Design, Implementation, and Performance Evaluation of an Advanced SIP-based Call Control for VoIP Services

Mauro Femminella, Roberto Francescangeli, Francesco Giacinti, Emanuele Maccherani, Alessandro Parisi, Gianluca Reali

Abstract—The growing interest in developing value-added services based on Internet Telephony brings into sharp focus several issues related to service creation. It becomes compelling for developers to take advantage of a platform that could offer high-level APIs and standard interfaces, in order to reduce the time-to-market and extend the service portability over multiple networks and devices. The Java community is tackling those issues within the Java APIs for Integrated Networks (JAIN) activity, utilizing Java Enterprise Edition (Java EE) technologies in order to ease the development process of telecommunication services. This paper describes a SIP-based call control service for Internet Telephony, designed for an IP-PSTN converged scenario. The service has a Back-To-Back User Agent (B2BUA) architecture and is deployed into a Service Logic Execution Environment (SLEE), located in the IP domain. Our SLEE is based on Mobicents, an open-source platform for telecom applications, which offers an implementation of the JAIN SLEE specifications. We present the design and a comparative performance evaluation of different architectural implementations of the service, carried out with SIPp, an open-source SIP traffic generation tool.

Index Terms—SIP, JAIN SLEE, Mobicents, Internet telephony.

I. INTRODUCTION

Today, telecom service providers have to face a very competitive market. In order to foster their business, it is necessary to go beyond legacy voice services and exploit the growing potential of end-user devices, in terms of multimedia capabilities and multiple access networks support. The natural trend is to aim at the PSTN-IP convergence, as it is easier to develop innovative value-added services in the IP domain, by using the Session Initiation Protocol (SIP). Therefore, service creation has become an important matter of research.

Telecom services are intrinsically asynchronous and should fulfill short latency, high throughput and high availability requirements. Their integration is not simple, as they may rely on different network protocols and interoperate with different software and hardware platforms. Regarding modularity and reusability, the development process should be improved by using event-oriented design patterns, new levels of abstractions, and well defined logical components. Also, programmers need to benefit from a development platform which grants high-level APIs that could mask the complexity of the underlying levels, so that service creation is simplified and the time-to-market decreases. The introduction of standard open interfaces, rather than proprietary ones, could bring a great advantage by enabling services to support multiple networks and devices, as well as involving a greater number of developers [1].

The JAIN activity aims at tackling those issues, offering middleware frameworks and protocol standard open APIs to allow the development of telecom services independently of the underlying network technology. This effort is pursued through several open standards, published as Java Specification Requests (JSR) under the Java Community Process (JCP).

JAIN SIP v1.2 [17] is a protocol API that provides a low-level interface to the SIP protocol stack and abstracts the network layers implemented beneath. Being at signaling level, JAIN SIP does not enable an easy implementation of the service functional logic, which requires higher level capabilities. This goal is addressed by the more complex JAIN SLEE (JSLEE) v1.1 [16].

The SLEE concept was brought in as a container for communication services, within the Intelligent Network architecture. Container, in Java terminology, is a software environment that provides the non logical features for the execution of an application, thus relieving the programmer from the burden of dealing with low-level implementation aspects. In particular, a SLEE is a framework where telecom services may benefit from common functions such as: Resource Adaptors for interfacing external resources (protocol stacks, devices, databases); an Event Router and Activity Contexts for event handling; Timer Facilities [14]. JSLEE provides a component based and service oriented programming model for the creation of network oriented services. It offers a number of already consolidated technologies imported from Java EE (JEE). It should be noted that JEE is developed for enterprise Application Servers, where enterprise services present different characteristics rather than communication ones, as they are typically synchronous and do not impose any real-time constraint. JSLEE goes beyond the Java EE limitations in developing new telecom services, offering a complementary framework which allows the integration of communication functions into existing enterprise services.

Authors are with the Department of Information and Electronic Engineering, DIEI, University of Perugia, email: [last name]@diei.unipg.it.
Another viable solution for network service creation is using a SIP Servlet API v1.1 implementation [18]. This specification consists of high-level functional interfaces, suited for applications based on SIP or HTTP. The SIP Servlet API is not as flexible as a JSLEE, since it offers fewer capabilities.

Telecom service providers are still relying on capabilities offered exclusively by high-cost commercial platforms in order to create, execute, and manage carrier-grade advanced services (e.g. Personeta TapPS [3]). The goal of this paper is to present a viable open source alternative, which may serve as a first step toward an implementation able to offer the needed carrier-grade capabilities. In more detail, we show how:

- we have designed and implemented from scratch a SIP-based Back-to-Back User Agent (B2BUA) architecture using an open source JSLEE implementation. This signaling architecture offers capabilities needed by enhanced services. We have developed on it a simple call control service, which may be seen as a building block to create more complex services.
- we have integrated the call control with a service selector front-end, which represents an additional functional layer. We have compared the performance of different implementations of the whole service, in terms of throughput and call setup delay.

The paper is organized as follows. The next section reports details about JAIN SLEE and its open source implementation Mobicents. Section III describes the service requirements and the implemented architectures. Section IV presents a performance comparison among these architectures. Finally, section V provides some concluding remarks.

II. SERVICE LOGIC EXECUTION ENVIRONMENT BACKGROUND

A. An Overview of the JAIN SLEE Specification

The JSLEE activity specifies a Java-based, event-oriented container for the execution of carrier grade telecom services.

The proposed component model requires that the service logic should be implemented as a distributed structure made of so-called Service Building Blocks (SBB). A service may be composed by different SBBs linked together in a parent-child relation, so that a service can be represented by a hierarchy tree. Each service has a root SBB, which is located at the top of the tree. In order to instantiate a service the JSLEE activates directly its root SBB, whereas child SBBs are instantiated by their parents.

SBBs operate asynchronously by receiving, processing and firing events they are entitled to handle. To this aim SBBs are attached to streams of events called Activity Contexts (AC) that act like a bus. Through ACs, an SBB can receive and fire events, as well as share common data with other SBBs of the same service (e.g. attributes of a SIP dialog). Events are grouped into event types.

The SBB component model requires that the core logic must reside in the SBB abstract class that implements the javax.slee.Sbb interface. The developer can optionally implements an SBB local interface to specify the methods that may be invoked synchronously by other SBBs of the same service. SBB objects are instances of the JSLEE generated concrete class that implements the developer abstract class. The JSLEE creates a pool of SBB objects and manages them according to their well-defined lifecycle. The SBB abstract class may include methods such as: event handler methods, used to process incoming events; fire event methods; lifecycle callback methods; Container Managed Persistence (CMP) accessor methods. The CMP fields are used to store SBB persistence-related state information that should persist across failures. The developer declares the SBB elements and configuration in a XML document called Deployment Descriptor, where the child relations with other SBBs are specified, the event types fired and handled by the SBB, the SBB "convergence name" for receiving initial events, the set of used CMP fields, and the RAs bindings.

JSLEE does not provide functional APIs that allow access to protocols and network elements. That capability is provided by an abstract interface layer made of Resource Adaptors (RA). Once plugged in, an RA acts as a bridge between the container and external resources by converting network occurrences (e.g. incoming SIP messages) into equivalent Java events ready to be processed by the JSLEE Event Router. This component is in charge of delivering events to all those SBBs that are interested in receiving them. In particular, when the Event Router receives a so-called Initial Event (e.g. the first SIP INVITE message of a call), it generates a new Activity Context and, through a “convergence name” advertisement, chooses a root SBB to handle the event. Vice-versa, when an SBB needs to send a message to the underlying network, it invokes directly the methods of the proper RA interface. The resulting benefit regards modularity as the container is logically decoupled from the underlying network.

The JSLEE framework offers capabilities such as the Timer Facility and the Profile Facility on which the implemented service logic may rely to obtain runtime data such as subscriber information. Also, transaction concepts are applied to the runtime environment in order to grant concurrency control and fault tolerant event processing.

The container environment is managed through Java Management Extension (JMX) entities called Managed Beans (MBean). For example, an MBean can be used on a management web interface to set an SBB attribute value.

Services are packaged into Deployable Unit files which can be dynamically installed into the SLEE container. The service Java code includes SBBs, Profiles and other classes.

Currently, there are only three available implementations of the JSLEE specification v1.0 [15]: Rhino, a commercial JSLEE by Open Cloud [6]; Convergent Service Platform, a commercial JSLEE by jNetX [7]; Mobicents, an open source JSLEE project, owned by Red Hat [12]. We have chosen the Mobicents JSLEE as our framework since it is an open source platform supported by a large community of developers [5].

B. Open Source Solution Adopted: Mobicents

The Mobicents Communication Platform offers capabilities for the creation and the deployment of next generation telecom
services. In addition to an implementation of the JSLEE specification, this platform provides other three core blocks: a SIP Servlet container, a Presence Server, and a Media Server. Also, Mobicents makes a set of RAs available.

The JBoss Application Server [8] is used to hold these four blocks. It is a hosting environment for higher level containers. It offers capabilities such as service and SLEE management, deployment, and thread pooling.

The Mobicents SLEE is still under development. We have used the 1.2.0.BETA3 version. Though the JSLEE v1.1 has been released during the preparation of this paper, our Mobicents SLEE version already implements some of the new features included in the by new specification.

Mobicents JSLEE relies on Java EE technologies, implemented as JBoss components, such as: JMX for the environment management and monitoring through MBeans; Java Naming and Directory Interface (JNDI) which offers lookup functions for service registration; Java Transaction API (JTA) for transaction management.

The Mobicents Management Console is a web interface based on JMX for the management of JSLEE components, by which it is possible, for example, to set-up a MBean attribute, to install a Service Deployable Unit, or to plug in a new RA.

We made use of the following open source Java tools for the development of our service within the Mobicents framework: Eclipse IDE [21], Maven project manager, and Ant for build automation.

III. SERVICE REQUIREMENTS AND IMPLEMENTATION

A. Reference Scenario and Service Requirements

Our work consists in the development of a call control service for Internet Telephony. The reference scenario is a trusted IP network integrated with a PSTN domain through one or more gateways, which are network devices in charge of performing media and signaling protocols conversion.

In the IP domain, the SLEE should implement the control logic of the service by means of handling the entire SIP signaling flow of the incoming calls. The inbound gateway routes the SIP messages to the SLEE statically, so they can be processed and sent to the outbound gateway (Fig. 1). It should be noted that a single gateway can act both as inbound and outbound for a given call. We will not consider the media flow, which concerns exclusively the gateways involved in the call.

Following the requirements, the service should:
- parse the SIP header and identify the callee and the service requested;
- query the provider database and retrieve the subscriber policy information (subscribed services, call duration limit and outbound gateway);
- append a given prefix to the SIP URI, in order to cooperate with the outbound gateway;
- select and activate the appropriate call control service, capable of terminating the active call as a duration timer expires;
- be able to modify the configuration settings at runtime, such as the call duration hard limit or a given gateway prefix.

The requirements ask for Third Party Call Control (3PCC) [11]. This solution consists of a centralized controller capable of manipulate the SIP dialogs between the IP end points, by which a telecom provider can manage the services offered to the subscribers.

B. Service Architecture

Our approach toward 3PCC is based on a Back-to-Back User Agent (B2BUA), which provides the needed flexibility to manipulate the entire SIP signaling between two end points.

As stated in [10], a B2BUA is a SIP based logical entity that implements the service logic for 3PCC. It acts both as:
- an UA server, processing the incoming request from a calling party, on a first call leg;
- an UA client, generating outgoing requests on a second call leg, towards the second calling party.

![Fig. 2 – SIP signaling flow using a B2BUA.](image-url)

As depicted in Fig. 2, the signaling flows of both legs pass through the B2BUA, during the entire call. Once received the SIP INVITE request from the caller gateway (1), the B2BUA sends a provisional 100 back to it (2), queries the provider database to retrieve the subscriber info, and decides what SBB should be activated to perform the desired service. Then, the B2BUA starts the communication setup with the callee gateway, on the second call leg (3). The B2B forwards the SIP messages from one leg to the other transparently, as done to the 180 response (4). The B2B is also 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 [11]. Finally, if the call reaches its duration limit, a timer expires and the B2BUA
sends a BYE request in each leg, terminating the end-to-end call (5).

As depicted in Fig. 3, the call processing flow of our service involves sequentially the following actions, starting with an initial INVITE message.

- The implemented SIP RA translates the SIP message received from the operating system protocol stack in a proper event (1,2).
- The Event Router sends the event through an Activity Context to the interested SLEE components, and instantiates the Selector as the service root SBB (3).
- The Selector SBB queries the database (4) and activates the proper child SBB (5). In order to test the Selector capabilities, we have implemented two SBBs, named CallControl_1 and CallControl_2, which have the same logic and differ only on the timer hard limit.
- The chosen call control SBB sets up the second call leg.
- The gateways are directly connected in the media plane.
- If necessary, the call control SBB manages the call termination (7,8) using the Timer Facility provided by the SLEE (6).

![Diagram of functional entities and message flow](image)

Fig. 3 – Functional entities and message flow in the testbed.

Implementing a B2B behavior is not trivial. In order to manage both call legs as previously described above, the B2BUAs should operate in a session stateful way, that is, by maintaining the 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 [19]. This RA 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. In our service, we use SIP Transaction as well as SIP Dialogs activities.

We have implemented and evaluated three different solutions regarding the Selector SBB instantiation and its relation with the chosen call control SBB (see Fig. 4). Below, we describe them giving them short names for later reference.

1. “SINGLE”: a single root Selector is instantiated for the processing of all the incoming calls. Its convergence name, used by the SLEE container to reference purposes, is obtained by specifying the following entity selector field in the XML SBB Deployment Descriptor file: <initial-event-select variable="EventType"/>.

2. “MULTI”: every call is managed by a different Selector root instance. The entity selector field changes into <initial-event-event-select variable="Event"/>. The chosen CallControl is activated as a child SBB.

3. “EVENT”: the Selector instantiation is the same as MULTI, whereas both CallControl SBBs are set with an ad-hoc initial event called Activate. The Selector chooses the proper SBB, inserts its name into a string inside the event object, and fires that event. Only the SBB the name of which matches the string is going to be activated. In this solution, the chosen call control SBB is activated as a root entity, so the SLEE operates with two distinct services: one based on the Selector, the other having the call control SBB.

4. In the following, we describe the implemented SBBs in more detail.

The Selector receives the initial INVITE request and identifies the subscriber, parsing the SIP Request URI. More precisely, it looks for a string prefix appended by the gateway. Then, it sends the 100 provisional response to the caller gateway, in order to stop any INVITE retransmission, and queries the database to gather information about the subscriber policy name, the call duration and the IP address of the outbound gateway. The suitable call control SBB is chosen in accordance with the subscriber policy. The Selector activates that SBB and attaches it to the server transaction activity context used for the initial INVITE, in order to grant the access to the needed state information to manage the first leg dialog. Finally, the Selector detaches itself from that activity context.

The provider database is deployed on a remote server. It is based on MySQL. The SLEE interrogates it by means of Enterprise Java Beans 3.0 (EJB3.0) objects. In the JBoss container implementation, this technology offers integrated MySQL APIs as well as Hibernate capabilities which enable persistence management and object/relational mapping, thus database tables can be translated into ad-hoc Java classes.

In our service, those persistent objects are handled by a pool manager, which is configured by a singleton class at the server startup.

The chosen call control SBB takes control of the first leg and establishes the second leg towards the callee, for which it appends a prefix to the SIP URI that indicates the outbound gateway how to manage signaling. During the SIP three way handshake, it also manages the end points SDP negotiation, by handling the request/offer mechanism between the two legs. After receiving the 200 OK from the callee, it fires the call duration timer. CMP fields are used to store call state parameters, whereas MBeans are used to setup configuration parameters through the Mobicents Management Console.

IV. PERFORMANCE EVALUATION

A. Testbed Setup

In order to test our system we have used the following tools:

- SIPp, a SIP traffic generator and analyzer [20];
- Wireshark, an Internet traffic statistic collector with relevant VoIP and SIP analysis tools;
- `vmstat`, a Linux OS monitoring tool for CPU utilization;
- `Tagtraum GCViewer`, a Java Garbage Collector monitoring tool.

The tests have been done by replacing the caller and callee gateways presented in the reference scenario (Fig. 1) with two commercial PCs (see Fig. 3). In the caller and callee PCs we have installed a SIPp client UA (UAC) and a SIPp server UA (UAS), respectively, using customized scenarios. Before this testing campaign, we have successfully tested our SLEE with a Cisco PGW 2200 Softswitch [4] operating as both inbound and outbound gateway.

The Mobicents JSLEE has been deployed on a PC with CPU Intel Core 2 Duo E6700 @ 2.66GHz, 3 GB RAM; OS Linux Zenwalk kernel 2.6.25.4 with symmetric multiprocessing and multi-core capabilities; Java Virtual Machine (JVM) jre1.6.0, with 2 GB allocated for JVM memory heap. A MySQL database has been installed on a separate Linux box. All the devices have been connected by a Fast Ethernet LAN switch. We have recompiled the SLEE OS kernel, by modifying it in order to enable an higher throughput on the I/O bus and a more efficient heap management. Also, we have disabled default JBoss and Mobicents logs to improve performance.

The SIPp UAC has been set to generate SIP traffic with Poisson arrival process with an average rate equal to \( \lambda \) ranging from 5 to 60 calls per second (cps) in steps of 5 cps, and exponentially distributed call duration with an average equal to 3.5 minutes. The call control service has been set to tear down each call with duration larger than 3.5 minutes. Thus the SLEE has torn down approximately 37% of the calls, whereas the remaining ones were closed by the UAC. We have configured the SIPp UAC to transfer SIP messages using UDP encapsulation.

Each test has been done with a constant \( \lambda \) value for 30 minutes. Before starting to collect statistics, we have performed a warm-up session of 15 minutes to allow the Garbage Collector to be executed at least once and avoid JVM transient effects.

**B. Performance Metrics and Numerical Results**

Throughput, obtained as the percentage of successful calls, and Session Request Delay (SRD) (Fig. 2) are used as performance metrics. SRD [9] is measured in the caller side and it is defined as the time interval from the initial INVITE to the first non-100 provisional response. It is used to measure the latency experienced by the caller for initiating the call session. Since our SIPp UAS is configured with no pause between the 180 RINGING and the 200 OK messages, we triggered the measuring timer at the reception of the 200 OK [13]. We have collected data for both metrics from SIPp statistics output files. Also, we have used `vmstat` for CPU utilization, configured to log values every 30 s.

Fig. 5 shows the throughput versus offered load for the three architectures. MULTI exhibits the best performance, since it achieves the highest throughput and grants stability also under heavy load. EVENT offers similar throughput for values of \( \lambda \) up to 40 cps. After that, the throughput decreases due to the implemented event-based mechanism for the activation of the SBBs. It introduces more overhead in the SLEE Event Router than the parent-child method adopted in MULTI. Also, EVENT drops beyond 50 cps. SINGLE exhibits the worst performance and shows instability even at low cps. Thus, handling all incoming calls with a unique root SBB mechanism is not a suitable service implementation.

In Fig. 6, we present the SRD average values versus offered load for the three architectures, with the relevant 99% confidence intervals. MULTI and EVENT offer similar SRD values, showing delays below 1s for \( \lambda \) values up to 50 cps. SINGLE shows worse performance, with significant delays even at low rates.

Finally, Fig. 7 depicts the CPU utilization percentage versus offered load for the three architectures, with the relevant 99% confidence intervals. This figure confirms our comments about the previous two figures. As a matter of fact, MULTI and EVENT require almost the same CPU for values of \( \lambda \) up to 50 cps.

![Fig. 4 – Service architecture alternatives.](image_url)

![Fig. 5 – Throughput vs. calls arrival rate.](image_url)

![Fig. 6 – SRD vs. offered load.](image_url)

![Fig. 7 – CPU utilization percentage.](image_url)
We have presented a performance evaluation of three different service implementations, which may represent a first step toward a fully fledged Mobicents JSLEE benchmark. Our tests have brought out the architecture with one root SBB per call as the best one (MULTI), which can process up to about 37 cps with a percentage of unsuccessfully handled calls below 1%. Our SLEE is suitable for production deployment, as the offered throughput performance value, obtained with a commercial PC, is greater than the traffic handled by a large gateway node of an Italian VoIP operator, located in Milan, as stated by measurement shown in [2].

We have shown that the open source Mobicents implementation of JSLEE may give a viable solution for addressing the topics of service creation, convergence of different telecom technologies, and telecom APIs standardization.

REFERENCES


Fig. 6 – Average setup delay vs. calls arrival rate.

For EVENT, the CPU utilization starts increasing exponentially at 55 cps and leads the SLEE to instability. This is due to the higher number of events that need to be handled by the Event Router which implies more CPU processing. In addition, we can exclude that the memory utilization is the reason of the unstable behavior of EVENT since we have verified with the GCViewer tool the proper operation of the JVM Garbage Collector. As for the SINGLE behavior, the CPU load is stably higher than MULTI and EVENT, and the system become highly unstable for values above 25 cps.

In conclusion, MULTI is the best service architecture, since it offers the best performance in terms of throughput and SRD delay together with a stable behavior in overload conditions.

V. CONCLUSION

In this paper we have presented a SIP-based call control service based on the open source Mobicents JSLEE container, featuring a B2BUA implementation. This signaling architecture was implemented from scratch since it was not part of the Mobicents platform. The B2B signalling enables the development of complex carrier grade services. Our application also relies on a service selector front-end that allows the SLEE integration with carrier grade devices in a production environment. To this aim, we have also successfully tested our SLEE with a Cisco PGW 2200 Softswitch.

We have shown that the open source Mobicents implementation of JSLEE may give a viable solution for addressing the topics of service creation, convergence of different telecom technologies, and telecom APIs standardization.

Fig. 7 – Percentage CPU usage vs. calls arrival rate.