constructing distributed systems. There are several different platforms used in middle ware applications, including messaging, remote procedure calls, and remote evaluation.

Message Oriented Middleware

MOM is an infrastructure for distributed communication using messages. Although originally all message based communication was presumed to follow the asynchronous interaction model, most MOM systems now support both synchronous and asynchronous interaction models. In addition, MOM provides two messaging models, *point-to-point* and *publish/subscribe*. In the point-to-point model, a sender sends messages to a particular client through a message

[2] http://www.springsource.org/
[3] http://www.eclipse.org/
[4] http://felix.apache.org/
[5] http://www.knopflerfish.org/
[6] http://www.eclipse.org/equinox/

queue. In the publish/subscribe model, a sender publishes messages to multiple clients through a message topic.

Java Message Service (JMS) [8] is a standard API from Sun Microsystems that enables Java programs to use message based communications. JMS is implemented in widely used MOM infrastructures, including Apache's ActiveMQ [7] and JBoss Messaging [8]. For the purposes of this paper, we evaluate the publish/subscribe model of ActiveMQ.

2.3 Remote Procedure Calls

Remote Procedure Calls (RPC) are the basis for a wide range of middleware implementations. In this model, each call to a remote interface is transfered from the client to the server for execution, and the results returned to the client. RPC has been extended to support object-oriented programming by introducing *object proxies*, which forward calls from client to server. This approach is the basis for DCOM [2] and CORBA [9].

Remote OSGi (R-OSGi) [12] is an RPC-based middleware platform for OSGi. The initial OSGi specification codifies inter-bundle communication as occurring within a single host. The R-OSGi distribution infrastructure allows accessing OSGi services remotely through a proxy-based approach, with proxies exposed as standard OSGi bundles. R-OSGi is based on RPC, but allows both synchronous and asynchronous calls, which can reduce latency. The distributed service registry of R-OSGi makes it possible to treat remote and local services uniformly.

More recently, the OSGi alliance released the OSGi R4.2 specification that describes how remote OSGi services can be discovered and used [10]. The OSGi R4.2 specification does not specify how remote OSGi services should be accessed. Instead, the specification codifies only how remote service interfaces should be discovered and retrieved. Once a remote service interface is obtained, it is up to the implementor of this specification how interface methods are to be invoked at a remote OSGi framework and how their results are to be transferred back to the caller.

The first reference implementation of R4.2 is D-OSGi [9], which implements the specification as Web services, using SOAP over HTTP for transmission and WSDL contracts for exposing services. This implementation is also RPC-based.

Although an RPC-based implementation naturally satisfies the method calling semantics of OSGi service interfaces, other middleware abstractions can also be used to implement R4.2.

[7] http://activemq.apache.org/

[8] http://www.jboss.org/jbossmessaging/

[9] http://cxf.apache.org/distributed-osgi.html

2.4 Remote Batch Invocation

Remote Batch Invocation (RBI) [5] is a distributed middleware abstraction based on partitioning blocks of code into remote and local parts, while performing all communication in bulk. Batches are specified using a *batch statement*. The body of a batch statement combines remote and local computation. In Java, a batch block looks like a collection of remote method calls but is executed using *remote evaluation* [15], in which all the remote calls are sent in a single *batch script*. In addition, data is moved in bulk between client and server. RBI differs from RPC in that the unit of distribution is a block of code rather than a single procedure call.

The details of RBI are discussed in the following section, which also shows how RBI can be used to provide remote access to OSGi services.

3 OSGi in RBI

RBI introduces a **batch** statement that executes multiple remote calls using a single remote round trip to the server. Figure 1 shows how the Lucene OSGi service can be accessed with RBI. Note that the **batch** block includes looping and conditional statements. The **batch** language extension is transformed into standard Java.[10]

```
1 batch (Lucene ls : new Service(Lucene.class)) {
2   final  TopDocs topDocs = ls.search(query);
3   StringBuffer  defBuffer  = new StringBuffer();
4   for  (ScoreDoc hits : topDocs.scoreDocs) {
5    Document doc = ls.doc(hits.doc);
6    if  (doc != null)  {
7     defBuffer.append(doc.getValues(DEFINITION));
8   } } }
```

Figure 1. Example of batch invocation.

The RBI runtime executes multiple calls (combined with conditional and looping constructs) to a given remote service. Finally, RBI/OSGi does not require any changes to remote service interfaces, which are discovered and bound using a standard OSGi registry.

3.1 RBI Runtime System

The runtime architecture of RBI, shown in Figure 2, consists of a service consumer, service provider, batch processor, and distribution provider. Once the service provider registers a service in the OSGi framework, the distribution

[10] Please refer to our ECOOP 2009 papers for translation details [5].

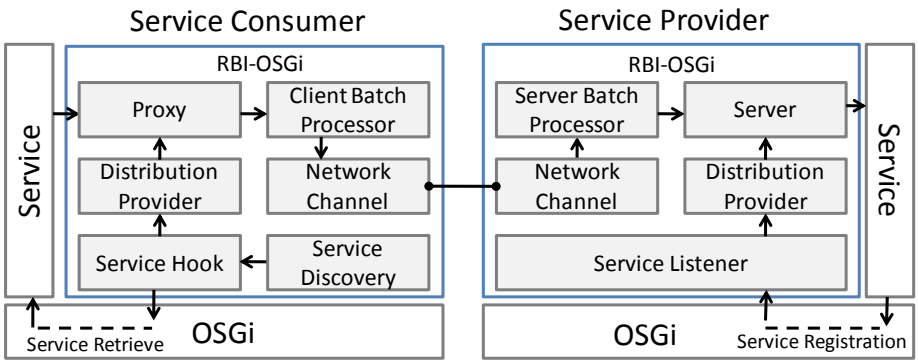

Figure 2. OSGi with RBI Architecture.

provider instantiates a server that can be accessed remotely. The service consumer discovers and retrieves the remote service, and then the distribution provider creates a proxy for importing the service. Upon the service consumer making remote calls, the batch processor aggregates them into a single descriptor, which is transmitted across the network to the service provider. The service provider's batch processor interprets the descriptor, invoking the appropriate service methods, and sends the results back to the service consumer.

To integrate OSGi with RBI, we connected RBI to the standard OSGi services, `Service Listener` and `Service Hook`. Once a `Service Listener` is registered with OSGi, it starts receiving lifecycle change events for the registered service. The distribution provider uses a `Service Listener` to determine when a server must be instantiated to process remote requests. The `Service Hook` service, introduced only in the OSGi R4.2 specification, intercepts service events, raised in response to the service consumer retrieving the remote service, and creates a proxy for accessing services remotely.

The `Service Hook` service makes it possible to treat local and remote services uniformly, with the only difference concerning their configuration. In other words, switching from using the local version of a service to a remote version and vice verse does not require any source code changes, which are confined to configuration files. Because the OSGi R4.2 specification requires that remote service interfaces be decoupled from their implementations, the `Service Hook` service accomplishes that by making it possible to switch implementations through a simple configuration file change.

4 Case study

To compare different middleware platforms, we compared remote access to a set of three services packaged as an OSGi bundle. We chose the Lucene search engine library, which is distributed as an OSGi bundle, thus providing a service interface. Using Lucene, we implemented three services to search for (1) a word's definition, (2) a word's list of synonyms, and (3) a list of spelling suggestions for a misspelled word. Note that service (2) extends the functionality of service (1), and service (3) extends the functionality of service (2). Thus, service (2) includes all the functionality of service (1), and service (3) includes that of services (1) and (2).

For our case study, we examined how these services can be accessed remotely using five different middleware platforms. To that end, we compared each of the five implementations in terms of their respective performance, expressiveness, and reliability.

For the purposes of this study, we define our metrics as follows:

- **Performance:** the total execution time it takes to execute a service, including both network latency and business processing.
- **Expressiveness:** ease of implementation, measured by the total of Uncommented Lines of Code (ULOC) it takes to write the service, and their McCabe cyclomatic complexity (MCC) [7].
- **Reliability:** the ability to withstand temporary network volatility, when the communication network experiences an outage.

In this benchmark, we compare these metrics for five middleware platforms: (1) synchronous R-OSGi, (2) asynchronous R-OSGi, (3) Message-Oriented Middleware, (4) raw sockets, and (5) our own RBI interface to OSGi.

4.1 Experimental Setup

All the experiments were conducted on the client machine running 3.0 GHz Intel Dual-Core CPU, 2 GB RAM, Windows XP, JVM 1.6.0 13 (build 1.6.0 13-b03), and the

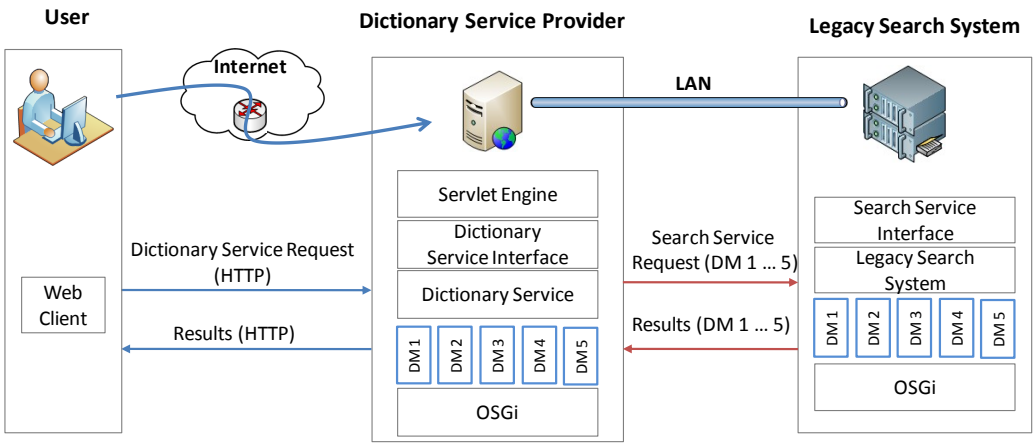

Figure 3. Dictionary System.

server machine running 1.8 GHz Intel Dual-Core CPU, 2.5 GB RAM, Windows 7, JVM 1.6.0 16 (build 1.6.0 16-b01, connected via a local area network (LAN) with a 100Mbps bandwidth, and 1ms latency.

Figure 3 depicts a diagram describing the specifics of our experimental setup. The Lucene OSGi bundle is located on a separate node (server) and is accessed remotely from another node (client). To start the benchmarking of a given setup, we constructed a simple Web client that communicates with the client node through HTTP. By navigating a Web browser to a URL associated with any of the five middleware implementations, a servlet at the client node invokes its corresponding benchmark method.

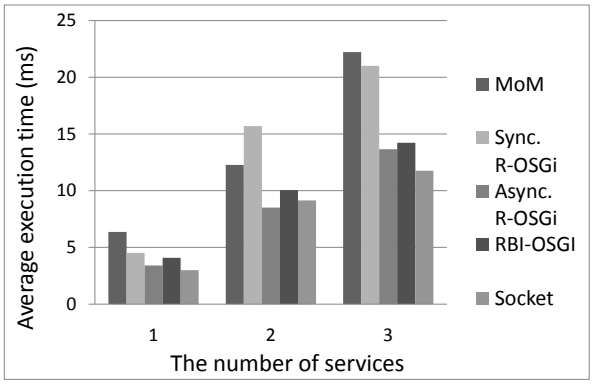

Figure 4. Performance Comparison.

4.2 Performance

Each benchmark method calls three services in sequence, repeating each service call 1,000 times and then reporting the averaged time. Only the time to invoke the Lucene-based services is taken into account, while the HTTP communication to trigger different benchmarks is omitted.

Figure 4 shows the averaged performance for each service. Because each of the three services takes an increasing number of remote roundtrips, for each middleware platform, the total execution time grows for services 2 and 3.

For each service, raw sockets provide the best performance. Asynchronous R-OSGi comes close second. RBI/OSGi using synchronous communication comes quite close to asynchronous R-OSGi. Synchronous R-OSGi is always slower than RBI/OSGi, due to the latter middleware platform aggregating multiple remote calls and invoking them in bulk.

Surprisingly, our MOM-based implementation consistently showed the poorest results across all benchmarks. The reason is because the implementation we used, ActiveMQ, is based on a publish-subscribe rather than peer-to-peer communication model. Publish-subscribe models are beneficial when messages have to be broadcast to a large number of recipients. In our setup, when using MOM for client-server communication, the overhead of involving a message queue was never amortized.

4.3 Expressiveness

Table 1 shows the total uncommented lines of code (ULOC) it takes to implement each of the three services using different middleware platforms. It also shows their McCabe Cyclomatic Metric (MCC). The ULOC numbers in Table 1 combine the client and server portions, while excluding 1918 ULOC that it takes to implement the functional processing part of all the remotely-accessed services.

As expected, our sockets-based implementation is the longest. A programmer has to design and express a low-

Table 1. Expressiveness Comparison.

Middleware platform	Service	ULOC	Max. MCC
Sync. R-OSGi	Service 1	14	7
	Service 2	14	10
	Service 3	14	17
Async. R-OSGi	Service 1	148	8
	Service 2	170	12
	Service 3	212	25
RBI/OSGi	Service 1	23	7
	Service 2	27	10
	Service 3	33	17
MOM	Service 1	1172	8
	Service 2	1207	13
	Service 3	1231	23
Sockets	Service 1	2722	8
	Service 2	2793	13
	Service 3	2839	23

level communication protocol, which also includes the format for each transferred message. In addition, avoiding deadlocks and ensuring good performance requires that message sending and receiving be handled by different threads.

The MOM implementation is the second longest. A programmer has to implement a listener interface and register it with the messaging system and handle messages that arrive out of order. In addition, the programmer must define the messages and process them at the application level.

Asynchronous R-OSGi follows next. A programmer also has to implement a listener, but R-OSGi eliminates the need for the programmer to implement messages and setup the communication.

The RBI/OSGi implementation takes about an order of magnitude fewer lines of code than the asynchronous R-OSGi one. RBI/OSGi is a method-based middleware mechanisms that does not require the programmer to write any communication-specific code.

The synchronous R-OSGi implementation takes about the same amount of code as that of RBI/OSGi. RBI adds a couple of lines of code to setup and express a batch.

MCC metrics is indicative of the programming effort required to understand a codebase. As expected, the raw sockets, asynchronouns R-OSGi, and MOM implementations have high MCC, while synchronous R-OSGi and RBI/OSGi ones have lower MCC.

4.4 Reliability

As it turns out, only our MOM-based implementation has built-in fault tolerance capabilities provided by ActiveMQ. It can operate in what is called "persistent mode" that stores every message to be sent in stable storage. Upon disconnection, the undelivered messages are rescheduled for delivery after the network becomes reconnected.

Table 2. Reliability Comparison.

Middleware platform	Fault handling	3rd party solution
Sync. R-OSGi	N/A	DR-OSGi
Async. R-OSGi	N/A	DR-OSGi
RBI/OSGi	N/A	DR-OSGi
MOM	built-in	N/A
Sockets	N/A	N/A

If reliability in the face of network volatility is required, Table 2 summarizes how fault handling mechanisms can be adopted in each middleware platform. When a middleware mechanism lacks built-in facilities for dealing with network volatility, our recent research has shown how such facilities can be factored into a middleware infrastructure [6].

4.5 Discussion

Here we discuss some of the implications of the performance, expressiveness, and reliability measurements presented above. In our discussion, we attempt to provide specific recommendation for the developers of service-oriented applications.

Figure 5 depicts the trade-offs between the performance, expressiveness, and reliability guarantees offered by each middleware platform. Because no platform satisfies all three guarantees, programmers should choose an appropriate platform having considered the immediate needs of their service applications.

Threats to Validity The measurements above are subject to both internal and external validity threats. The internal validity is threatened by the way in which we chose to implement our subject services by using different middleware platforms. In our daily programming practices, we do not regularly use all of the five platforms. Therefore, the way we chose to implement our service may not be fully optimal, in terms of using the proven design patterns. We believe, however, that our programming practices are representative of that of the common programmer.

The external validity is threatened by our choice of an existing OSGi bundle to be accessed remotely. OSGi public interfaces have been carefully designed to be coarse-grained, and more naively-designed service interfaces can have finer granularity. In that case, the performance disparities between synchronous R-OSGi and the asynchronous alternatives would be even more pronounced.

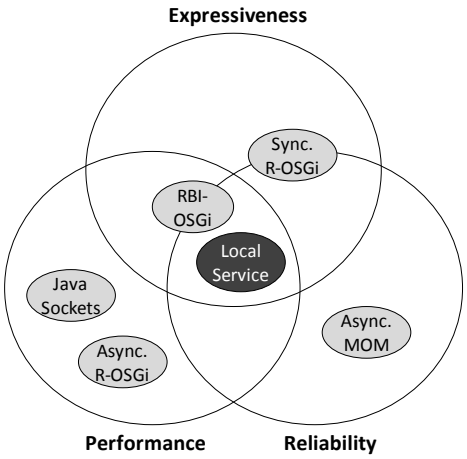

Figure 5. Trade-offs between the expressiveness, performance and reliability levels.

Performance Even coarser grained service interfaces cannot completely eliminate latency concerns. As our measurements show, asynchronous communication leads to better performance. Unfortunately, business logic may require synchronous service calls. Our RBI/OSGi platforms can reduce the aggregate latency of multiple remote service calls without asynchronous processing.

Expressiveness Despite their performance advantages, asynchronous designs tend to be more complicated, taking more code that is more complex to express. RPC-based abstractions, including our own RBI/OSGi, are more straightforward to implement and understand.

Reliability The reliability of a distributed application is dependent on the reliability of its constituent components, which include both the execution units implementing the application's functionality and the network connecting them. One can argue that the ULOC metrics is inversely proportional to the level of reliability of an individual software component. If the probability of a bug can be expressed in terms of the lines of code and its complexity (e.g., $X\%$ that a software defect exists within N lines of code), then shorter and less complex implementations are less likely to contain bugs. In the light, our ULOC and cyclomatic complexity metrics can also serve a double duty as local reliability metrics.

With respect to distributed execution, the common wisdom of distributed system development suggests that reliability is best implemented on a per-application basis. There is value, however, in handling system-level errors at the middleware level. In that light, using MOM leads to applications that can withstand temporary network disconnections. Such fault-tolerance capacities can be factored into existing systems, as demonstrated recently [6].

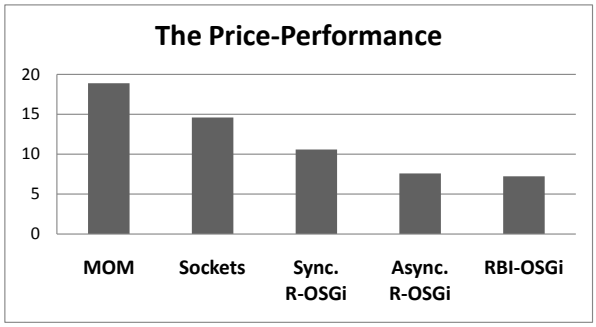

Figure 6. The price-performance ratio comparison.

Price-Performance Ratio So far, we compared our different middleware platforms using a single metrics. To obtain deeper insights, we introduce a new metrics, *price-performance*, represented by the following

$$PP = \frac{RULOC/LULOC}{LET/RET}$$

where $RULOC$ and $LULOC$ are local and remote uncommented lines of code, respectively; and LET and RET are local and remote execution times, respectively. The minimum *price-performance* ratio is 1, which can only be achieved when no distribution is present. In other words, the *price-performance* ratio is minimized when its numerator and denominator are approaching 1. Since $LULOC$ and LET are fixed, only RET and $RULOC$ can affect the ratio.

Figure 6 shows that MOM has the largest *price-performance* ratio, followed by sockets, synchronous R-OSGi, asynchronous R-OSGi, and RBI/OSGi. The *price-performance* ratio of MOM is most likely not fully representative; our benchmark does not exercise the advanced features of ActiveMQ (i.e., efficient broadcasting of messages to multiple receivers). If standard middleware must be used, asynchronous RPC (i.e., as in R-OSGi) seems to minimize *price-performance*. Based on this analysis, RBI/OSGi represents a highly-promising alternative to standard middleware, offering a low *price-performance* ratio along with an intuitive programming model.

5 Related Work

Remote Procedure Call (RPC) [17] has been one of the most prevalent communication abstractions for building distributed systems, but its shortcoming and limitations

have been continuously highlighted [16, 21, 14]. Some experts even claim that RPC has been harmful in terms of its influence on distributed systems development, and argue that a different communication abstraction becoming dominant instead would have been beneficial [20]. Asynchronous messaging and events, including publish-subscribe platforms [3], are frequently mentioned as superior alternatives to RPC.

As confirmed by our study, exposing distributed functionality through a familiar procedure call paradigm of RPC and its object-oriented counterparts provides expressiveness and ease of implementation advantages. Our RBI/OSGi abstraction attempts to address some of the limitations of RPC, but to retain its advantages without the complexities of asynchronous processing incurred by message- and event-based abstractions.

Some research has evaluated MOM and JMS implementations in terms of their respective performance, scalability, and reliability [18, 13]. This research pursues a similar objective by evaluating five different middleware platforms to help developers choose an appropriate platform for accessing services remotely.

6 Conclusion

Due to the advantages provided by services, SaaS has entered the mainstream of commercial software development and a growing percentage of computing functionality is becoming accessible as a service. The programmers who need to access remote services are faced with the challenges of choosing an appropriate middleware platform for the task at hand. To assist the programmers in their decision process, in this paper, we described a case study that compared the performance, expressiveness, and reliability of five different middleware platforms for accessing services remotely. Our measurements and analysis not only help the programmers in choosing between different middleware platforms, but also can inform the design of new abstractions for accessing services remotely.

References

[1] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing*, pages 1–18. Springer-Verlag London, UK, 1999.

[2] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

[3] C. Damm, P. Eugster, and R. Guerraoui. Linguistic support for distributed programming abstractions. In *Proceedings of the* 24^{th} *International Conference on Distributed Computing Systems*, 2004.

[4] B. Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[5] A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In *The* 23^{rd} *European Conference on Object-Oriented Programming*, July 2009.

[6] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong. DR-OSGi: Hardening distributed components with network volatility resiliency. In *Proceedings of the ACM/IFIP/USENIX* 10^{th} *International Middleware Conference*, Urbana, IL, USA, December 2009.

[7] T. J. McCabe. A complexity measure. In *Proceedings of the* 2^{nd} *International Conference on Software Engineering*, Los Alamitos, CA, USA, 1976.

[8] R. Monson-Haefel and D. Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[9] Object Management Group. The CORBA component model specification. Specification, Object Management Group, 2006.

[10] OSGi Alliance. OSGi service platform release 4.2 specification, 2009.

[11] M. P. Papazoglou and W.-J. V. D. Heuvel. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology*, 2(4):412–442, 2006.

[12] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX* 8^{th} *International Middleware Conference*, Newport beach, CA, USA, November 2007.

[13] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the specjms2007 benchmark. *Performance Evaluation*, 66(8):410 – 434, 2009.

[14] U. Saif and D. Greaves. Communication primitives for ubiquitous systems or RPC considered harmful. In *Proceedings of the* 21^{st} *International Conference Distributed Computing Systems Workshop*, 2001.

[15] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Trans. Program. Lang. Syst.*, 12(4):537–564, 1990.

[16] A. S. Tanenbaum and R. v. Renesse. A critique of the remote procedure call paradigm. In *EUTECO 88*, 1988.

[17] B. Tay and A. Ananda. A survey of remote procedure calls. *Operating Systems Review*, 24(3):68–79, 1990.

[18] P. Tran, P. Greenfield, and I. Gorton. Behavior and performance of message-oriented middleware systems. In *Proceedings of the* 22^{nd} *International Conference on Distributed Computing Systems*, pages 645–654, Washington, DC, USA, 2002.

[19] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[20] S. Vinoski. RPC under fire. *IEEE Internet Computing*, pages 93–95, 2005.

[21] J. Waldo, A. Wollrath, G. Wyant, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1994.