NETW 696
2844

J2EE-based Middleware for Low Latency Service Enabling Platforms

Bruno Van Den Bossche, Filip De Turck,
Bart Dhoedt, Piet Demeester
Ghent University - IBBT - IMEC
Department of Information Technology
Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium

Gerard Maas, Johan Moreels,
Bert Van Vlerken, Thierry Pollet
Alcatel
Research & Innovation
Copernicuslaan 50, 2018 Antwerpen, Belgium

***Abstract*—While the Java programming language and the J2EE platform are increasingly popular for implementing business logic on backend platforms, new emerging Java technologies such as JAIN SLEE and SIP Servlet are focusing on the development of low latency Java applications. As J2EE mainly focuses on enterprise applications with complex long lasting transactions, this technology is considered unsuitable for applications with low latency and high throughput characteristics.**

This paper compares these telecom oriented Java technologies to J2EE both in terms of functionality and through a detailed performance evaluation. JVM performance tuning has been studied as well and is explained in the paper. We performed a SIP Proxy benchmark with strict low latency requirements of which the results are presented. Furthermore, design guidelines for J2EE applications are discussed to optimize for low latency behavior together with an interpretation of the obtained performance results.

I. Introduction

Java and J2EE are increasingly popular for developing large scale backend applications. An important advantage of using J2EE is that it simplifies managing complex transactions, allows for a fast development of complex applications and it is platform independent. Another feature of Java with important implications is the use of a garbage collector. Java includes automatic memory management (garbage collection) as a part of the Java runtime. This means that very common errors made by developers related to memory management cannot occur. Since the garbage collection is part of the Java runtime it is not completely under the control of the application developer. One of the side effects of the garbage collection is that the application execution can be paused, at unpredictable times, to allow for garbage collection. These pauses are highly undesirable when dealing with low latency services such as Voice over IP (VoIP) offerings. Thus, the question arises whether Java in general, and J2EE in particular are suitable for applications with strict low latency requirements.

Apart from J2EE two new Java Application Frameworks have emerged, namely JAIN SLEE and SIP Servlet. Both are part of the Java APIs for Integrated Networks (JAIN [1]) which provide us with an extensive set of standardized APIs to facilitate the development and deployment of telecom services. Telecom applications often have very strict requirements regarding throughput (e.g. the number of VoIP call setups a softswitch can process per second) and latency (e.g. the setup of a call should be very fast).

In this paper we will evaluate SIP Servlet as well as JAIN SLEE and compare them against J2EE, which was not originally designed for this type of applications, but has the advantage it is a mature, well known technology.

The structure of this paper is as follows: firstly the Session Initiation Protocol (SIP) and the use case for the benchmark is briefly explained in Section II. Next a description of the technologies is presented in Section III, highlighting the different architectures and features. A more detailed description of the J2EE implementation of the selected use case is given in Section IV. Section V details the test setup used, followed by the actual test results and design guidelines in Section VI. Final conclusions and future work are discussed in Section VII.

II. Low Latency Services

A. Motivation

Easy deployment of services and combining existing services is getting more important in current software and platform architectures. For example the IP Multimedia Subsystem (IMS) is an open standardized multi-media architecture for mobile and fixed IP services [2]. It is a VoIP implementation based on a variant of SIP, and runs over IP. The aim of IMS is not only to provide new services but to provide all the services, current and future, that the Internet offers. Massively Multiplayer Online Gaming will be an important domain requiring low latency and high throughput capabilities. Online virtual worlds will become more pervasive and are already stretching out into the real world through auctions of in-game properties and game characters contacting you in real life thus having the same requirements of existing telecom applications. Convergence of these types of applications are a domain where Service Enabling Platforms can play an important role. In this paper we evaluate a VoIP use case with strict low latency and high throughput requirements.

B. Session Initiation Protocol (SIP)

The Session Initiation Protocol is defined in RFC 3261 [3] and describes an application-layer control (signaling) protocol for creating, modifying and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences. RFC 3261 also defines the use of SIP proxies and how they should interact with other SIP applications and proxies.

The scenario used for testing is the Proxy 200 test, shown in Figure 1 and defined in the SIPstone benchmark [4]. We are interested in the time it takes to set up a call. This is the time it takes from the initial INVITE of Alice till she receives an OK from Bob, indicating the call is set up. Subsequently Alice will send an acknowledgment to Bob saying she received the OK and the media session (e.g. voice or video conference) can start. When benchmarking the call was immediately terminated by the caller and no media session was initiated.

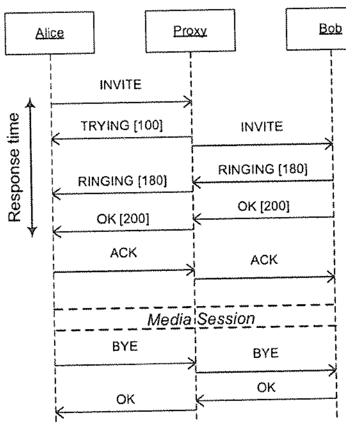


Fig. 1. Proxy 200 test used for the performance evaluation of the discussed technologies

The SIP Proxy use case was chosen as it is an important step in setting up a typical SIP Session and the requirements are very well defined. Both low latency requirements and high throughput requirements can be evaluated using this example.

III. Technology Description

All technologies discussed are component based and offer a container for the applications to be deployed and run in. A major function of the container is the life cycle management of the application (i.e. starting and stopping the application or application components). Also a number of non-functionals are delegated to the container in stead of the actual application (e.g. logging, authentication, management of external resources etc).

The global overview of a container managed framework is shown in figure 2. In the container multiple applications and/or application components are deployed. These components can interact with each other and other data sources, such as a database. Clients may interact directly with the application or may interact with the application container through the use of "Resource Adapters" (RAs) which manage communication with the outside of the container.

The following sections describe each application framework and a comparative overview is presented in table I. For each platform a SIP Proxy implementation has been evaluated. A detailed description of the J2EE implementation is given in Section IV.

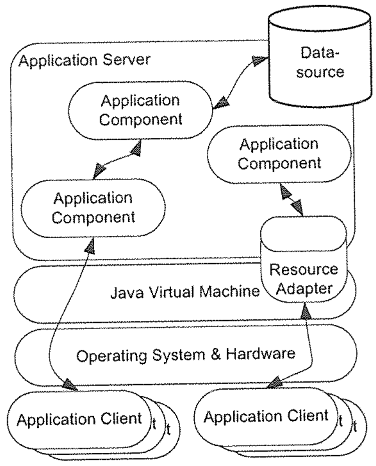


Fig. 2. Container based architecture that all of the discussed technologies have in common.

A. J2EE

A typical J2EE Application Server consists of both an Application Container and a Web Container. The Web Container hosts all web related application components such as HTTP Servlets, Java Server Pages, etc. The application container hosts J2EE applications composed of Enterprise Java Beans (EJBs). There are three type of EJBs: Session Beans which usually contain business logic, Entity Beans which represent data and Message-Driven Beans which allow for asynchronous communication.

J2EE supports communication with clients through its web container, using HTTP, or application clients can interact directly with application components using a J2EE application client which performs Remote Method Invocation. Apart from this there is also support for deploying J2EE Connector Architecture Resource Adapters (RA) [5] into the application server. Through the use of these RAs, which can be deployed on any J2EE application server, it is possible to extend the application server and support extra methods of communication. A well known example using the JCA is the Java Message System (JMS). The J2EE Connector Architecture was originally designed for implementing RAs to interface with existing (legacy) systems, but it can be used in a more general context to extend the J2EE application server with extra protocol stacks.

B. SIP Servlet

The primary goal of the SIP Servlet API [6] is to simplify the development of SIP enabled applications. By using the existing servlet architecture it is relatively easy for developers who are familiar with HTTP Servlets to create SIP enabled applications. For each type of SIP Message a method is defined to handle it. One example is a "doInvite" method which will

TABLE I

COMPARISON OF THE MAIN CHARACTERISTICS OF J2EE, SIP SERVLET AND JAIN SLEE

	J2EE	SIP Servlet	JAIN SLEE
Design Goal	Manage complex business transactions and data management.	Simplify SIP development.	Applications which require high throughput and low latency event processing.
Architecture	Component based, Object Orientated architecture. Unit of logic is the EJB Support for composition and reuse	Based on HTTP Servlets. Unit of application logic is the Servlet. No standard model for composition and reuse.	Component based, Object Orientated architecture Unit of logic is the SBB Support for composition and reuse
Protocol Support	Limited to RMI and HTTP by default. But can be extended through the use of the J2EE Connector Architecture.	Limited to SIP and HTTP. Can not be extended by application programmers.	No protocols supported (protocol agnostic) out of the box. Any protocol can be added through the use of Resource Adapters.
Communication	Optimized for synchronous request-response model. Support for event driven logic through Message-Driven Beans.	Optimized for synchronous request-response model.	Optimized for event driven logic. Limited Support for synchronous request-response model.
Clustering	Vendor specific extensions for clustering multiple application servers.	Vendor specific extensions for clustering multiple application servers.	Replication in cluster defined in the specification.

handle SIP INVITE messages. This technology is targeted specifically toward SIP applications and is less generic than J2EE or JAIN SLEE.

In the container multiple applications can be deployed, each consisting of one or more SIP Servlets. Each application also contains a deployment descriptor which describes the application and tells the container which SIP message it should direct to which servlets. The servlets can then process the SIP messages and if necessary pass them on to other servlets. Furthermore multiple servlets may process the same SIP message.

The proxy implementation used for evaluating the platform consists of one SIP Servlet and made use of the proxy component part of the SIP Servlet specification. The SIP communication is completely managed by the application server and does not require any additional components.

C. JAIN SLEE

The JAIN Service Logic Execution Environment (SLEE) specification [7] provides an application server tailored for telecom. Applications are composed of Service Building Blocks (SBBs) which are the equivalent of the J2EE EJBs. One of the key features of the JAIN SLEE application container is the use of asynchronous communication. Internally almost all communication in the SLEE happens by using events. The idea behind the event based communication between SBBs is that every SBB performs its own task and then hands the result off to the next SBB in line. One could compare this to an assembly line in a factory

The internal routing is completely taken care of by the application server. A key component in the routing of events is the Activity Context which manages the links between logically connected SBB entities. When an initial event (e.g. a SIP INVITE) enters the SLEE an Activity Context is created. The SBBs processing this event will be attached to this Activity Context and all following events which logically belong together (e.g. all events related to the same SIP call) will be fired on the same Activity Context and processed by the same SBB entity. This is important as the SBB entities can share data on this Activity Context.

Communication with the outside world happens through RAs. For example we have an RA for SIP related communications which accepts incoming SIP messages, parses them and turns them into events understandable by the SLEE. Through the use of RAs the SLEE can be protocol agnostic. This means that any protocol can be supported by adding an appropriate RA. To perform the SIP Proxy test a SIP Resource Adapter was deployed in the application server together with a proxy SBB. The proxy implementation consisted of one SBB which accepted incoming SIP events and performed the necessary routing.

IV. J2EE SIP PROXY ARCHITECTURE

This section will give a detailed overview of the SIP Proxy Design in J2EE. To build a functional proxy, two key components are required. A Resource Adapter to extend J2EE with SIP capabilities and one or more components to implement the actual proxy logic. An overview of the J2EE SIP Proxy application is given in figure 3.

A. SIP Resource Adapter

Implementing an RA requires fulfilling a number of system level contracts regarding the management of connections, transactions and security. Furthermore, a listener interface needs to be defined which listening application components (Message-Driven Beans) have to implement to receive incoming events. If required, custom event types can be defined as well. In order to send outgoing messages a connection interface must be defined which application components (an J2EE component type) can use to connect to the RA.

To implement the SIP RA we used the publicly available JAIN SIP [8] stack and exposed its API to the J2EE application components. It is however possible to insert an extra layer of abstraction and define a new API for SIP communications, just like SIP Servlet does.

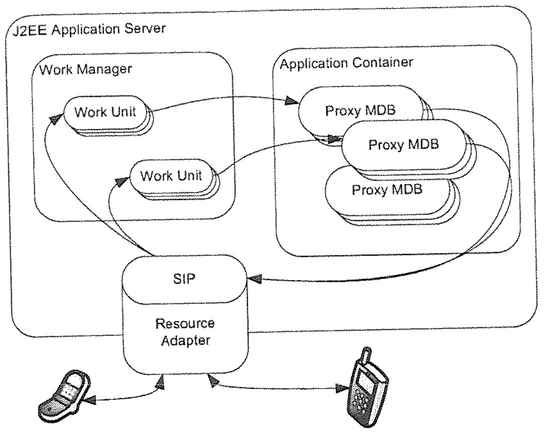


Fig. 3. Application architecture of J2EE SIP Proxy implementation. The RA accepts SIP Messages and uses the Work Manager for multi-threading. The Proxy MDB can then process the messages and uses the RA to send out SIP Messages again.

B. *Proxy Message-Driven Bean*

The proxy logic of the implementation is embedded in a Message-Driven Bean. This component registers itself to the SIP RA and listens for any incoming SIP Messages. Upon receiving a SIP Message it is determined which call it is part of and the necessary routing is performed. Any responses or forwarded messages are sent using the SIP RA.

Message Driven Beans are stateless components and can easily be pooled by the application server. Hence, if it is necessary to maintain a certain state over multiple SIP Messages it is required to keep this state manually. This can be done either on the application side or inside the RA, which then needs to provide the necessary methods to access this information. In the current implementation all state considering the SIP Sessions is maintained by the SIP RA.

C. *Event Filtering*

Based on the description so far, all SIP Messages that arrive at the SIP RA will be sent to every application component. This is usually not desirable and it can be a cause of overhead. Furthermore, not every application or service is interested in receiving every SIP Message. A billing component for example only needs to know when a call is started and when it is ended. All intermediate messages, are of no use and can be discarded.

To allow this, an Event Filter was implemented which accepts regular expressions that are added to the applications deployment descriptor and can be used to filter out any unwanted messages.

V. Test Setup

Before we discuss the obtained results, a short overview of the test setup used, is presented.

A. *Software Setup*

For benchmarking purposes SIPp [9] is used. SIPp is a free Open Source test tool/traffic generator for the SIP protocol. It allows generating SIP traffic and establishing and releasing multiple calls. It can also read custom XML scenario files, describing from very simple to complex call flows. It includes a few basic SIPstone defined test setups. All tests in this paper were performed using SIPp and the previously specified scenario.

B. *Hardware Setup*

All tests were performed using a dual Opteron 242 (1.6GHz) HP DL 145 with 2GB of memory for the proxy. The two clients were run on AMD athlonXP 1600+ machines with everything interconnected in a 100Mbit switched ethernet network. All platforms were running Debian GNU/Linux with a 2.6 kernel and the Sun JDK 1.4.2.

C. *Platform Tuning*

For all the presented test results extensive tuning of the JVM and garbage collector was performed. Without tuning of the JVM the garbage collector initiated pauses in the execution of the application which resulted in calls to timeout, even at low call rates. With appropriate tuning, these pauses can be minimized and thus the obtained results significantly improved. The basic set of tuning options used for all platforms is shown in table II. Detailed results of JVM tuning were previously reported on in [10].

-Xmx512m	(1)
-XX:NewSize=32m	(2)
-XX:MaxNewSize=32m	(3)
-XX:MaxTenuringThreshold=0	(4)
-XX:SurvivorRatio=128	(5)
-XX:+UseParNewGC	(6)
-XX:+UseConcMarkSweepGC	(7)
-XX:+CMSIncrementalMode	(8)
-XX:+CMSIncrementalPacing	(9)
-XX:CMSIncrementalDutyCycleMin=0	(10)
-XX:CMSIncrementalDutyCycle=10	(11)

TABLE II
Virtual Machine Tuning options for low latency behavior.

The options specified in table II are specific to the Sun JVM, but other virtual machines offer similar tuning options which can be used to achieve the same effect. The memory managed by the virtual machine is divided into multiple generations (Young, Tenured and Perm), depending on the age of the objects. As objects live longer they are moved into the next generation after a certain amount of time or a number of garbage collections. By specifying the sizes of the generations (1-3) and limiting the amount of time before an object is promoted to the next generation (4-5), we can achieve that objects lasting the whole call are moved to an older generation very fastly. This is beneficial as the older generations do not need to be garbage collected as often since the the majority of objects die very young. The garbage collector itself can also be tuned (6-11) to use multiple threads on multi-cpu machines and to work concurrently with the application execution for as long as possible. This allows to limit the time the execution of the virtual machine needs to be paused completely for garbage collection.

VI. Evaluation Results

This section gives an overview of the obtained test results. All platforms were submitted to a number of subsequent test runs that allowed us to evaluate and interpret the obtained data. Although setting up a session using SIP is not bound by the same latency requirements of the VoIP media session itself, it is necessary to achieve a low latency behavior as typically 6 to 7 hops, such as proxies, need to be traversed in the network. Considering additional network delays when setting up a SIP call using the specified scenario (see figure 1), 95% of the acknowledgments for the INVITE messages should arrive within 50ms and 50% should arrive within 25ms. Furthermore, design guidelines are proposed to optimize the architecture and implementation of J2EE applications for low latency behavior.

A. *Performance Results*

Figure 4 shows the performance results of the tested application servers. The graphs show the average response time, the 50th percentile and the 95th percentile plotted against the average cpu load at the given call rate. As all tests were performed on a dual cpu machine the cpu load is the average of the load of the two cpus during the test.

During each test every call rate, starting at 10 calls per second (caps) was run for 5 minutes. Then there was a pause of 1 minute after which the subsequent call rate was tested. The call rates were increased for as long the system under test could sustain the tested call rate. Each such test run was repeated three times to check consistency among the different test runs. Before every test run a dummy run was performed to allow the JVM to "warm up". This is necessary as the Virtual Machine will perform internal optimizations, try to reuse allocated memory resources etc. which could influence the measurements.

There is a significant difference in the maximum call rate all the application servers can sustain that lies within the specified requirements. For J2EE the maximum achievable call rate is approximately 150 caps (at higher call rates some calls did timeout), for JAIN SLEE this is 190 caps (at higher call rates the latency requirements are not met anymore) and for SIP Servlet this is 240 caps (at higher call rates some calls did timeout). A general remark is that all application servers are able to meet the low latency requirements for a certain call rate. At a call rate of 150 caps, J2EE even shows the best results regarding latency, although it is supposed to be the slowest technology.

It is important to note that with the J2EE test the load was not equally distributed over the two cpus. We assume this is due to limitations of the SIP stack used. The implementation of the JAIN SIP Stack used is a reference implementation, not optimized for production use. Tests, previously reported on in [11], show that an example proxy implementation based on the same stack could only sustain a call rate of approximately 30 caps. This is also the reason why the cpu load does not increase above 70% as the SIP Stack is overloading one cpu. In the tests with the other application servers the load was distributed more equally over the two cpus.

Figure 5 shows the response time distribution for a call rate of 100 caps for each tested platform. The ceiling for the number of calls completed, plotted on the Y-axis, has been limited to a maximum of 1000 calls for clarity. For all platforms the majority of the calls is answered within a few milliseconds. This also shows in figure 4.

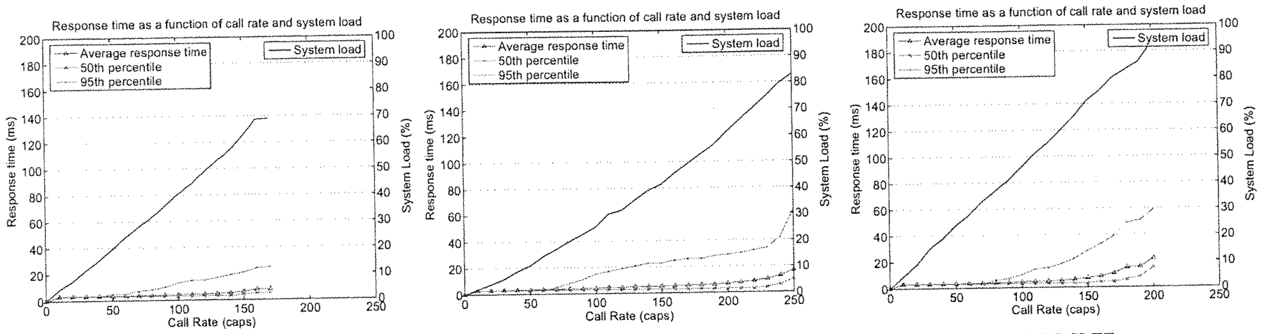
The tail of the distribution does show some distinct differences between the tested platforms. For J2EE (figure 5(a)) it is very short and the vast majority of the calls is answered within 30ms. For JAIN SLEE (figure 5(c)) and SIP Servlet (figure 5(b)) the tail is much longer, however they are capable of sustaining higher call rates. A possible cause for this behavior is the J2EE Server used as different implementations could have an influence on the application latency. However, we verified the results by deploying the SIP RA on other J2EE servers which performed similarly, excluding the J2EE server as a possible cause of this result. Our feeling is that the shorter tail of the distribution is caused by a difference in the implementation of SIP Resource Adapter. The management of SIP transactions and SIP dialogs is all included in the J2EE SIP RA, whereas this is (partially) handled by application components for SIP Servlet and JAIN SLEE.

B. *J2EE Design Guidelines*

An RA is responsible for its own thread and resource management, it is in fact the only J2EE software component allowed to create and manage threads by itself. However, it is suggested to use existing features such as the Work Manager which allows the RA to submit Work units. These Work units are then scheduled and executed automatically, not requiring the RA to explicitly manage the actual threads. As much actual processing possible should be done inside Work units and the processing in the RA should be limited to the receiving and sending of messages.

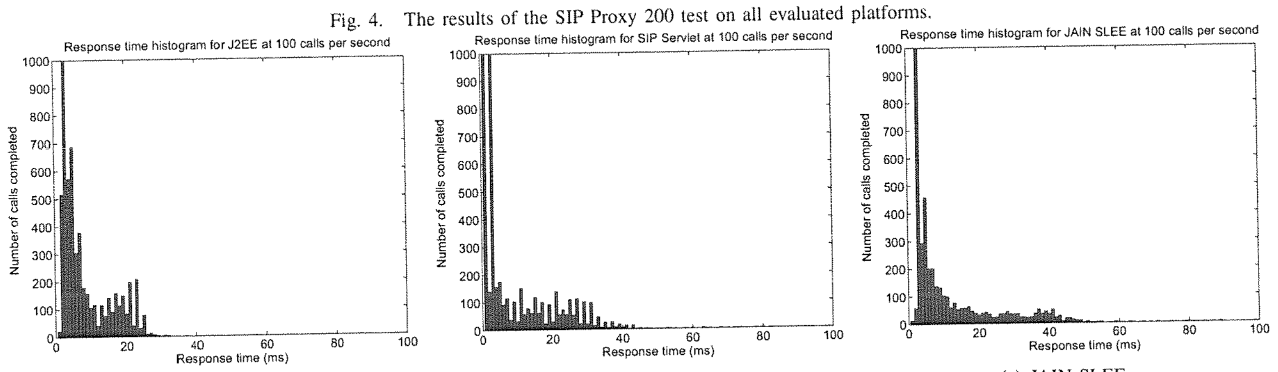
By using this approach the RA implementation is simplified as no manual thread management needs to be performed and the actual core of the RA is restricted to receiving and sending messages over the network. An additional benefit is that it allows the J2EE application server to optimize the scheduling of work for the entire application server whereas local thread management inside the RA is limited to the threads and resources managed by the RA itself.

Depending on the application and the services to be provided it can also be beneficial to embed more (or less) responsibilities inside the RA. The SIP Proxy could for example be embedded in the RA. Or the RA could be used to interact with existing platforms. For example, to develop a billing service it would be enough to be notified when a SIP call starts and when it ends. All intermediate SIP Messages do not necessarily need to be processed inside the J2EE container but could be handled either inside a SIP Proxy RA or in an existing Proxy platform.



(a) J2EE (b) SIP Servlet (c) JAIN SLEE

Fig. 4. The results of the SIP Proxy 200 test on all evaluated platforms.



(a) J2EE (b) SIP Servlet (c) JAIN SLEE

Fig. 5. The response time distribution of the SIP Proxy 200 test on all evaluated platforms for a call rate of 100 calls per second.

VII. Conclusions

In this paper we described the use of J2EE for low latency use cases and compared it to both JAIN SLEE and SIP Servlet. Based on the obtained results we can conclude that Java technologies in general are suitable for low latency, high throughput applications. All tested platforms were able to meet the low latency requirements and could sustain a significant call rate in the proxy 200 test.

The test results also show J2EE (although being limited by the SIP Stack used) being capable of producing satisfying results for this type of applications. As all technologies are capable of achieving low latency performance, increasing the capacity is possible by clustering multiple application servers. Improving the low latency behavior usually is much more complicated.

Depending on the type of application or Service Enabling platform to create, J2EE, JAIN SLEE, SIP Servlet or a combination of the different platforms are all suitable candidates and decisions should be made based on the required base functionality, service complexity and available development time.

Acknowledgment

We would like to thank OpenCloud for providing us with a license for their JAIN SLEE implementation Rhino and BEA for providing us with a license for their Communications platform. The research presented in this paper is partially funded by the IBBT T-Case project. F. De Turck is a postdoctoral Fellow of the Fund for Scientific Research - Flanders (F.W.O.-Vlaanderen).

References

[1] J. de Keijzer, D. Tait, and R. Goedman, "JAIN: A New Approach to Services in Communication Networks," *IEEE Communications Magazine*, vol. 38, no. 1, pp. 94–99, January 2000.
[2] 3GPP, "3GPP A Global Initiative," [online], http://www.3gpp.org/.
[3] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "Session Initiation Protocol," [online], 2002, http://www.ietf.org/rfc/rfc3261.txt?number=3261.
[4] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle, "SIPstone - Benchmarking SIP Server Performance," [online], April 2002, http://www.sipstone.org/.
[5] Sun Microsystems, "J2EE Connector Architecture 1.5," [online], http://www.jcp.org/en/jsr/detail?id=112.
[6] ——, "SIP Servlet API," [online], http://jcp.org/en/jsr/detail?id=116.
[7] ——, "JAIN SLEE API Specification," [online], http://jcp.org/en/jsr/detail?id=22.
[8] M. Ranganathan and P. O'Doherty, "jain-sip: JAVA API for SIP Signaling," [online], https://jain-sip.dev.java.net/.
[9] Hewlett-Packard, "SIPp : SIP benchmarking utility," [online], http://sipp.sourceforge.net/.
[10] B. Van Den Bossche, F. De Turck, B. Dhoedt, and P. Demeester, "Enabling Java-based VoIP backend platforms through JVM performance tuning," 2006, to be published in the proceedings of The 1st IEEE workshop on VoIP Management and Security: VoIP MaSe co-located with IEEE NOMS 2006.
[11] B. Van Den Bossche, F. De Turck, B. Dhoedt, T. Pollet, B. Van Vlerken, J. Moreels, N. Janssens, P. Demeester, and D. Colle, "Evaluation of Current Java Technologies for Telecom Backend Platform Design," in *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 2005, pp. 699–709.

IEEE

GLOBECOM 2006

EXPO

IEEE GLOBECOM 2006

2006 Global Telecommunications Conference

27 November –
1 December 2006
San Francisco
California, USA

IEEE

IEEE
COMMUNICATIONS
SOCIETY

Communications: A Global Bridge

ISBN: 1-4244-0357-X
Copyright© IEEE 2006. All rights reserved.
IEEE Catalog Number: 06CH37800C