Scalability and Performance Evaluation of a JAIN SLEE-Based Platform for VoIP Services

M. Femminella, R. Francescangeli, F. Giacinti, E. Maccherani, A. Parisi, G. Reali
DIEI – University of Perugia
Via G. Duranti 93, 06125 Perugia, Italy, email: *name.surname@diei.unipg.it*

***Abstract*—Advanced Telecom services typically go beyond traditional two parties telephone calls. In particular, SIP-based service creation in the Telephone-IP convergence scenario has become an important research issue. Telecom services are typically asynchronous, require low latency, high throughput and high availability: those peculiarities ask for a specific event-oriented framework. Developers can today benefit from a platform that could grant high-level APIs and standard interfaces, in order to ease the process of service development and deployment, as well as extend the service portability over multiple networks and devices. The JAIN SLEE activity tackles those issues, defining a Java-based technology for the development of telecom services and their deployment into a SLEE server. This paper focuses on some key issues relevant to the Mobicents SLEE platform, an open source implementation of the JAIN SLEE specifications. We present the results of an experimental campaign in which several SIP-based VoIP services were tested, in order to evaluate the performance and scalability of the system. Each of the tested services is characterized by a different complexity, in terms of processing and signaling flow. Moreover, we investigate some of the performance limitations that have been observed experimentally, which may relate to any Java-based telecom service.**

***Index Terms*—Service creation, Internet telephony, JAIN SLEE, Performance, Scalability, Experimental evaluation.**

I. Introduction

Today telecom providers have to deal with a very competitive market in which a continuous renewal of the offer of value-added services is mandatory to protect their revenue. The natural trend is towards the Telephone-IP convergence since it is convenient to develop innovative value-added services in the IP domain, by making use of the SIP protocol. Therefore, SIP service creation has become an important research topic.

Telecom services are intrinsically asynchronous and typically require short latency, high throughput and high availability. Their integration is not simple as they may rely on several network protocols and interoperate with different software and hardware platforms. In that context, developers need a suitable development platform including high-level APIs and standard open interfaces, as well as an event-oriented architecture [1].

The goal of the Java APIs for Integrated Networks (JAIN) activity is to tackle these issues. JAIN has released several development frameworks for service creation, published as open standards under Java Specification Requests (JSR). In particular, the JAIN Service Logic Execution Environment (JSLEE) provides the non functional features for the execution of applications, thus relieving programmers from the burden of dealing with low-level implementation aspects. In the Java terminology, such an environment is called "container".

We have tested several VoIP services within the Mobicents SLEE, which is an open source implementation of the JSLEE specs. We have chosen this platform for the following reasons:

- it has proven to be a viable solution, alternative to high-cost commercial platforms (e.g. Personeta TappS [2]), as we have already implemented and tested a Call Control service for VoIP based on a SIP B2BUA architecture [3];
- being open source, the code is kept updated; at the time of writing, several improvements have been introduced in order to make it compliant with the new JSLEE 1.1 specs released in July 2008 [4];
- the Mobicents Communication Platform is a solid project. It is currently owned by Red Hat [5] and supported by a large community of developers [6].

The scope of this paper is to present the results of an experimental campaign which has investigated some key issues of the Mobicents SLEE. They include:

- the scalability features of the Mobicents JSLEE; due to the complexity of this platform, which is multi-tier and based on multiple levels of abstraction, scalability cannot be given for granted, and need some analysis;
- platform performance limitations observed during experiments, which are related to the complexity of the platform architecture, the Java memory management system, based on the Garbage Collection (GC), the suitable selection of the transport protocol used for the transmission of the SIP signaling messages.

This paper is organized as follows. The next section reports an overview of the issues investigated in this paper. Section III illustrates the JSLEE specs and the Mobicents implementation. Section IV describes some platform limitations and the scalability benchmark methodology. Section V outlines the experimental setup used to evaluate the platform performance, and the relevant results. Section VI provides some final remarks.

II. Related Work

The subject of service creation is tackled in [7], in which several suitable frameworks are described. The Author introduces the concept of SLEE and an implementation case study. Some drawbacks which may result from using high-level development platforms are addressed, including a high

programming skills requirements, and the loss of a very granular control over lower-level objects.

In [8], Authors present a qualitative comparison of different Application Server (AS) implementation technologies for the IP Multimedia Subsystem (IMS): SIP programming techniques (SIP CGI, SIP servlet), SIP programming interfaces (Parlay and Telephony APIs like TAPI, TSAPI, and JTAPI), and JSLEE. The Authors state that a JSLEE implementation can meet the TINA-C requirements [9] very closely and may be used as an IMS AS since it enables rapid service creation and deployment, easy service customization, and the independent evolution of services and network infrastructure. On the other hand, Authors also stress that the JSLEE framework requires a high learning curve.

[10] illustrates the performance achievable by a SIP proxy based on OpenSER [11]. In particular, the impact of the transport protocol on SIP performance is analyzed. The same subject is also considered in [12], in which the Author analyzes the behavior of SIP servers under overload and the congestion control for SIP. Although the choice of a transport protocol may have many implications, it is deduced that TCP provides better performance than UDP under overload for a SIP server. Moreover, as regards the SIP congestion control, this paper explicitly illustrates the difference between SIP over TCP (SIP relies on the TCP congestion control mechanism, which controls a flow of many packets from a single sender to a receiver), and SIP over UDP (overload control is based on application layer mechanisms which, being end-to-end, is focused on the individual SIP requests). Thus, establishing the reference network scenario (i.e. UAs – SIP server, or SIP server – SIP server configurations) is critical for analyzing performance, in order to properly choose the transport protocol and determine its implications on results.

[13] gives a performance comparison of SLEE, Java Enterprise Edition (JEE), and SIP Servlet implementations. The evaluated SLEE is based on the commercial Open Cloud Rhino [14] solution. The paper also deals with the tuning of the Java Virtual Machine (JVM) and GC.

III. A JAIN SLEE Overview

In this section we summarize the JSLEE specifications and the Mobicents architecture, by focusing on some elements that have proven to be critical for the overall performance.

The JSLEE activity consists of a Java-based, event-oriented container for the execution of carrier grade telecom services [4]. A JSLEE employs several JEE technologies, by adapting them to the specific needs of telecom ASs.

The service application logic is implemented into functional modules called Service Building Blocks (SBB), linked together in a parent-child relation.

The state information of an SBB can be stored into Container Managed Persistence (CMP) fields, which guarantee data persistence across various “SBB object” instances of a given “SBB entity”. It should be noted that the JSLEE specification indicates neither how persistence is implemented, nor the level of persistence to be offered.

SBBs operate asynchronously by receiving, processing and triggering events. To accomplish this, they are attached to streams of events called Activity Contexts (AC), which work like a bus. ACs also provide capabilities for sharing common data among SBBs of a given service (e.g. attributes of SIP dialog). The events are delivered to the target SBB by an Event Router. This functional element is critical since it processes all the server events.

Another system performance key aspect is the overall architecture of the deployed service. Several design choices may be critical:

- the number of SBBs implemented and events processed; the way SBBs are instantiated by the SLEE (as “root” SBBs, after an initial event), or by a parent SBB (as “child” SBBs);
- the way SBBs communicate with each other: through events of a given AC, or through synchronous methods of ad-hoc SBB local interfaces [3];
- the selected structures to store and share data: CMP fields or AC attributes.

The access to protocols and network elements is provided by an abstract interface layer made of entities called Resource Adaptors (RA). The RAs are responsible for the conversion of network events, such as reception of SIP messages, into equivalent JSLEE events. They provide a set of interfaces to be used in the SBB coding (e.g. methods to access the SIP header fields of a message).

Mobicents is an open-source communication platform that includes a SLEE, a Media Server, a Presence Server and a SIP Servlet AS [5]. We have used the Mobicents SLEE v.1.2.2 GA, which includes a JSLEE 1.1 compliant SIP RA. Below the Mobicents SLEE will be referred to as MSLEE. It makes use of the JBoss AS v.4 [15] hosting environment. JBoss offers capabilities for service and SLEE management through Java Management Beans (MBean), service deployment, and thread pooling. It represents a container for higher level service containers. MSLEE does not rely on the CMPs offered by the JBoss container. Instead an ad-hoc solution is implemented, which does not include any disk writing.

IV. Platform Limitations and Scalability

The evaluation of the scalability and the limitations of a MSLEE server is not trivial. The JSLEE complex architecture, together with the underlying JBOSS server, the JVM, the transport protocol and external resources, such as databases, considerably increase the number of variables to take into account. Thus, it is necessary to identify the variables that have a major impact on the JSLEE behavior. As already pointed out, the service implemented architecture and the resources used, such as Timers, CMP fields, and external databases, have a major impact over the platform performance. In addition, we have identified as critical elements for the MSLEE operation the GC, used by the JVM, and the transport protocol used by SIP.

A. *Platform Limitations*

1) *Java Memory Management*

The MSLEE is based on the Java technology. One of the

most controversial features of this programming language is the bundled memory management (garbage collection). Having a GC that automatically cleans up and defragments the memory used by an application is useful in most cases. Nevertheless it may represent a potential problem when dealing with low latency constraints. Direct memory management often leads to errors that can be avoided letting Java Runtime do the garbage collection. Typically the Java GC behavior is not under the full control of the developer: one of its side effects is that it can pause the running application for unpredictable times [13]. To avoid this problem, different GCs have been included in Java6 VM [16], thus trying to meet the requirements of different classes of applications. We have evaluated two GCs.

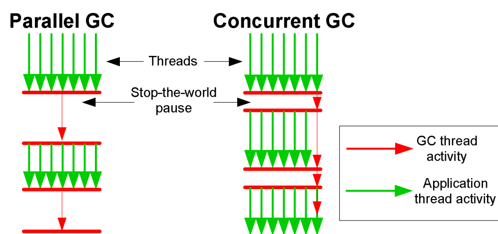

Fig. 1 - Parallel and Concurrent Garbage Collector: Pauses.

Parallel GC is the default configuration and it is best suited for most scenarios, but it introduces unpredictable pauses lasting up to few seconds under heavy load (Fig. 1). In this situation, the MSLEE may enter in a congestion state, since it cannot process messages during this pause. As the GC finishes his collection, the server needs to handle all the unprocessed events accumulated during the pause time. In the worst case, the MSLEE might freeze, thus being unable to manage such an avalanche restart. All that may increase the system latencies, and perhaps neglect stability.

The use of the Concurrent GC seems to be the recommended solution for telecom service scenarios, but it comes with a performance compromise. It tries to minimize the time spent collecting old resources from memory by collecting them while other application threads are running. Only two very small pauses affect the running threads, one at the beginning of a major collection and one at the end of it (Fig. 1). The drawback is that the Concurrent GC trades processor resources (which would otherwise be available to the application) for shorter major collection pause times [16]. Thus a considerable CPU overhead is incurred during the normal JSLEE server operation.

2) Transport Protocol

The MSLEE server version 1.2.2.GA includes both a UDP and a TCP capable SIP RA. In this paper, the usage of these protocols is investigated, by comparing the performance achieved by using them to enable different services. To this aim, we have considered a server to server SIP scenario [12] in which all connections are mono socket.

This architecture is consistent with a service provider reference scenario, made of a trusted IP network integrated with the PSTN domain through one or more gateways, which are network devices in charge of performing media and signaling protocols translation (Fig. 2).

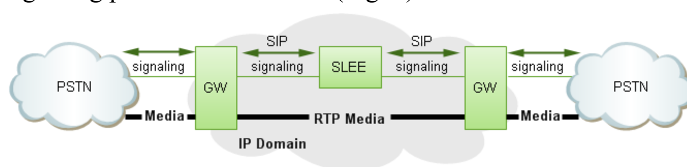

Fig. 2 - Network Reference Scenario.

The UDP does not perform any congestion control, which may be useful during a GC pause. The reliable message delivery service is guaranteed by the SIP transaction sublayer, through retransmission mechanisms.

The use of TCP decreases the SIP retransmission overhead on the SIP RA since only 2xx responses to INVITEs need to be retransmitted [17]. As a matter of fact, when SIP relies on TCP, the SIP transaction sublayer is not fully exploited [1] and many of the SIP retransmission timers are not activated. Thus, a smaller number of retransmissions reduces the number of events internally managed by the server and, consequently, decreases the processing load of the MSLEE Event Router. This should lead to a better throughput performance under heavy load [12]. In particular, complex services may benefit from this lighter SIP operation, whereas simpler ones may suffer from some features of the TCP, such as the higher latency due to the congestion avoidance algorithms in case of packet loss, the greater header size, the higher processing required by the TCP stack (e.g. error checking, retransmissions, and packets ACKing).

B. Services Deployed: a Scalability Benchmark

To test the MSLEE scalability performance and limitations most of our effort have focused on the definition, analysis, and implementation of five different services, characterized by the different internal complexity and the amount of used resources. These services are all SIP-based and include different SIP signalling network elements described in [17]: a User Agent Server (UAS) and Client (UAC), a Stateful Proxy, and a Back-to-Back UA (B2BUA). All the services have been implemented by emulating real communication use cases, such as an auto-responder, a SIP proxy, and a Call Controller for accounting and billing functions. The latter is based on a 3rd Party Call Control (3PCC) scheme [19].

1) Mobicents Load Test

In the first and simplest service implemented, referred to as Mobicents-Load-Test (MLT), the MSLEE acts like a SIP UAS. It is one of the examples included in the MSLEE package, that we have used to compare the results obtained by the Mobicents developers. Its SIP signaling flow is very simple (Fig. 3): for each call, an external SIP UAC sends an INVITE message; once the MSLEE receives it, the service answers with a 200 OK response, waits for the ACK and, as the SIP handshake is done, starts a timer. When the timer expires, after 180 s, a BYE message is sent to the SIP UAC. Finally, a 200 OK response is sent by the UAC to the UAS.

The MLT uses only one timer from the TimerFacility integrated in the JSLEE server. The implementation consists

of a simple root SBB whose instantiation is triggered by an initial INVITE event. Therefore, for each call attempt, a new root SBB entity is created, which last for the whole call. There are no variables stored in CMP fields during all the service operation, thus the service complexity is kept at minimum.

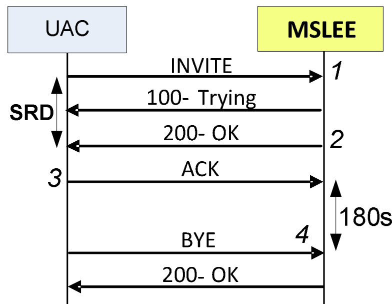

Fig. 3 - Mobicents-Load-Test Service Signaling Flow.

2) *Stateful Proxy*

The second service implemented is a SIP Stateful Proxy (SP). Two external UAs are involved in the communication (Fig. 4): the UAC wishes to establish a session with the UAS and sends an INVITE to the proxy service. Being stateful, the proxy answers with a 100 TRYING message [17] and then forwards the message to the UAS address contained in the SIP URI. The UAS acknowledges the receipt of the message by sending a 180 RINGING response message back, followed by a 200 OK message. Both responses are forwarded upstream by the proxy to the UAC that responds with an ACK message. Once the ACK is correctly forwarded to the UAS, the media session between the two UAs is established and they can communicate directly. After 180 seconds, the UAC is configured to hang up, by generating a BYE request that is routed to the proxy which forwards it to the UAS. A 200 OK response message is then routed back from the UAS to the UAC passing through the Proxy. As regards the processing of Route and Record-Route headers, this scenario implements the Loose Routing technique [17]. The implementation consists of root SBBs whose creation is triggered by new SIP transactions. All events with the same branch of the SIP Via header use the same root SBB. A CMP field that stores a boolean variable is used to understand if the current SIP transaction has ended thus returning the SBB to the MSLEE pool for later re-use without managing the message.

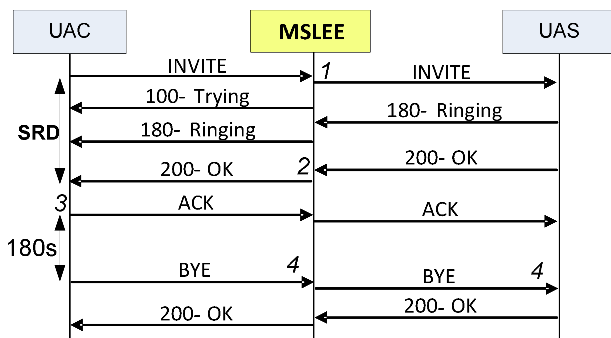

Fig. 4 - SIP Proxy Service Signaling Flow.

3) *Back-to-Back Services*

The other three services implemented are based on the B2BUA architecture. This architecture provides the needed flexibility to manipulate the entire SIP signaling between two end points. A B2BUA is a SIP based logical entity that implements the service logic for 3PCC [19], acting both as:

- a UA server, by processing the incoming request from a calling party, on a first call leg;
- a UA client, by generating outgoing requests on a second call leg, towards the second calling party.

The B2BUA architecture is very useful for service providers who want to exploit the JSLEE server for the deployment of complex services such as call blocking, call forwarding, billing, accounting, conference, and media server.

In order to manage both call legs, the B2BUA should operate in a session stateful way, by maintaining the complete state information of the two SIP dialogs. This is achieved with the use of the Mobicents JAIN SIP Resource Adaptor v.1.2, based on the National Institute of Standards and Technology (NIST) implementation of the JAIN SIP specifications [18]. That RA is compliant with the new JSLEE 1.1 specifications. It simplifies the application development, by providing high-level APIs with specific functions for processing the SIP messages inside their own dialog and transaction contexts, so as to mask the underlying SIP protocol complexity to the service developer.

The B2BUA services must participate in all requests sent on the active dialogs [17]. The SIP signaling flow is similar the one of the SP service, except for the two dialog legs, which are now managed by the MSLEE during the entire call duration (Fig. 5). Moreover, the B2B is in charge of handling the SDP negotiation for the media session setup, which may require SDP manipulation and re-INVITE messages, as described in different scenarios in [19].

The three B2BUA services are implemented in such a way to have an incremental complexity and resources usage.

The first B2BUA service simply forwards the messages received by maintaining the state of the two dialogs instantiated. This service does not make use of timers to control the call duration, so we will refer to is as No Timer (NT). The main difference between SP and NT resides in the internal architecture (Fig. 5). Each time an INVITE request is received a new Selector root SBB is instantiated which, in turn, creates a new child SBB called CallControl SBB that is used to manage the new call. This SBB architecture was chosen to support dynamic service selection by triggering the appropriate child SBB. This feature is turned off in this first simple case, thus the Selector always activate the same SBB child type with no further processing. The SIP call ends when one of the UAs hangs up by sending a BYE request message to the B2BUA that forwards it to the other leg. Two CMP fields are used to store the two dialogs managed by each CallControl SBB and to properly forward every SIP message belonging to the same call flow.

The second B2BUA based service includes a query to a remote MySQL database to obtain the policy to be applied to the call. That query is made by the Selector SBB against the SIP Request URI's User, of the INVITE message. Once the policy information is obtained, the Selector SBB creates the

proper CallControl SBB, and triggers a timer at the beginning of the call. The timer duration is set to 180 s. We will refer to this service as One Timer (1T). When the timer expires, an event is fired to the CallControl SBB which, in turn, sends a BYE request to both legs. At the same time, it performs a final query to the remote database, in order to emulate an offline billing service. The complexity of 1T service is higher than the NT one since the former adds the DB queries and one more CMP field in which to save the TimerID of the started timer. External resources such as the database queries also reduce performance under heavy load, by locking resources and stealing time to event processing. Since the MSLEE is executed inside the JBoss AS which natively supports Java Transaction API (JTA), we have decided to enable this feature to manage secure database accesses. Transactional behavior ensures that after a set of operations has being executed on the remote database the state of the information remains consistent during all the time, as well as after a system failure. Such a guarantee comes with a large performance overhead, in terms of disk writing operations, which are directly related to the number of transactions processed by the service.

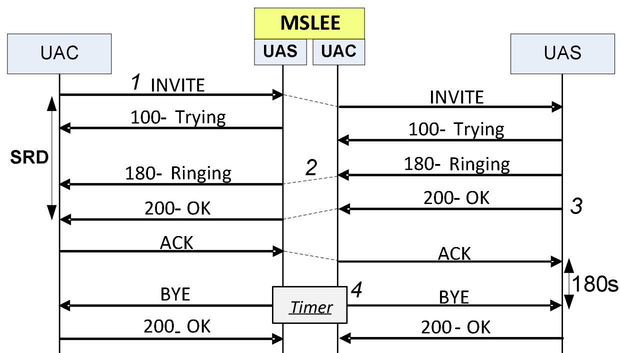

Fig. 5 - B2B Services Signaling Flow.

The fifth and final service is also based on the above 1T. In addition it performs a periodic DB query that is triggered by a periodic timer set with a period of 10 s. We refer to it as Two Timers (2T). This service simulates a dynamic service selection with per-call billing and online charging. The 2T complexity is higher due to the use of one more CMP field, compared to 1T, containing the periodic timer reference, used in each SBB and therefore for each call. Moreover, the periodic queries to database substantially increase the computational overhead and latencies.

To summarize, we have implemented five services: a simple answering UAS (MLT), a SIP proxy (SP), three B2BUA-based services (NT, 1T, 2T). All the services introduced can be dynamically configured through their MBean interfaces from the JBOSS JMX Agent View.

V. Performance Evaluation

A. Testbed Description

Each service has been tested by using the network topology illustrated in (Fig. 6). SIP UAC and UAS have been implemented by two Linux-based PCs with the SIPp traffic generator with customized scenarios, running on top of the Ubuntu Linux distribution v.8.0.4. The MSLEE v.1.2.2 GA, deployed on JBoss v.4.2.3 GA, has been installed on a Fujitsu-Siemens server-class machine PRIMERGY TX300 S4 with dual Intel Xeon E5410 @ 2.33 GHz and 8GB RAM. The server OS is Arch Linux x64 (kernel 2.6.27) and its JVM is version 1.6.0_11 64-bit Server. The subscriber policies have been stored into a MySQL database, deployed in a dedicated remote Linux PC. We have used MySQL server version 5.0.51a, and the MySQL Connector/J version 5.1.6 JDBC driver to access the DB from the MSLEE. Moreover, each DB interrogation relies on an object-relational mapping which is implemented through the Hibernate technology in the JBoss container. All the devices have been connected using a dedicated Gigabit Ethernet LAN switch.

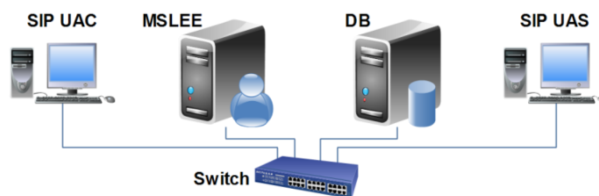

Fig. 6 - Test Bed.

In order to configure the JVM and select the desired GC we have added the command line flags in Table 1 to the configuration script of the JBoss AS. Moreover, before collecting statistics, we have performed a warm up session of 15 minutes to allow the GC to be executed at least once and avoid both JVM and MSLEE transient effects.

Parallel GC	Concurrent GC
-Xms6000m -Xmx6000m -XX:+UseTLAB	-Xms6000m -Xmx6000m -XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode -XX:+UseTLAB

Table 1 - JVM Tuning.

We have deployed the Mobicents JAIN SIP RA v.1.2 in the MSLEE server. We have tuned it by increasing the number of threads, in order to allow a greater number of events to be processed at once. Also we have run the JAIN SLEE 1.0 Technology Compatibility Kit (TCK) on the MSLEE server, which have successfully passed all tests.

All logs have been disabled during the execution of the tests to improve MSLEE and JBOSS AS performance.

The SIPp UAC has been set to generate new SIP calls, by using deterministic arrival process, with a rate equal to λ. Different arrival rates were chosen depending on the type of the service under test. Every test has been done using the same λ value for 60 minutes. The call duration was set to 3 minutes for all tests. Both TCP and UDP have been used for the tests, with the SIPp UAC and UAS set to mono socket mode.

The quantities Successful/Total calls ratio, which is strongly related to throughput, Maximum Throughput, Session Request Delay (SRD), number of SIP messages retransmissions, and system CPU utilization are used as performance metrics. We have specified the traffic load as calls per second (cps). Overload behavior has been also taken into account, being it a key metric to analyze services performance. Overload

condition is reached when the call rate grows beyond Maximum Throughput, which is defined as the served load while maintaining at least a 95% of successful managed calls.

SRD is measured at the UAC side. SRD is defined as the time interval from the initial INVITE to the first non-100 provisional response [20]. SRD values are used to measure the latency experienced by the caller for initiating the call session. Since our SIPp UAS scenario is configured with no pause between the 180 RINGING and the 200 OK messages, we have collected the SRD value at the reception of the 200 OK.

Finally, we have used vmstat, a system monitoring tool, to measure the CPU utilization.

B. Numerical Results

As regards Successful/Total calls ratio achieved versus offered load, for each service four curves are shown, each related to a particular configuration of GC and transport protocol. Fig. 7 depicts the results for the MLT service. Concurrent GC exhibits the best performance with both TCP and UDP transport protocols. In fact, the simplicity of the deployed service does not benefit of the reduced complexity introduced at RA layer by TCP. In addition, Concurrent GC outperforms Parallel one, since it minimizes GC pause times. Fig. 8 shows average CPU load versus offered load for the MLT service. As expected, the Concurrent GC configurations are the most CPU-hungry (see [16] for details). Moreover, the higher the offered load, the higher the performance gap existing between Concurrent GC and Parallel GC. The same service, running with the Parallel GC, exhibits a higher average CPU load when TCP is used, due to the larger kernel processing needed. In the case of Concurrent GC, this difference is hidden by the larger GC CPU usage.

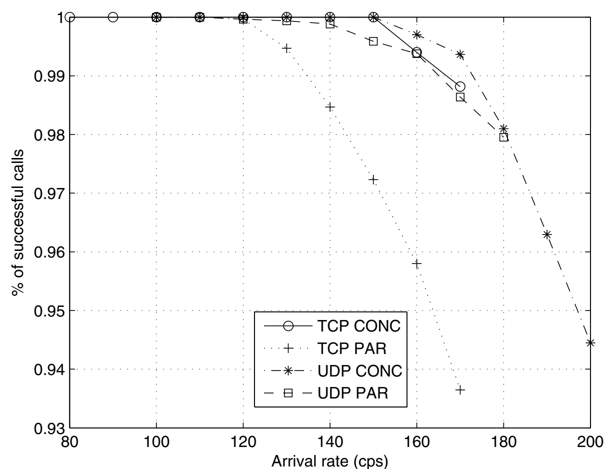

Fig. 7 - Throughput: MLT Service.

Fig. 9 shows throughput (i.e. the Successful/Total calls ratio) versus offered load for the Proxy service. In this case, the need to manage a lot of additional SIP messages with respect to the MLT service makes the TCP as the best choice for the SIP RA. As for GC configuration, this is the only service where the Parallel GC outperforms the Concurrent one. Our explanation is that, even if the service requires much more SBBs, they do not last for the whole call, as in the MLT service, but they are quickly released and returned to the JSLEE SBB pool to be reused. Thus, the GC has globally less aged objects to garbage, and the higher collection efficiency of the Parallel GC allows better performance.

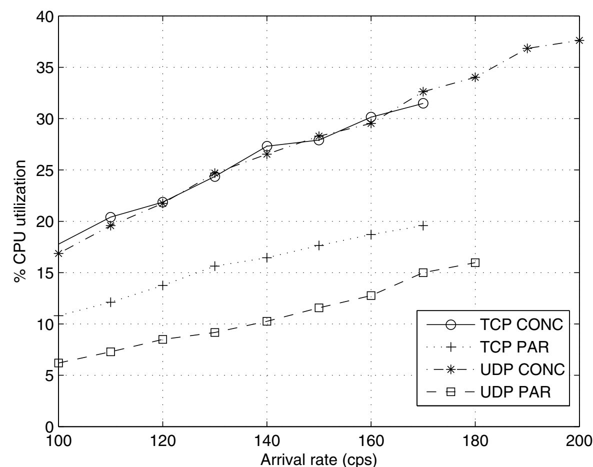

Fig. 8 - Average CPU Load: MLT Service.

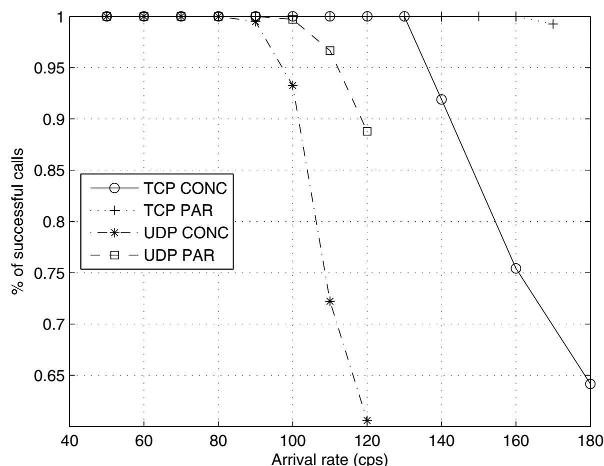

Fig. 9 - Throughput: SIP Proxy Service.

Fig. 10 shows the throughput versus offered load for the B2B NT service. In this case, due to the large number of messages, the SIP RA performs better with TCP, since it manages retransmissions with its own timers, instead of using those of SIP. As for the GC scheme, most objects have a long lifetime equal to the call duration, thus the Concurrent GC, which tries to minimize GC pauses, outperforms the Parallel one. Thus, the best performance is reached with TCP and Concurrent GC scheme. As for set up latencies, Fig. 11 illustrates the SRD values versus offered load for the B2B NT. As expected by analyzing throughput results, Concurrent GC performs better than Parallel GC since it avoids long pauses that causes delays and retransmission, and TCP performs better than UDP. Finally, Fig. 12 illustrates the average number of retransmissions per call versus offered load for the B2B NT service when using UDP. As expected, the number of

messages retransmitted increases together with the offered load, and this effect is more evident for Parallel GC than for Concurrent, due to its longer pause times introduced to perform collections (thus creating an avalanche restart effect).

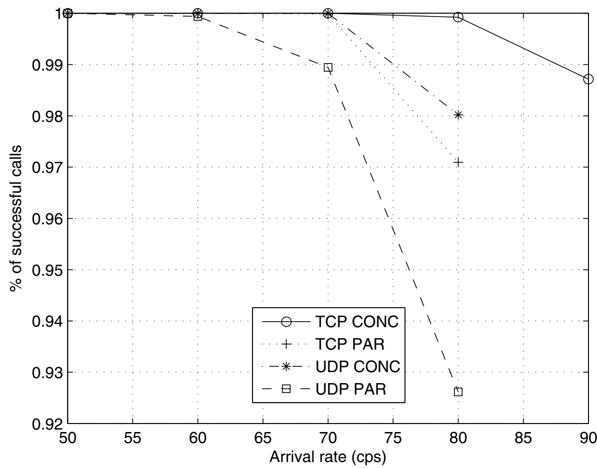

Fig. 10 - Throughput: B2B NT Service.

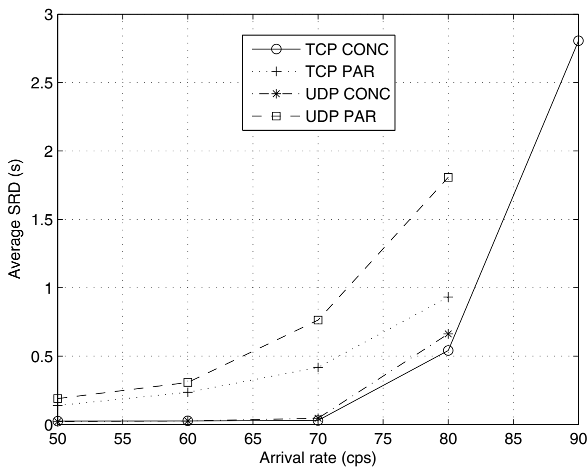

Fig. 11 - Session Request Delay values: B2B NT Service.

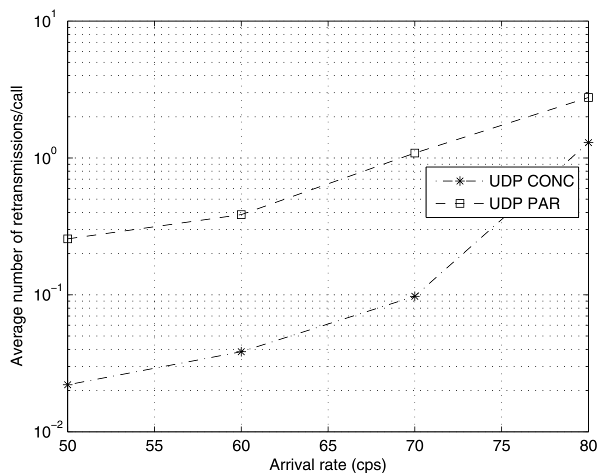

Fig. 12 – Average Number of Retransmissions per Call: B2B NT Service, UDP transport protocol.

The two last services (B2B 1T and 2T) will be commented together, since they exhibit the same behavior. In these services, the system has to cope also with the additional pauses introduced by transactional DB queries (disk writing operations). Fig. 13 presents the throughput versus offered load for the B2B 1T service, and Fig. 14 for the B2B 2T one. The main comment, valid for all configuration, is that the strong throughput reduction observed when compared to the NT service is due to the high number of transactional DB queries, which are the real system bottleneck.

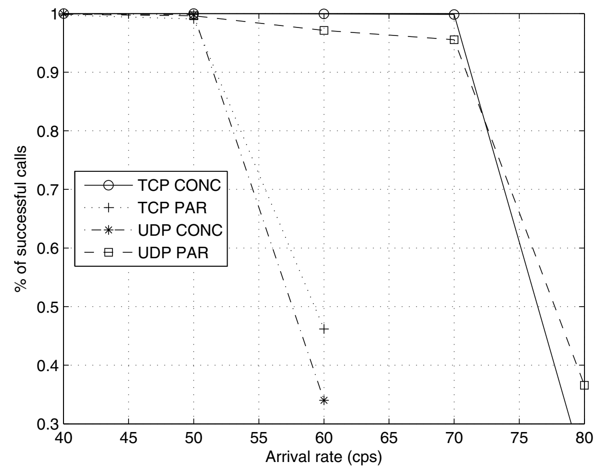

Fig. 13 - Throughput: B2B 1T Service.

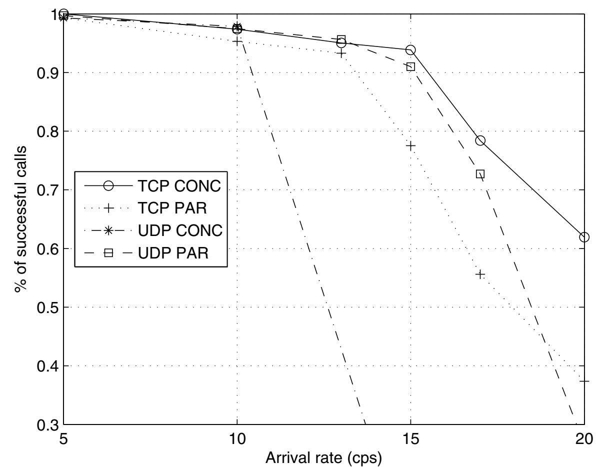

Fig. 14 - Throughput: B2B 2T Service.

The UDP Concurrent configuration has to face not only the short pauses introduced by the GC, but also the pauses due to disk writing transactions operation. This means that the advantages introduced by using the Concurrent GC are nullified, and the avalanche restart is accentuated. Thus, this scheme shows a stable behavior for low loads, but exhibits a sharp throughput decrease when it reaches the overload condition. These problems are a bit mitigated in case of UDP Parallel scheme, in which fewer, even if longer, GC collections allow to deal better with the avalanche restart phenomenon, thanks also to a lower CPU usage.

In the case of TCP Parallel scheme, the long pauses

introduced by the Parallel GC, together with those due to disk writing, may cause many events to suffer from lifetime expiration in the Event Router queues. In this case, the more frequent but shorter pauses of the GC scheme allows the TCP Concurrent configuration to be the best performing one, in terms of throughput and stability in overload.

Finally, Fig. 15 shows the Maximum Throughput values obtained for all services. The figure illustrates the scalability performance of the MSLEE server. As the deployed service complexity increases, the achievable throughput decreases. It is interesting to note that for the most complex services, TCP with Concurrent GC is always the best scheme, moreover, in the last two services, the UDP Parallel configuration provides similar performance. The last two services, characterized by transactional DB queries usage, have in disk writing operation the real bottleneck of the system.

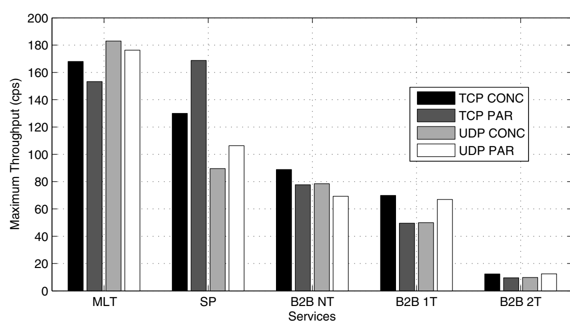

Fig. 15 - Scalability: Maximum Throughput.

C. Discussion

The TCP Concurrent GC configuration seems to be the most effective one for the services introduced and examined in this paper, since it exhibits the lowest average SRD and the highest Maximum Throughput in almost all the services tested. It guarantees stability in overload due to both the congestion control mechanism of the TCP protocol, which reduces the number of retransmissions, and to the Concurrent GC operation, which reduces the collection pauses duration. The drawback of using the Concurrent GC is a greater CPU utilization which, as expected, has been observed during all the tests. However, since the MSLEE seems to not be able to exploit all the CPU capabilities of the server in which it is run (maximum CPU usage never goes over 50%), this drawback does not seem critical at this time.

In [10], the Authors experienced a considerable loss of performance when using TCP, in contrast with our results; this is due to the different architecture of the server (OpenSER is a SIP Proxy developed in C) and the huge difference in the number of handled calls. The higher kernel processing needed by the TCP to handle each call diminishes noticeably the overall SIP server performance. This overhead is not perceived with the load values of our experiments.

VI. CONCLUSION

In this paper we have analyzed the open source Mobicents JSLEE server scalability performance through a set of VoIP services characterized by an increasing internal complexity and resource demand.

The results show that the platform is suitable for production deployment. In fact, even if we consider the most complex B2B 2T service, which may be considered a realistic telecom service, the obtained throughput is comparable with the traffic handled by a large gateway node of an Italian VoIP operator, [21]. The system proves to be scalable with the complexity of the service deployed, although it does not fully utilize the available CPU in all the tests. In addition, the system suffers from unpredictable pauses introduced by the Java GC that can be partially avoided by using the Concurrent GC.

The main conclusion is that Java-based telecom service implementations, and in particular complex, multi-layer systems such as JSLEE, cannot have a single best configuration valid for all kind of services, since it is strongly dependent on how the service is built, how system resources are allocated, number of handled messages, and offered load, etc. However, in the analyzed cases, the best configuration for the most services in terms of performance (throughput, delay) and stability seems the combination of TCP with Concurrent GC. Also the UDP configuration with Parallel GC achieves similar performance for complex services in terms of throughput, but worse performance in terms of latency and stability under overload.

REFERENCES

[1] R. M. Perea, “Internet Multimedia Communications Using SIP,” Morgan Kaufmann, 2008.
[2] Personeta TappS Web Site, available at: http://www.personeta.com/page/programmable_service_platform.aspx
[3] G. Reali et al., “Design, Implementation, and Performance Evaluation of an Advanced SIP-based Call Control for VoIP Services”, IEEE ICC 2009, June 2009, Dresden, Germany.
[4] JSR 240, JAIN SLEE v1.1 Web Site http://jcp.org/en/jsr/detail?id=240.
[5] Mobicents Web Site, available at: http://www.mobicents.org .
[6] Mobicents Google Pages Web Site, available at: http://groups.google.com/group/mobicents-public/web?pli=1.
[7] R. H. Glitho, F. Khendek, A. De Marco, “Creating value added services in Internet Telephony: An overview and a case study on a high-level service creation environment,” IEEE transactions on systems, man and cybernetics. Part C, 2003, vol. 33, no 4, pp. 446-457.
[8] H. Khlifi, J.-C. Gregoire, “IMS Application Servers: Roles, Requirements, and Implementation Technologies”, IEEE Internet Computing, May-June 2008, Volume: 12, Issue: 3, pp. 40-51.
[9] TINA-C Consortium Web Site: http://www.tinac.com.
[10] E. M. Nahum, J. Tracey, C. P. Wright, “Evaluating SIP Proxy Server Performance”, NOSSDAV ’07, Urbana, Illinois, June 2007.
[11] OpenSIPS Project Web Site: http://www.opensips.org.
[12] V. Hilt “Controlling Overload in Networks of SIP Servers”, ICNP 2008, Orlando, Florida, October 2008.
[13] B. Van Den Bossche et al., “J2EE-based Middleware for Low Latency Service Enabling Platforms”, IEEE Globecom 2006, Nov.-Dec. 2006.
[14] Open Cloud Web Site, available at: http://www.opencloud.com.
[15] JBoss Web Site, available at: http://www.jboss.com.
[16] Java SE 6 HotSpot Virtual Machine GC Tuning, Web Site: http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html#available_collectors.
[17] J. Rosenberg et al., “SIP: Session Initiation Protocol,” IETF RFC 3261, June 2002.
[18] NIST JAIN SIP implementation, Web Site, available at: http://snad.ncsl.nist.gov/proj/iptel/jain-sip-1.2/javadoc/.
[19] J. Rosenberg et al.,“Best Current Practices for Third Party Call Control in the Session Initiation Protocol (SIP),” IETF RFC 3725, April 2004.
[20] D. Malas, “SIP End-to-End Performance Metrics”, IETF PMOL WG internet-draft, October 31, 2008.
[21] R. Birke, M. Mellia, M. Petracca, “Understanding VoIP from Backbone Measurements,” IEEE INFOCOM 2007, Anchorage, USA, 2007.