An extended Java Call Control for Session Initiation Protocol

Mauro Femminella, Francesco Giacinti, Gianluca Reali

Department of Information and Electronic Engineering - DIEI

University of Perugia

Via G. Duranti, 93, Perugia, Italy

{mauro.femminella,francesco.giacinti,gianluca.reali}@diei.unipg.it

Abstract

In this paper we propose a functional mapping between Java Call Control (JCC) and Session Initiation Protocol (SIP). We show its effectiveness in enabling easy service design and implementation through experimental results. For this purpose, we have implemented a JCC-SIP Resource Adaptor for a Jain Service Logic Execution Environment (JSLEE). In particular, we have used the Mobicents JSLEE, which is the only existing open source JSLEE implementation. Results, obtained by implementing a typical VoIP service, show both feasibility and good performance of our proposal.

Keywords: SIP, JCC, JSLEE, Mobicents

1. Introduction

Improving the design and implementation processes of multimedia services is essential for telcos aiming to rapidly innovate their services by resorting to the use of third-party applications and libraries.

A growing number of open source technologies are available to fulfill the requirements of open service architectures. Due to its platform independence, Java is a natural candidate for implementing advanced services. In particular, the Java APIs for Integrated Networks (JAIN) Server Logic Execution Environment (SLEE) and Session Initiation Protocol (SIP) Servlet, have been created to explicitly aid developers in creating, deploying, and managing advanced telecom services. The JSLEE specifications explain how to realize communication platforms able to fulfill the constraints of multimedia services, such as low latency, high throughput, and high availability. They also provide a point of integration of multiple network resources and protocols [15].

A further simplification of service creation is achievable by the use of the JCC APIs [8][16][17]. They have been created with the specific objective to free service developer from the burden of handling the underlying set of network protocols (e.g. Signaling System No. 7, H.323, SIP).

Currently, the JAIN SLEE package does not include any JCC support for SIP, although SIP is currently the *de-facto* standard protocol for session management in VoIP (*Voice over IP*) networks.

So far, few papers show proposals for mapping JCC onto SIP, and unfortunately none of them include very detailed implementation information. The JAIN community has produced a document that illustrates a possible mapping between JCC and SIP [9]. However, this mapping is only descriptive and provides only general implementation details about JCC mapping onto underlying network. Each JCC API method has been mapped onto a SIP message but, in various cases, correspondence between methods and messages has not been provided.

A more detailed approach can be found in [1], where the authors exploit SIP and JCC complementarities. Nevertheless, the paper refers to the initial JCC specifications and does not provide exhaustive information on *business* (i.e. server side) interface design and implementation.

Currently, implementations of JAIN SLEE JCC adaptors exist for intelligent network (IN) protocols (e.g. CAMEL, CAP, INAP). Even the main JSLEE implementers have not provided a JCC support for SIP yet, probably since SIP is rather different from other IN protocols, and the JCC-SIP mapping is not straightforward. For this reason, in the JCC-SIP mapping of our proposal, the semantic of some JCC methods has been slightly modified. All changes and extensions are thoroughly explained in what follows, along with their implementation within the Mobicents Communication Platform [6], which is the only available open-source JAIN SLEE.

The paper is organized as follows. Section II illustrates background concepts about JSLEE, JCC, and SIP. Section III describes the JCC-SIP operation. Section IV reports some implementation details and presents experimental results. Section V reports some concluding remarks.

2. Background

2.1 Jain SLEE overview

The JSLEE is a Java standard specification for creating and hosting telecom services. It is event-driven and designed for hosting high performance, asynchronous, and fault tolerant application servers (AS) [15]. A JSLEE AS is a software container suited for hosting communication services. It also provides the non logical features for executing applications, which relieves programmers from the burden of dealing with low-level implementation aspects. A JSLEE includes instruments such as: (i) Resource Adaptors (RAs), used for accessing external resources, such as protocols, devices, and databases; (ii) an Event Router, which delivers each event to the appropriate SBB; (iii) Timer Facilities, useful to implement a service logic including scheduled actions.

A service logic is organized in components called Service Building Blocks (SBB) which operate asynchronously by receiving, processing, and firing events. SBBs may be arranged hierarchically, according to parent-child relations, thus increasing service modularity and flexibility.

Events from external, such as reception of a SIP message, are translated into internal Java events by RAs, and dispatched to the correct SBB by the Event Router.

Currently three main implementations of the JSLEE specification exist: Rhino, a commercial JSLEE owned by Open Cloud [7]; jNetX Convergent Service Platform, a commercial JSLEE owned by Amdocs [13]; Mobicents JAIN SLEE (MSLEE), owned by Red Hat [6], that we have selected since it is open source and supported by a large community of developers. It includes a JSLEE container, a Media Server, a Presence Server, and a SIP Servlet container. It is hosted within the JBoss container, which offers capabilities for service and SLEE management through Java Management Beans (MBean), service deployment, and thread pooling. The MSLEE complements J2EE to enable convergence of voice, video, and data for implementing advanced applications.

2.2 Java Call Control API Overview

The JCC API can handle communication sessions through a variety of heterogeneous networks [1][8][9][16]. Its goal is to hide most of complexity of underlying network, which can be a combination of circuit and packet-switched components, wireless or wired. It includes interfaces, classes, operations, events, capabilities, and exceptions. Four key objects are specified:

- *JCC Provider*: it is the interface through which an application can access the implemented JCC functions; it includes specific methods to add and remove event listeners, to get the name and state of the Provider, to shut it down, to create a JCC Call object and to return the object given a JCC Address string.

- *JCC Call*: it represents a communication between two or more parties through a dynamic collection of physical and logical entities involved in the communication;
- *JCC Connection*: it is a dynamic relationship between a *Call* and an *Address*;
- *JCC Address*: it is a logical endpoint, such as a directory number or an IP address.

Figure 1 shows the hierarchical relationship of these objects, their relationship type, and their Finite State Machine (FSM).

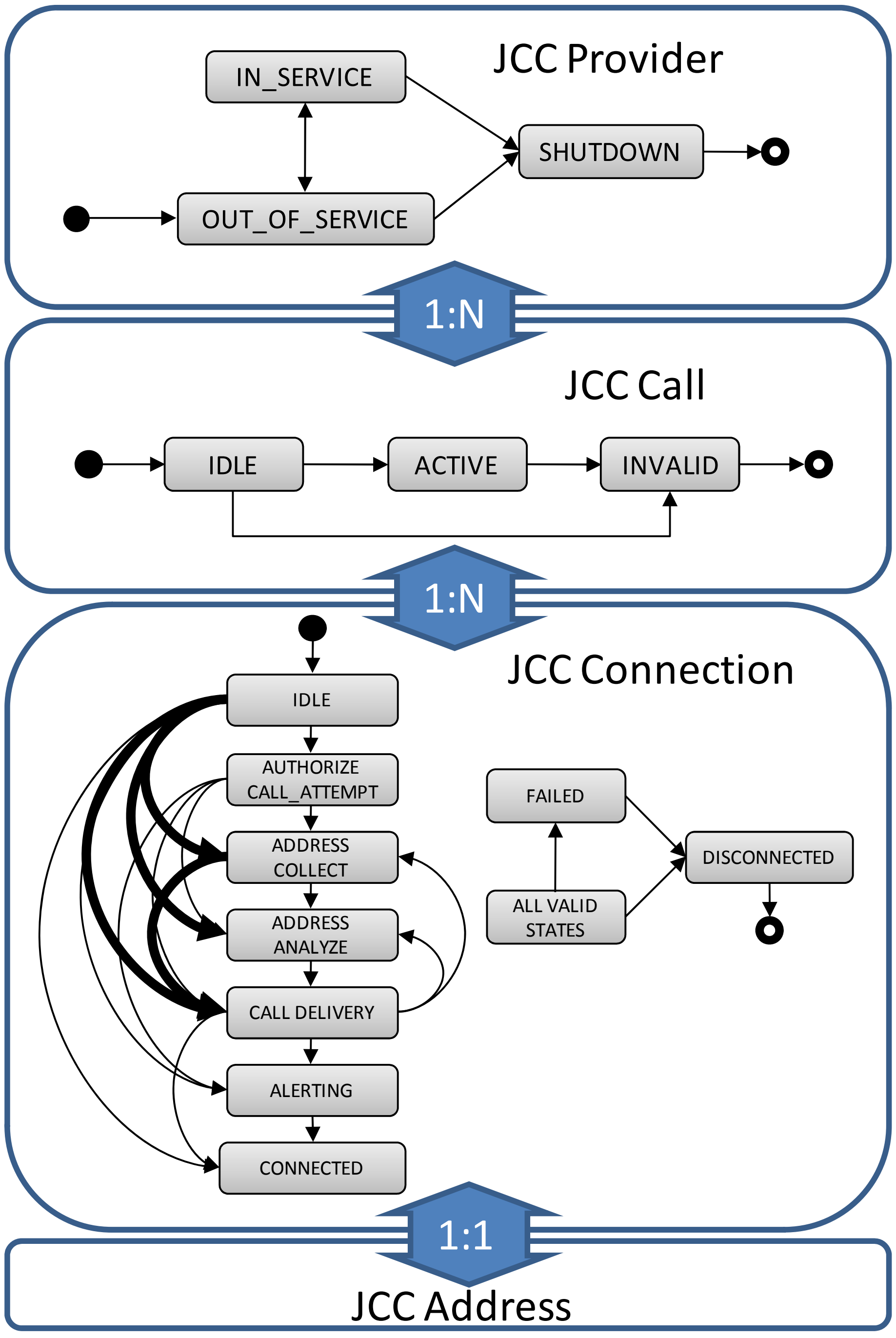

Figure 1:Relationship and FSM of the JCC API key objects. Thick arrows indicate modified transitions.

The *IN_SERVICE* state of the JCC Provider indicates that it is alive and available for use; the *OUT_OF_SERVICE* state indicates that it is temporarily not available; the *SHUTDOWN* state indicates that it is permanently no longer available.

The JCC Call object can be created either by invoking the *createCall()* method on the JCC Provider for an outgoing call or in response to a received network notification of an incoming call. A call may be controlled by any of the involved parties. In the *IDLE* state, a JCC Call is not associated with any JCC Connection; in the *ACTIVE* state, a JCC call has some ongoing activity and must have at least one associated JCC Connection; the *INVALID* state indicates that a JCC Call has lost all its JCC Connection objects.

A JCC Connection object consists of the relationship between a JCC Call object and a JCC Address object. Figure 1 shows the FSM associated with a JCC Connection, along with our modifications shown by thick arrows, proposed to deal with the SIP protocol. The *IDLE* state is the initial state. The *AUTHORIZE_CALL_ATTEMPT* is associated with authorizing the originating and terminating terminals for the Call. In the *ADDRESS_COLLECT* state, the initial address information package is collected and processed according to a “dialing plan” to determine the final address; the *ADDRESS_ANALYZE* state is entered when the complete initial information package from the originating party is available. The information collected is processed according to a dialing plan in order to resolve routing address and call type; the *CALL_DELIVERY* state of the originating party implies the selection of the route and forwarding a message to notify the called party of the desire to start a call. On the terminating side this state implies checking the busy/idle status of the terminating access so as to notify the terminating side of an incoming call; the *ALERTING* state means that the Address is being notified of an incoming call; the *CONNECTED* state implies that a JCC Connection and its Address are part of a telephone call. Thus, two communicating parties are represented by two JCC Connections of the same JCC Call staying in the *CONNECTED* state. In the *DISCONNECTED* state, a Connection is no longer part of the telephone call, even if the Call and Address references are still valid; the *FAILED* state indicates that a Connection has failed.

Each state transition of the JCC Connection FSM generates an event. In order to deliver events to the appropriate listeners, the JCC specifications use *EventFilters*. Events are discarded if no listeners are identified.

A JCC Address object represents an endpoint. It has a string representation, such as a telephone number. During a communication between parties, an Address object is related to a Call object via a Connection object.

2.3 SIP Overview

The SIP protocol is a application-layer peer-to-peer signaling protocol [3] used for creating and managing application sessions over IP networks. It is a text-based request-response protocol which allows supporting innovative services, such as voice-enriched e-commerce, web page click-to-dial, advanced Instant Messaging, and IP Centrex services.

SIP Sessions are established through a three-way signaling exchange between a so-called User Agent Client (UAC) and a User Agent Server (UAS). An exchange starts with an INVITE message, sent by the UAC to the UAS, one or more provisional responses (i.e. 100 Trying, 180 Ringing) replied back by the UAS to the UAC, a final response sent by the UAS to the UAC (200 OK), and a final acknowledgment (ACK) sent by the UAC to the UAS. The set of messages including a request and the relevant responses consists of a SIP transaction.

The session setup messages include also Session Description Protocol (SDP) offers [4], used for describing media and specifying their parameters, such as ports and encoders.

Typically, signaling is exchanged through SIP proxies, providing user registration, location, and call routing functions. A sample service showing the use of proxies is shown in Figure 2. Its implementation will be shown as a proof of concept of our proposal. It consists of a third party call control based on a central *Back to Back User Agent* (B2BUA), implemented by an MSLEE, which acts as both SIP UAC and SIP UAS. It splits each call (represented by a *JccCall* instance) in two SIP dialogs over two distinct call-legs (*JccConnection A* and *JccConnection B*). Five main signaling entities are involved: SIP UAC, SIP UAS, SIP Proxy, MSLEE implementing the B2BUA, and Database (DB), used to store user profiles and service related parameters.

A signaling flow is started by the SIP UAC, which sends a SIP INVITE message to the SIP proxy, which in turn routes the message towards the MSLEE. When this INVITE is received, the MSLEE event routing subsystem is invoked and a *Selector* root SBB is created. It immediately queries the DB in order to retrieve the subscriber profile. Now the *Selector* SBB has all the information needed and can both activate a child SBB, called *CallControl*, and leave the signaling flow control to it. At this point, the child SBB creates the second call leg towards the UAS and establishes the media session between endpoints (represented by the *JccConnection* instances). The subsequent DB queries are used to update service related information, such us the residual credit for prepaid services [10]. A final DB query, upon call termination, is used to update service-related information in the database.

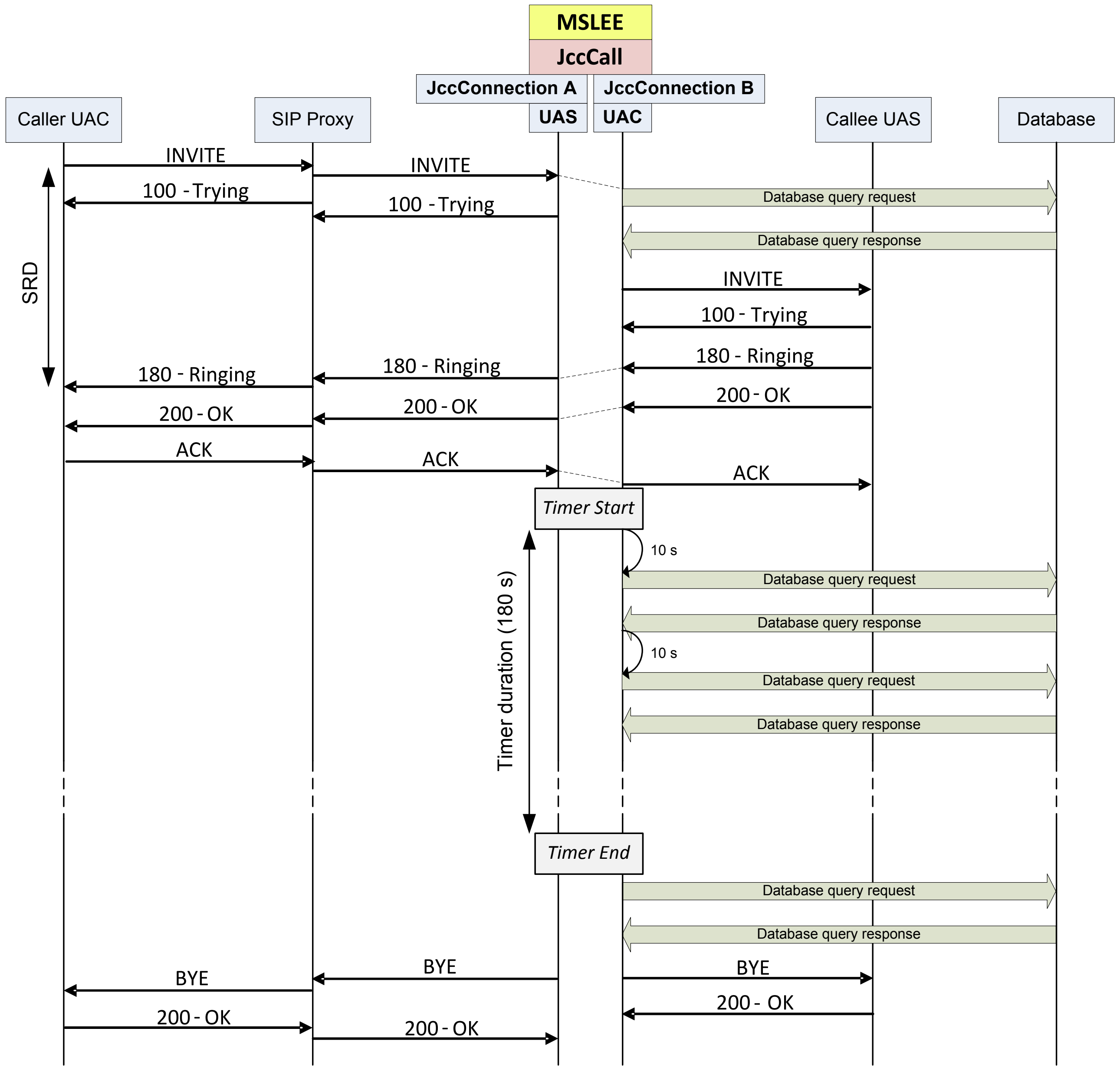

Figure 2 –Signaling messages for session setup and tear-down.

For implementing SIP-based applications, programmers may use specific APIs. JAIN SIP is a Java implementation of a low-level SIP interface. It allows manipulating messages, headers, parameters, ports, and IP addresses. It can be used to build SIP entities such as UAs, proxies, and

B2BUA. Anyway, programmer is left with the burden implementing the SIP logic first, and then an application on top of it, which can be time-consuming and error prone.

3. JCC-SIP operation

Our JCC-SIP combination includes significant novelties compared to previous proposals. In [1], which is the most complete JCC-SIP mapping description found in the literature, the authors' aim was to demonstrate that SIP and JCC are complementary. In this paper there are few and/or not exhaustive details on business level interface design and implementation relevant to the mapping between SIP message exchange and state transitions of the JCC Connection objects. In our paper, we have realized a detailed mapping, at business level, between JCC and SIP, relevant to the creation of third party call control complex applications. In addition, the mapping between the SIP third party call control mechanism, illustrated in IETF RFC 3725 (Flow III) [2], and JCC, that we have designed, is an original contribution of this paper. Finally, the paper [1] refers to the obsolete JCC specifications v.1.0, thus to the JCC 1.0 Connection FSM, whereas our mapping is based on the JCC specification v.1.1, also known as v.1.0b [8]. In the JCC specifications v.1.1, the JCC Connection FSM has been modified, thus some adaptations are necessary. Our proposed modifications, that also simplifies and allows complex call setup, are highlighted by the thick arrows in Figure 1. The operation of the modified JCC Connection FSM is illustrated by the signaling exchange shown in Figure 3 and in Figure 4.

The JCC responses to SIP requests have been implemented through six SIP Request handlers, according to the RFC 3261. Five of them are used to manage a specific SIP request, and one more to handle possible unsupported ones. A further Response handler, able to handle all SIP responses, has also been implemented.

- *InviteRequestHandler*:
  when a SIP INVITE is received, this handler checks if it refers to an existing call. In case of a new call, the handler creates new JCC Call and Connection objects that are associated with the UA which has sent the SIP INVITE message. In case of an existing call, the handler retrieves the JCC Call and Connection objects relevant to the sending UA. This handler makes use of the JCC Call object, relevant to the existing call, to infer whether this call is a B2BUA call or a single leg one. To this aim, it retrieves the mapping, stored in the JCC Call object, between the JCC Connection objects (i.e. between the SIP legs). If the JCC

Connection object, relevant to the UA that has sent the SIP INVITE, is associated with another JCC Connection object, then the call is a B2BUA one, otherwise it is a single leg call. If the call is established by a B2BUA, the INVITE message is forwarded towards the second leg. If it is single leg, the handler replies with a SIP 200 OK response.

- *AckRequestHandler/CancelRequestHandler/ByeRequestHandler*
  when a SIP ACK/CANCEL/BYE request is received, this handler retrieves the JCC Connection object relevant to the UA that has sent the ACK/CANCEL/BYE and delivers it to this JCC Connection object for subsequent processing.
- *OptionsRequestHandler*:
  when a SIP OPTIONS request is received, this handler replies with a SIP 200 OK response to the UA that has sent the OPTIONS message. All information about the server capabilities are included in the response message.
- *UnsupportedRequestHandler*:
  when an unsupported SIP request is received, this handler replies with a SIP 501 NOT IMPLEMENTED response. All information about the server capabilities and the supported SIP requests are inserted in the response message.
- *ResponseHandler*:
  when a SIP response is received, this handler retrieves the relevant JCC Connection object relevant to the UA that has sent the SIP response and delivers it to this JCC Connection object for subsequent processing.

In what follows we illustrate the basic operation of our JCC-SIP implementation tool by showing the signaling exchange of two services, by using the same graphical style of [1]. Although some implemented capabilities fall out of the scope of these services, such as the SIP *Redirection* [3], they are sufficient to depict the potential of our proposal.

The involved entities are: JCC Application, JCC Calls, JCC Connections, JCC Provider, stateful SIP server, and SIP end Parties (UAC and UAS).

The first service consists of a JCC-SIP point-to-point telephone call, established through a B2BUA that manages each call leg independently of each other. The call setup signaling is depicted in Figure 3. It is assumed that authentication, authorization, and accounting (AAA) operations for each user have already been executed. In order to preserve readability and neatness of the figure, arguments of JCC methods are not shown. The current state of the JCC Call and Connection objects is shown in dashed boxes. All messages exchanged by the JCC Application represent the interaction between the implemented service and the underlying system, which we have integrated within a single Mobicents JCC-SIP RA.

1- By sending the initial SIP *Invite(B,A)* message, the UAC Party A initiates the signaling exchange to setup a communication session with the SIP Party B (UAS).

2- The *InviteRequestHandler* in the receiving SIP Server sends a 100 Trying provisional acknowledgement back to the sender SIP UA.

3-4- The SIP Server that receives the SIP Invite invokes the *handleInvite()* method of the JCC Provider which, in turn, creates a *JCC Call* instance by invoking the *CreateCallAndConnection()* method, the initial state of which is *IDLE*.

5- The *JCC Call* entity can *optionally* notify the *JCC Application* of its creation by firing an event generated by invoking the callCreated() method.

6- The *JCC Call* entity can then instantiate the *JCC Connection A*, the initial state of which is *IDLE*.

7- The *JCC Connection A* can *optionally* notify the *JCC Application* of its creation by firing an event generated by invoking the connectionCreated() method.

8- The state of the *JCC Call* entity changes to *ACTIVE* and it can *optionally* notify the *JCC Application* of this change by firing an event generated by invoking the callActive() method. The state of the *JCC Connection A* entity changes to *ADDRESS_ANALYZE*. This transition is possible only in the modified JCC Connection FSM shown in Figure 1. Avoiding the Authorize Call Attempt state is due to the typical SIP operation that requests a preliminary AAA phase. This allows saving resources for managing a relevant event by a dedicated listener.

9 - *JCC Connection A* notifies the *JCC Application* of this change by firing an event generated by invoking the connectionAddressAnalyze() method. This is a further proposal, deriving from the observation that in the SIP philosophy a call is typically managed by a service, and not by the underlying system.

10-11- Now the JCC application can invoke the only needed method to establish a call: routeCall(B,A, …). This way the *JCC Call* is required to instantiate the *JCC Connection B* in its initial *IDLE* state.

12- The *JCC Connection B* can *optionally* notify the *JCC Application* of its creation by invoking the connectionCreated() method.

13-14- By the method *sendInvite()*, invoked by the *JCC Call* the *JCC Connection B* triggers the *SIP Server* to send the SIP INVITE(B,A) towards the *SIP Party B*.

15-16- The state of the *JCC Connections A* and *B* change to CALL_DELIVERY. Also this change is possible only in the modified JCC Connection FSM shown in Figure 1. The *JCC Application* can be notified of these changes by the connectionCallDelivery() method. The SIP *Trying* message is not shown for the sake of neatness.

17- The SIP signaling proceeds at the *SIP Party B* by sending the *180 Ringing* message back to the *SIP Server*.

18-20- The *180 Ringing* message triggers the invocation of *handleResponse ()*, *handleRinging()*, and *forwardRinging()* methods.

21-22- The relevant response, implemented by the *sendRinging()* method, cause the SIP *180 Ringing* message to be sent from the *SIP Server* back to the *SIP Party A*.

23- The state of the *JCC Connection B* entity changes to *ALERTING* and it can *optionally* notify the *JCC Application* of this change by firing an event generated by invoking the connectionAlerting() method.

24-27- The SIP Party B can than send a SIP 200 OK message back to the SIP Server, which triggers the invocation of *handleResponse ()*, *handleOK()*, and *forwardOK()* methods.

28-29- The relevant response, generated by invoking the *sendOK()* method, causes the SIP *200 OK* message to be sent from the *SIP Server* back to the *SIP Party A*.

30-35- The *SIP Party A* completes the three-way-handshake by the next *ACK*, which is forwarded to the *SIP Party B* through a set of methods similar to the ones used to carry the SIP 200 OK message.

36-37 - The states of the *JCC Connection A* and *JCC Connection B* change to *CONNECTED*. They notify the *JCC Application* of this change by firing events generated by invoking the *connectionConnected()* method.

In our implementation the entities JCC Call, JCC Connection A, JCC Connection B, JCC Provider, and SIP Server are included within the Mobicents JCC-SIP RA. Their methods are RA internal methods.

From Figure 3, it is evident the complexity reduction achieved. Instead of programming a complete SIP exchange, a single mandatory RouteCall() method is sufficient, thus reducing the implementation time and avoiding the error prone management of the SIP methods.

Digital Object Indentifier 10.1109/MMUL.2011.58 1070-986X/$26.00 2011 IEEE

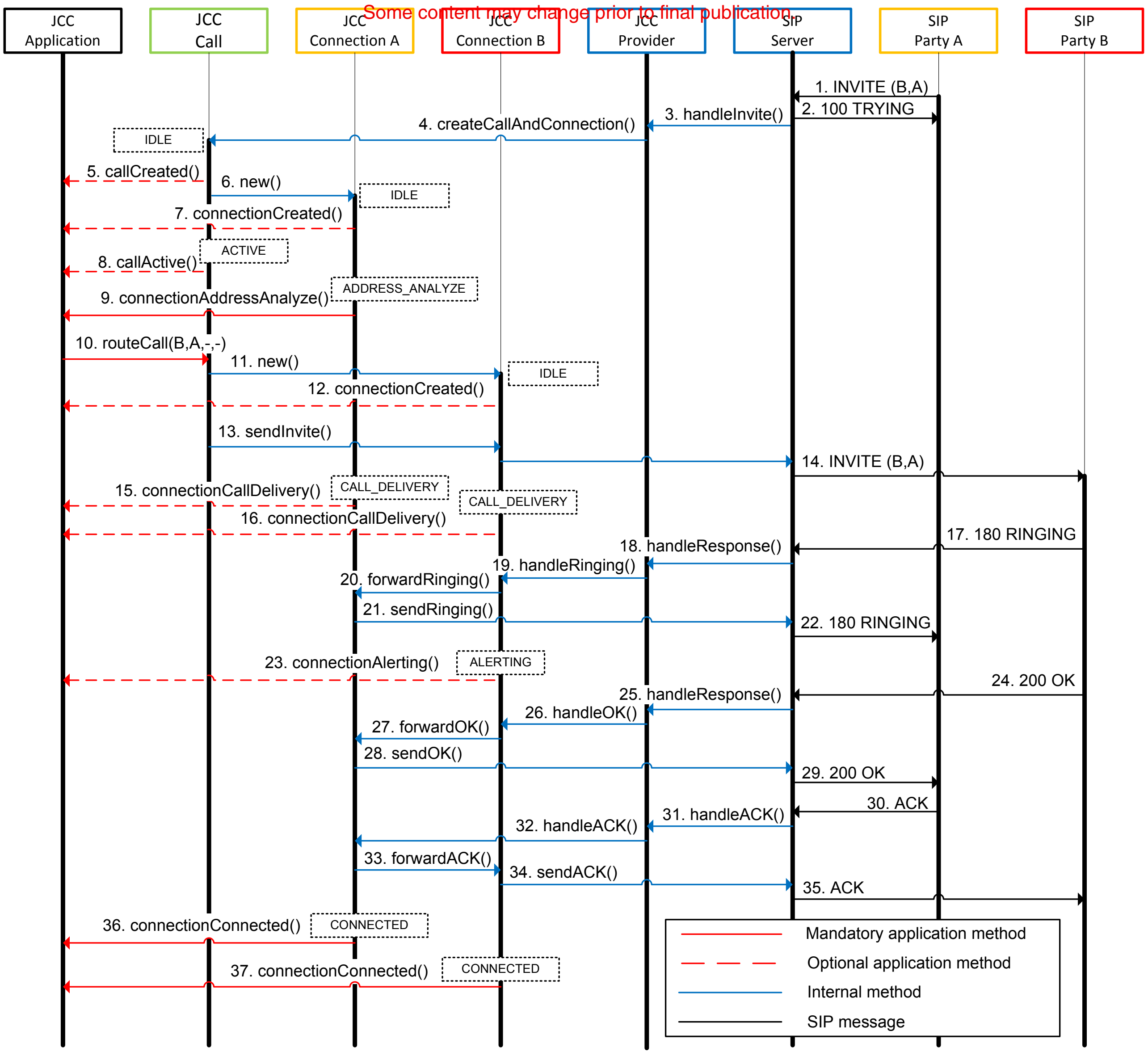

Figure 3 JCC-SIP signaling exchange: point to point telephone call.

The second service consists of a JCC-SIP B2BUA call generated by the *JCC Application*. This service is representative of all services in which third-party applications can establish calls between user terminals, such as click-to-dial services. The signaling exchange is depicted in Figure 4. This example refers to the best practices for implementing third party call control mechanism illustrated in RFC 3725, Flow III [2]. To the best knowledge of the authors, this JCC extension has not been published before.

We assume again that AAA operations have already been executed.

1- The *JCC Application* invokes the *createCall()* method of the *JCC Provider*.

2- After that, the *JCC Provider* instantiates the *JCC Call* in the *IDLE* state.

3- The *JCC Call* entity can *optionally* notify the *JCC Application* of its creation by firing an event generated by invoking the callCreated() method.

4-5 - The *JCC Application* requires the JCC Call to create the JCC Connection A by invoking the method createConnection(), which is created in the *IDLE* state.

6- The *JCC Connection A* can *optionally* notify the *JCC Application* of its creation by invoking the connectionCreated() method.

7- The state of the *JCC call* entity changes to *ACTIVE* and it can *optionally* notify the *JCC Application* of this change by invoking the callActive() method.

8-9- Now the JCC application invokes the method routeCall(B,A, …) to require the *JCC Call* to instantiate also the *JCC Connection B* in the *IDLE* state.

10- The *JCC Connection B* can *optionally* notify the *JCC Application* of its creation by firing an event generated by invoking the connectionCreated() method.

11-12- After that, the JCC Call invokes the method sendInvite() of the JCC Connection A object, which in turn requires the SIP Server to create and send the *SIP INVITE* message towards the *SIP Party A* without any SDP offer.

13- The state of the *JCC Connection A* entity changes to *CALL DELIVERY* and it can *optionally* notify the *JCC Application* of this change by invoking the connectionCallDelivery() method. The relevant SIP Trying message is not shown in Figure 4 for the sake of neatness.

14- The SIP signaling proceeds at the *SIP Party A* by sending the SIP *180 Ringing* message back to the *SIP Server*.

15-16- The *180 Ringing* message triggers the invocation of *handleResponse ()* and *handleRinging()* methods.

17- The state of the *JCC Connection A* entity changes to *ALERTING* and it can *optionally* notify the *JCC Application* of this change by invoking the *connectionAlerting*() method.

18-20- The SIP Party A can send a SIP 200 OK message back to the SIP Server, along with its SDP service configuration parameters. This message triggers the invocation of the *handleResponse ()* and *handleOK()* methods.

21- The subsequent response, sent back by the *JCC Connection A* to the *SIP Server* by invoking the *sendBlackHoledACK()* method, is compliant with the specifications of the RFC 3725 [2]. In particular, the answer is a "black hole" SDP, with its connection address equal to 0.0.0.0.

22- The ACK is propagated back by the *SIP Server* to the *SIP Party A*.

23- The state of the *JCC Connection A* entity changes to *CONNECTED* and it notifies the *JCC Application* of this change by firing an event generated by invoking the *connectionConnected*() method.

24-26- Now the *JCC Connection A* can invoke the *sendInvite()* method to ask the awaiting *JCC Connection B* to require *SIP Server* to send the SIP INVITE message to the SIP Party B without SDP. The state of the *JCC Connection B* changes to *CALL_DELIVERY* and it can *optionally* notify the *JCC Application* of this change by firing an event generated by invoking the *connectionCallDelivery*() method.

27- The SIP signaling proceeds at the *SIP Party B* by sending the SIP *180 Ringing* message back to the *SIP Server*.

28-29- The SIP *180 Ringing* message triggers the invocation of *handleResponse()* and *handleRinging()* methods.

30- The state of the *JCC Connection B* entity changes to *ALERTING* and it can *optionally* notify the *JCC Application* of this change by invoking the *connectionAlerting*() method.

31-34- The SIP Party B can than send a SIP 200 OK message back to the SIP Server, along with its SDP offer. This message triggers the invocation of the *handleResponse()*, *handleOK()*, and *sendReInvite()* methods.

35- After an eventual negotiation for matching the SDP offers, the *JCC Connection A* require the *SIP Server* to re-invite the SIP Party A.

36-38 - The relevant SIP 200 OK message replied by the SIP Party A back to the *SIP Server* triggers again the invocation of *handleResponse()* and *handleOK()* methods.

39-40 - By invoking the *sendACK()* method, the *JCC Connection A* requires the *JCC Connection B* to trigger the SIP Server to send a SIP ACK to the SIP Party B.

41-42 – By invoking the *sendACK()* method, the *JCC Connection A* triggers the SIP Server to send a SIP ACK to the SIP Party A.

43 - The state of the *JCC Connection B* changes to *CONNECTED* and it notifies the *JCC Application* of this change by invoking the *connectionConnected()* method.

Even in this case, from Figure 4 it is evident that the complexity reduction achieved is significant (5 JCC mandatory messages against at least 11 SIP ones).

4. JCC-SIP RA Implementation and Results

The JCC RA for SIP has been implemented as part of the Mobicents SLEE version 1.2.6. It is compliant to the JAIN SLEE 1.0 specifications, the latest stable version at the beginning of this work. The link between the RA and the underlying SIP network has been realized by using the JAIN SIP API and JAIN SIP RI (*Reference Implementation*) v1.2.

In what follows we show the results of the experiments performed by using the service shown in Figure 3, implemented by our JCC-SIP RA.

The SIP UAC and UAS have been implemented by two PCs hosting the Ubuntu Linux operating system v.9.10, running the SIPp [11] traffic generator. Multiple MSLEE v.1.2.6.GA instances have been virtualized by using the VMWare ESXi 4.1 hypervisor, installed on a Fujitsu-Siemens PRIMERGY TX300 S4 server, dual Intel Xeon E5410@2.33GHz, with a total of 8 CPU cores and 16GB of RAM. Each virtual machine (VM) containing an MSLEE runs the CentOS 32-bit v5.5 operating system and a JVM version 1.6.0_21 32-bit Server, with 2 virtual CPUs and 3 GB of RAM. Thus, four MSLEE instances can run in parallel over the shared hardware resources. This deployment has been suggested by the analysis carried out in [5]. Subscriber policies have been stored into a MySQL database v. 5.0.51a. In the same PC hosting the database, a SIP proxy, acting as a load balancer between all MSLEE virtual instances, has been deployed by using the OpenSIPS server [12]. Its task is to route SIP traffic towards MSLEE instances running in different VMs in a round robin fashion. All the test bed devices have been connected through a dedicated Gigabit Ethernet network.

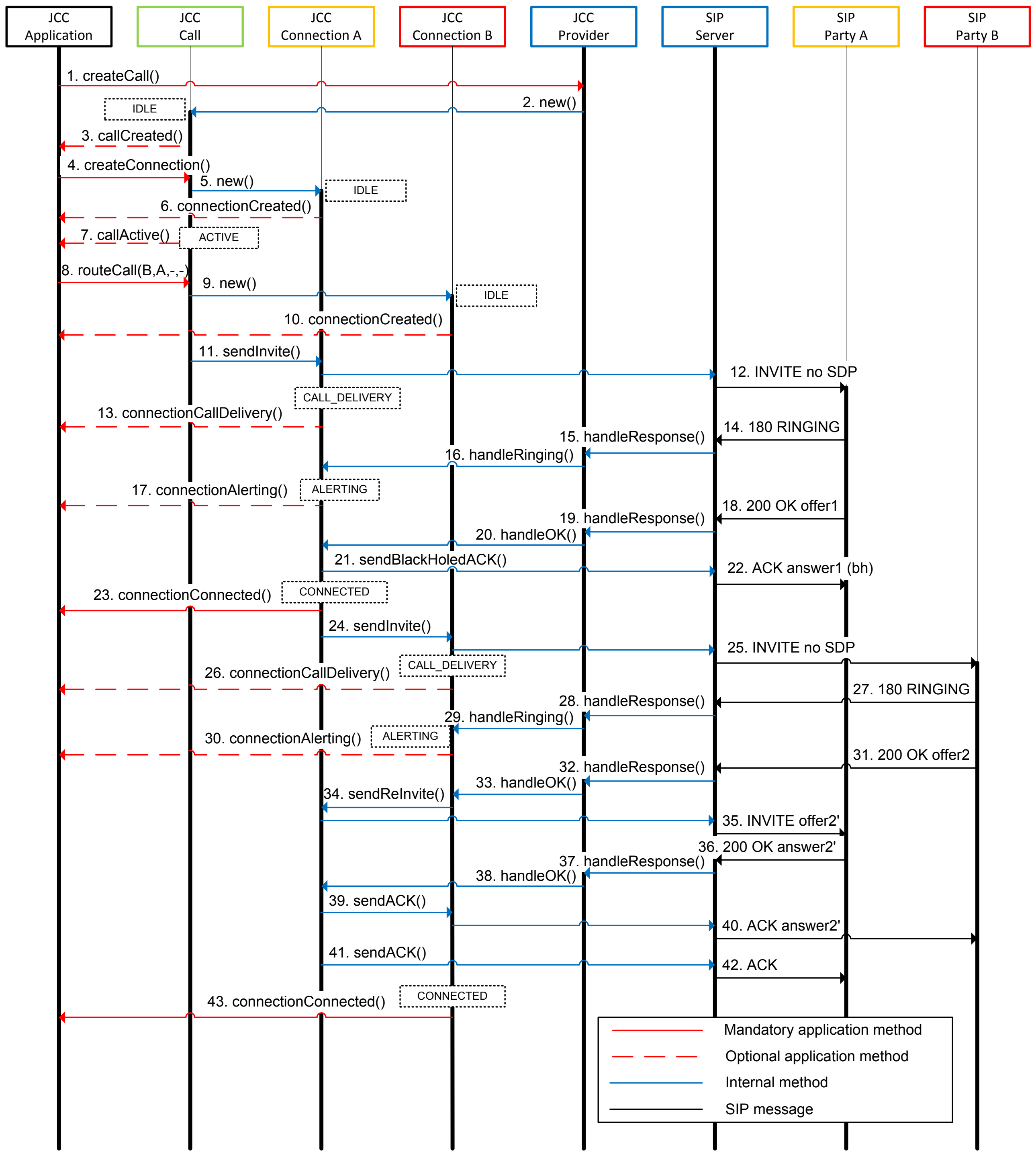

Figure 4: JCC-SIP signaling exchange: third party initiated telephone call.

The service has been tested by optimizing the configuration of both the ESXi VM and the JVM [5].

Table 1 reports the basic JVM parameters for both parallel and concurrent garbage collection configurations and some experimental results.

JSLEE RA and GC Type	JVM Parameters	MCT (cps)	Average SRD (ms)	MCT (cps) with 95th pct ≤ 200 ms	COCOMO (mm)
JCC-SIP RA Parallel GC	-Xms2436m -Xmx2436m -XX:PermSize=64m -XX:+UseTLAB -XX:NewRatio=18	170	421	145	1.24
JCC-SIP RA Concurrent GC	-Xms2436m -Xmx2436m -XX:PermSize=64m -XX:+UseTLAB -XX:+UseConcMarkSweepGC -XX:NewSize=220 -XX:MaxNewSize=220	185	1185	120	1.24
SIP RA	-Xms2500m -Xmx2500m -XX:+UseTLAB -XX:NewRatio=10	135	159	110	1.52

Table 1: JVM configuration and achieved performance

Calls have been generated by the SIPp UAC according to a deterministic arrival process. The call length was 180 seconds, and the duration of each test was 60 minutes. As performance metrics, we report the maximum call throughput (MCT, defined as the maximum input call rate generating a percentage of lost calls lower than 1%), expressed in calls per second (cps), and the average *Session Request Delay* (SRD), defined as the time interval from the initial SIP INVITE to the first non-100 provisional SIP response. SRD values are important since they are related to the latency experienced by a caller initiating a session. Since in operation SDR values need to be bounded, we also reported the MCT value corresponding to the further limitation of generating a 95-th SRD percentile of 200 ms.

Table 1 also reports the Constructive Cost Model (COCOMO), which is a model used for estimating man-month (mm) effort associated with software projects [14], and the performance achieved by the Mobicents SIP RA. It is worth considering that while our implementation is optimized for the implemented functions, the SIP RA includes a larger set of capabilities and relevant control procedures. Thus, even if we have added a further abstraction layer to speed up service implementation and deployment, our implementation can outperform the SIP RA. On the other hand, this result demonstrates the suitability of our proposal and the effectiveness of our implementation. Finally, it is worth noting the both JCC RA and Mobicents SIP RA can be deployed in the same MSLEE and used simultaneously for implementing any service. This could be extremely useful for executing legacy services implemented by using the SIP RA together with services implemented by using our JCC-SIP RA.

5. Conclusion

In this paper we have presented a JCC-SIP mapping and the relevant implementation as a JCC-SIP RA for the Mobicents JSLEE. We have shown the effectiveness of the implemented JCC RA in allowing service developers to realize carrier-grade services in an easier and faster way with respect to creating the same services by directly using the SIP methods. Through a higher layer abstraction provided by the JCC RA, a complex service logic does not have to include specific SIP signaling issues, and the achievable performance results to be very promising.

Further activities for simplifying service design and implementation are ongoing, such as JCC mapping with the Media Gateway Control Protocols.

References

[1] R. Jain, J.-L. Bakker, F. Anjum, “Java Call Control (JCC) and Session Initiation Protocol”, IEICE Transactions on Communications, vol. E-84-B, no. 12, December 2001.
[2] J. Rosenberg, J. Peterson, H. Schulzrinne, G. Camarillo, “Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)” IETF RFC 3725 , April 2004.
[3] RFC 3261, "SIP: Session Initiation Protocol", 2002, http://www.ietf.org/rfc/rfc3261.txt.
[4] RFC 4566, “SDP: Session Description Protocol”, 2006, http://www.ietf.org/rfc/rfc4566.txt.
[5] M. Femminella, E. Maccherani G. Reali, “Performance Management of Java-based SIP Application Servers”, IFIP/IEEE IM’11, Dublin, Ireland, May 2011.
[6] Mobicents web site, http://www.mobicents.org.
[7] Open Cloud web site: www.opencloud.com.
[8] Sun Microsystems, Inc., “Java Call Control (JCC) Application Programming Interface (API) Version 1.0b, Overview of the API”, July 2002
[9] H. Sasaki, J. L. Bakker, P. O’Doherty, “Java Call Control v1.0 to Session Initiation Protocol Mapping”, Oct 2001.
[10] M. Femminella, R. Francescangeli, F. Giacinti, E. Maccherani, A. Parisi, G. Reali, “Scalability and Performance Evaluation of a JAIN SLEE-based Platform for VoIP Services”, ITC 21, 15-17 September 2009, Paris, France.
[11] SIPp Web Site: http://sipp.sourceforge.net.
[12] OpenSIPS Project Web Site: http://www.opensips.org.
[13] Amdocs Web Site, available at: http://www.amdocs.com/Products/Service-Delivery/Convergent-Service-Platform/Pages/index.aspx.
[14] B. Boehm, “Software Engineering Economics”, Englewood Cliffs, Prentice-Hall, 1981.

[15] H. Khlifi, J.-C.Gregoire, "IMS application servers: roles, requirements, and implementation technologies," IEEE Internet Computing, May-June 2008, 12(3), pp. 40-51.

[16] R. Jain, F.M. Anjum, P. Missier, S. Shastry, "Java call control, coordination, and transactions", IEEE Communications Magazine, 38(1), pp. 108-114.R.

[17] Jain, J.-L. Bakker, and F. Anjum "Programming Converged Networks: Call Control in Java, XML, and Parlay/OSA", Wiley, 2004.

Authors' biographies

Mauro Femminella received both the master degree and the Ph.D. in Electronic Engineering from University of Perugia in 1999 and 2003, respectively. Since November 2006, he is assistant professor at the Department of Electronic and Information Engineering, University of Perugia. His current research interests focus on nano-scale networking and communications, middleware platforms for multimedia services, location and navigation systems, and network and service management architectures for the Future Internet.

Francesco Giacinti received the master degree in Telecommunication and Computer Engineering from University of Perugia in 2011. Since September 2011, he is contract researcher at Department of Electronic and Information Engineering, University of Perugia His research interests focus on the design and performance evaluation of middleware platforms for multimedia services.

Gianluca Reali is an associate professor at the University of Perugia, Department of Information and Electronic Engineering (DIEI), Italy, since January 2005. He received the Ph.D. degree in Telecommunications from the University of Perugia in 1997. From 1997 to 2004 he was researcher at DIEI. In 1999 he visited the Computer Science Department at UCLA. His research activities include resource allocation over packet networks, wireless networking, network management, and multimedia services.

Contact Information

Mauro Femminella
Department of Information and Electronic Engineering - DIEI
University of Perugia
Via G. Duranti, 93, Perugia, Italy
mauro.femminella@diei.unipg.it
Tel. +39 075 585 3630

Francesco Giacinti
Department of Information and Electronic Engineering - DIEI
University of Perugia
Via G. Duranti, 93, Perugia, Italy
francesco.giacinti@diei.unipg.it
Tel. +39 075 585 3647

Gianluca Reali (contact author)
Department of Information and Electronic Engineering - DIEI
University of Perugia
Via G. Duranti, 93, Perugia, Italy
mauro.femminella@diei.unipg.it
Tel. +39 075 585 3651