IEICE TRANS. COMMUN., VOL.E84–B, NO.12 DECEMBER 2001

INVITED PAPER *Special Issue on Network Software*

Java™ Call Control (JCC) and Session Initiation Protocol (SIP)

Ravi JAIN†, **John-Luc BAKKER**†, *and* **Farooq ANJUM**†, ***Nonmembers***

SUMMARY This paper describes the JAIN™ Java™ Call Control (JCC)* Application Programming Interface (API), and its relationship to network protocols, in particular the Session Initiation Protocol (SIP). JCC is a high-level object-oriented open, standard API for Next Generation Network (NGN) softswitches that enables rapid creation, by third parties, of services that can run independently of the underlying network technology (e.g. wireless, wired, packet, IP, PSTN) and protocols. SIP is a protocol that has been proposed for a wide variety of uses in IP networks, including call control. We argue that instead of being competitors, JCC and SIP are complementary, with JCC offering higher-layer programming abstractions and protocol-independence, and demonstrate by examples how to map JCC version 1.0 to a SIP environment. We thus show that for common call control applications using JCC is simpler, faster and less maintenance intensive than using SIP directly.

key words: *Java call control (JCC), session initiation protocol (SIP), call agent, application programming interface (API), next generation network (NGN)*

1. Introduction

Future telecommunications networks will be characterized by new and evolving architectures where packet-switched, circuit-switched, wired and wireless networks are integrated to form what is called a Next Generation Network (NGN), and where the NGN will offers subscribers an array of innovative multimedia, multi-party applications. Equally importantly, it is expected that service providers will increasingly turn to third-party applications developers and software vendors in order to provide a broad portfolio of novel, compelling applications rapidly. To enable this vision NGN will offer a set of standard, open Application Programming Interfaces (APIs) so that applications are portable across vendor systems, thus reducing development time and cost.

JAIN is a community of companies led by Sun Microsystems that is developing standard, open, published Java APIs for NGN [13], [14]. These APIs include interfaces at the protocol level, for different protocols like MGCP, SIP, H.323, ISUP and INAP, as well as at higher layers of the telecommunications software stack. One of the key higher-layer APIs for NGN is for *call control*, and is called Java Call Control (JCC), where a "call" refers to a multimedia session over an NGN, i.e., a IP, Public Switched Telephone Network (PSTN), wired or wireless network. JCC [12] applies to NGN elements such as *Call Agents* or *softswitches*.

Recently, the Session Initiation Protocol (SIP) being standardized by the IETF has been proposed as a protocol for call control in NGN generally and IP networks in particular [7]. While SIP is sometimes seen as an alternative to higher-layer APIs like JCC, we believe that JCC and SIP are complementary. JCC offers the application programmer the rapid development, portability, software reusability and ease of use of high-level object-oriented programming, and allows the application to run on systems with different underlying protocols (e.g. MGCP, H.323, ISUP, INAP, in addition to SIP). SIP, on the other hand, offers the programmer fine-grained control and interoperability. To some extent, programming using JCC versus SIP is roughly analogous to programming in high-level languages versus assembly. In fact, we argue that a likely implementation scenario is that JCC is implemented on top of a SIP layer, thus potentially combining the advantages of both.

We have previously implemented a prototype call-processing platform in 100% pure Java that completes basic calls, performs advanced services, and also allows dynamic service deployment [1]. The high-level API that the prototype offered was similar in spirit to JCC, and the underlying network signalling protocol we used was SIP. In this paper we briefly describe the standard JCC API and then sketch how JCC can be implemented using SIP, i.e., a *mapping* from JCC to SIP. The JAIN Expert group on JCC is currently generating mappings from JCC to several protocols, including H.323, ISUP, and INAP protocols in addition to SIP, further substantiating the protocol-independence of JCC. In addition, a Telcordia experimental research prototype of an NGN softswitch implements a JCC programming interface while using SIP for IP network signalling (in addition to MGCP for interaction with endpoints).

This paper is organized as follows. The next section briefly clarifies some terminology and the distinctions we make between protocols like SIP and APIs like

Manuscript received June 14, 2001.

†The authors are with Telcordia Technologies, Inc., 445 South Street, Morristown NJ 07960, USA.

*Java and JAIN are trademarks of Sun Microsystems.

JCC. Section 3 provides a brief overview of JCC, and Sect. 4 describes the mapping from JCC to SIP. The mapping is described both in terms of mappings from SIP messages to JCC states as well as by two examples: First party call setup and how an application that monitors call redirection (forwarding) in a SIP environment would be written using JCC. Finally, Sect. 6 ends with a discussion of the mapping and relative merits of JCC and SIP, and brief remarks about future work.

2. Call Models and Protocols

It is worthwhile to first clarify some terms and concepts that are often misused or used interchangeably in this area. Telecommunications applications have been built using a variety of call models, such as (Advanced) Intelligent Network (A/IN) Basic Call State Model and the Java Telephony API (JTAPI) [2]. We regard a *call model* as an abstract virtual machine for building telecommunications and NGN applications, and the corresponding API as an interface to that machine [1].

An API like JCC thus represents a "horizontal" interface between the software layers of a functional platform (even if the different layers are physically executed on separate machines). The aim of the API is to allow application *portability*; an application can run on any vendor's platform as long as the platform correctly implements the API. In contrast, a protocol like SIP provides a "vertical" interface between separate platforms connected via a network. The aim of the protocol is to provide *interoperability*; two platforms can communicate even if they have different hardware, operating systems or implementation languages as long as they obey the protocol.

3. Java Call Control

The JCC Expert Group [12] has developed an API that provides an interface to a generic call model that provides the applications programmer with a convenient and powerful abstraction for manipulating calls and managing the interaction between the application and calls. In addition, the API is extensible, so that as additional functions are required, they can be added incrementally and in a modular fashion to the API. A specific requirement of the JCC API is that it be *protocol-independent*, i.e., applications need not be aware of the network protocol used to implement the API.

The JCC API is a Java interface for creating, monitoring, controlling, manipulating and tearing down communications sessions in a converged PSTN, packet-switched, and wireless environment. It provides facilities for first-party as well as third-party applications, and is applicable to network elements (such as switches or Call Agents) both at the network periphery (e.g. Class 5 or end-office switches) and at the core (e.g. Class 4 or tandem switches). JCC allows applications to be invoked or triggered during session set-up in a manner similar in spirit to the way in which IN or AIN services can be invoked. Example applications that are required to be supported by the API include first and third-party session origination and termination, Voice Virtual Private Networks (VVPN), and toll-free number translation. Applications may execute on the JCC platform itself (e.g. a softswitch or Call Agent) or in a distributed manner using any underlying distribution technology, e.g. Java Remote Method Invocation (RMI).

The API is not intended to open up telecommunications networks' signaling infrastructure for public usage. Rather, network capabilities are intended to be encapsulated and made visible using object technology in a secure, manageable, and billable manner. This approach allows the network to execute third party applications without compromising network security, integrity, and reliability. JCC is intended to be consistent with the call control APIs issued by the JTAPI and Parlay [3] groups, e.g. Parlay's Multi-Party Call Control Service (MPCCS). For further details on the current status of this ongoing standards alignment process, see [5]. Note that this paper is based on JCC version 1.0 and future versions of JCC may differ from this paper as the alignment process proceeds and other changes are made. Also note that the brief description of JCC here is only to assist the reader; the current specification [12] is the definitive source.

JCC defines four objects, which model the key call processing objects manipulated by most services; the Provider and Address are logically static (although for performance reasons the implementation may instantiate the latter dynamically), while the Call and Connection are logically created dynamically for each call (although for performance reasons the implementation may choose to instantiate them from a pool of pre-instantiated reusable objects). In the rest of this paper, physical generic entities are denoted by lower-case letters (e.g. "call"), while the logical objects representing them in the JCC model are capitalized (e.g. "Call"). The Provider represents the "window" through which an application views the call processing. The Address represents a logical endpoint (e.g. directory number or IP address). The Call represents a call and brings two or more endpoints together. The Connection represents the dynamic relationship between a Call and an Address and is used to model the detailed progress of call processing for a particular call leg.

The relationship among the JCC objects is depicted pictorially in Fig. 1 for a two-party call. Multiple parties are represented—with no inherent limitations in the model—simply by additional Connections and Addresses associated with a Call. The Connection object contains a Finite State Machine (FSM) that models call processing (see Fig. 2) in a manner similar to the JTAPI Connection object, but differing in many details.

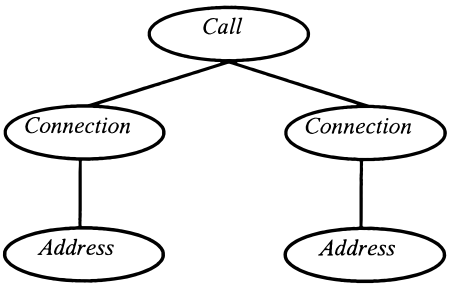

Fig. 1 JCC object model of a two-party call.

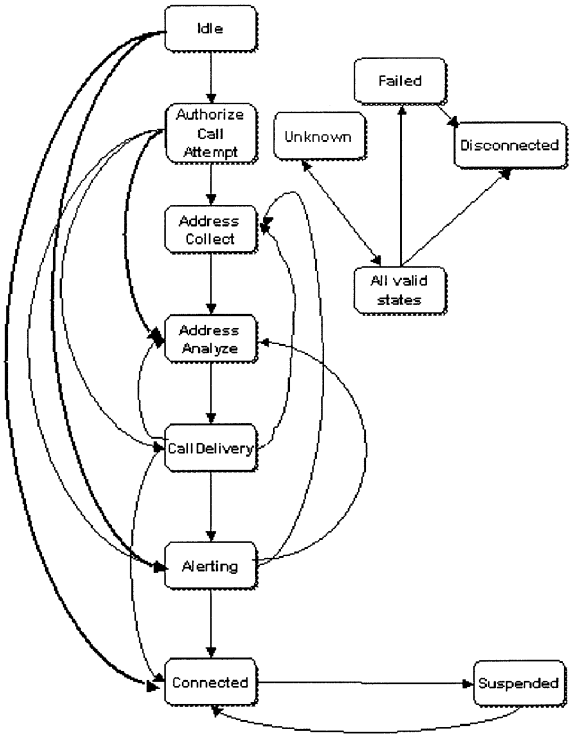

Fig. 2 JCC 1.0 connection object FSM.

The basic paradigm for an application interacting with a JCC platform is that the application makes synchronous method calls on the JCC objects and uses the Listener pattern for asynchronous notifications from JCC, i.e., the application provides a Listener object to which JCC can report Java Events. Thus for example if the application registers for the AddressAnalyze event for a particular Address, JCC issues a Java event to the application's Listener when the corresponding Connection object FSM enters the ADDRESSANALYZE state.

4. Java Call Control and SIP

This section briefly gives a descriptive mapping from JCC to SIP. We do not consider other protocols e.g. H.323, ISUP, and INAP, although the JCC Expert group is generating such mappings. A similar document

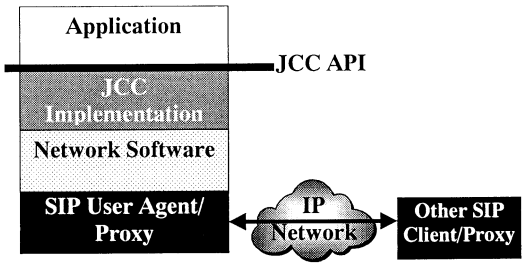

Fig. 3 Architecture for mapping JCC to SIP.

[6] explores the mapping from Parlay's Multi Party Call Control Service (MPCCS) to SIP.

SIP [7] is an Internet protocol originally designed for session initiation. It is transport independent and text-based, and aims to deal with the signaling associated with establishing calls between parties. Once a call is established then many different types of media stream may be used, e.g. H.323.

The architecture considered in this document is depicted in Fig. 3†. The top layer consists of JCC applications. The applications invoke methods on and register interest in particular events with the JCC implementation. The JCC implementation runs on a proprietary Network Software stack. The Network Software is 'glued' to the protocol; its responsibility is to map the state of the network as seen by the protocol to JCC states and vice versa. All relevant JCC processing occurs within the JCC implementation; the application view on call processing is impacted by JCC method invocations or by interactions occurring between the Network Software layer and the protocol implementation. Finally, protocol messages are sent between the network elements, in this case by a SIP protocol stack implementation. Note that the SIP protocol software could be encapsulated in the standard JAIN SIP API [9], but this need not be the case.

4.1 SIP

We give a brief description of SIP here so as to explain the JCC to SIP mapping. The main network entities in SIP are the User Agent, the Proxy Server, the Redirect Server and the Registrar. User Agents communicate with other User Agents directly or via an intermediate server. The User Agent also stores and manages call states.

SIP intermediate servers have the capability to behave as Proxy or Redirect Servers. Proxy Servers forward requests from the User Agent to the next server or User Agent within the network. Redirect Servers

†This architecture is not the only architecture that applies when mapping a SIP flavor to JCC. For example, JCC could be implemented on top of SPIRITS [8]. However, this interesting subject is not pursued in this paper due to space constraints.

Table 1 SIP message types.

SIP	Description
INVITE	Invites a user to a call or establishes a new call
CANCEL	Terminates a request or the search for a user
BYE	Terminates a call between two users or to decline an invitation
ACK	Acknowledges receipt of an INVITE message
REGISTER	Conveys information about a user's location to a SIP Registrar
OPTIONS	Solicits information about a SIP servers' capabilities

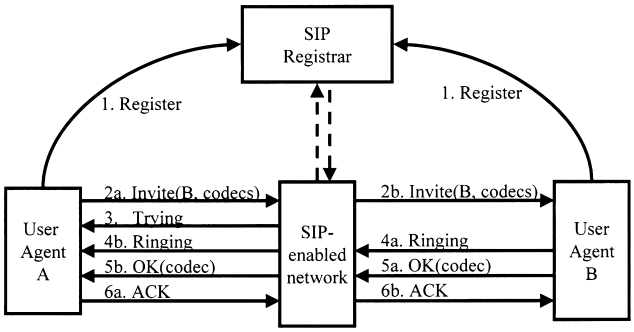

Fig. 4 Message sequence for a "canonical" SIP call setup.

respond to client requests and inform them of the requested server's address. Numerous hops can take place until reaching the final destination. SIP allows the servers to contact external location servers to determine user or routing policies, and therefore, does not bind the user into only one scheme to locate users. In addition, to maintain scalability, the SIP servers can either maintain state information or forward requests in a stateless fashion.

SIP supports six different main messages (see Table 1). Numerous other messages exist, categorized either as informational (1xx)†, successful (2xx), redirection (3xx), request failure (4xx), server failure (5xx), or global failure (6xx). Within this paper, mapping all these messages to JCC is not carried out due to space constraints.

The operation of a "canonical" SIP call setup is shown in Fig. 4, assuming that user A is trying to initiate a session to user B. (Once again, this example is shown for illustration only; the reader is referred to the SIP specification [7] for definitive information). The numbers in the arrow labels indicate the message sequence.

To simplify presentation we assume the network is "SIP-enabled" in the sense that there is a collection of SIP Proxies, Agents, etc. that can carry out SIP functions. The User Agents for each party must first have registered with the SIP Registrar. User Agent A sends an INVITE message to Agent B, which contains among other items a list of acceptable codecs. Entities in the SIP-enabled network may send a TRYING message back to Agent A. After Agent B indicates that it has been alerted (RINGING) and parameter negotiation is successful (OK), Agent A sends an acknowledgement (ACK) and the users can communicate.

4.2 Mapping

JCC is intended to be protocol agnostic, and to support a variety of underlying signaling protocols in addition to SIP. Therefore the mapping of JCC to SIP does not expose all SIP messages and parameters, but only those that present a common abstract view of call processing. Thus, in the following, some SIP messages are not mapped to JCC, and some SIP parameters containing media or routing information are not exposed.

The following table (Table 2) gives a mapping of messages received by a SIP application server and translated into JCC events. Applications receive these events if they have registered for the appropriate JCC events with the API implementation. A JCC event typically signals a JCC Connection object FSM state transition. We briefly describe Connection object states relevant to the mapping: IDLE, CALL_DELIVERY, ALERTING, CONNECTED, and DISCONNECTED (see Fig. 2). IDLE signals that the Connection object was just created, in CALL_DELIVERY a connection is routed to the terminating party's address, in ALERTING the terminating party's terminal is alerted (e.g. ringing), in the CONNECTED state user information flows from one party to the other, and in the DISCONECTED state the party to which this connection was connected has stopped participating in the call. See [12] for further elaboration on the state definitions and specifics of the JCC specification.

5. Java Call Control to SIP Mapping

In this section we use message sequence charts (or "call flows") to illustrate the behavior of the mapping. We illustrate two applications, one for the simple case where a first-party call is placed using the API, and the second for a much more complex situation where an application is monitoring SIP activity, e.g. for billing, Quality of Service (QoS) monitoring, performance evaluation, or testing purposes. Further examples of JCC usage are given in the Call Flow document available in conjunction with the API specification itself [5]. Descriptive JCC to SIP, JCC to H.323, JCC to ISUP, and JCC to INAP mappings will be made available by the JAIN consortium.

†The 1xx denotes the messages status code; e.g. 100 (Trying), 180 (Ringing), etc.

Table 2 Mapping main SIP messages to JCC events.

SIP	JCC Events Received by the Application
INVITE	If this INVITE is sent: (a) at the start of a new session. A Call object, as well as the originating and terminating Connection objects are created (in that order) (b) by a party in an existing session. Thus a Call object with two or more Connections already exists. One new Connection is added to the Call. The originating address attribute for that Connection is set to the address of the party sending the INVITE. (c) to a destination corresponding to an address already engaged in a call. We assume in this case that by some means the destination Connection is in the SUSPENDED state. Then this INVITE causes the Connection to re-enter the CONNECTED state.
CANCEL	Suppose the CANCEL message is sent from a particular address, *A*. Then the Connection object attached to *A*'s Address object transitions to the FAILED state.
BYE	Suppose the BYE message is sent from a particular address, *A*. Then the Connection object attached to *A*'s Address object transitions to the DISCONNECTED state.
ACK	If the Connection originating the ACK message is in (a) the CALL_DELIVERY state, then both it and the terminating Connection transit to CONNECTED. (b) the CONNECTED state (e.g. in a 3-party call) only the terminating Connection transits to the CONNECTED state.
REGISTER	There is no JCC equivalent event or state transition. The impact of this message is that an address registered using this message is now a valid address within the JCC Provider's domain.
OPTIONS	An OPTIONS message is similar to an INVITE message except that the user agent that receives the message does not alert the user, but indicates how it would reply to an INVITE. The information in the returned message includes media information and may indicate if the user is busy or unavailable. JCC applications are shielded from this information.

The diagrams should be read as follows: the colours of the boxes along the top correspond with the colours of layers in Fig. 3. A box along the top of the figure represents one or more objects. Solid arrows represent standardized interactions: those on the right side of the figure represent SIP messages, those on the left represent JCC method invocations. Dotted arrows are not standard; they represent internal object creation methods and invocations that represent one possible implementation of the mapping. Boxes in the central flow of the diagram (e.g. "IDLE") represent state transitions of Connection objects.

5.1 First Party Call Setup

Figure 5 shows a JCC application (shown on the extreme left) for a party A setting up call to a party with address B. The application creates a Call using the *createCall*() JCC API method invocation, and routes a Connection using the *routeCall*() method. In response, the JCC implementation sends an INVITE and handles the subsequent protocol messages. (For simplicity we do not show the initialization steps where the JCC application obtains a reference to the Provider object using the common Java Factory pattern.) The internal method *new*() is the standard Java method for instantiating an object.

5.2 Redirection Monitored by Application

In this section we describe a more complex application and scenario. From the user's perspective, user A calls user B, who has redirected (forwarded) her calls to C. A JCC application could carry out this forwarding function using API methods defined in the specification. However, we describe a more complicated scenario, where the JCC application we are interested in is monitoring the redirection process e.g. for billing, QoS monitoring, performance evaluation or testing purposes.

The JCC monitoring application (see Fig. 6) is executing on top of a state full SIP server (called simply the SIP Server in the figure) that intercepts all SIP messages flowing between A, B and C, each of which in turn is executing on a SIP platform. When user A calls user B, the SIP server receives an INVITE message, resulting in a JCC Call object and Connection objects being created and the JCC application being notified via a Java ConnectionCreated event. The JCC implementation forwards the INVITE to the B party. However, the B party has instructed its user agent to return a MOVED message. The receipt of this message at the monitoring application results in disconnecting Connection B and creating Connection C (it is assumed that the JCC application has previously registered interest in these events and is thus notified of these state changes; we have omitted this initialization step for simplicity). These events do carry meta-information; the cause code CAUSE_RE-DIRECTED. With this information the application can tell a redirection has occurred. Subsequent processing is similar to the First party call setup example in many respects.

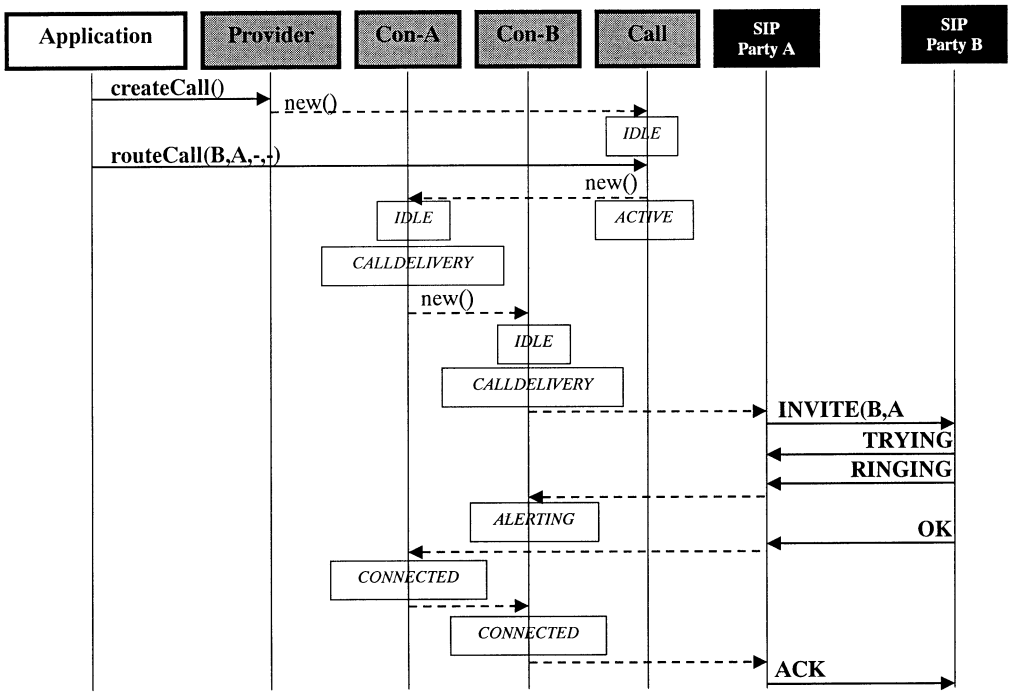

Fig. 5 First party call setup.

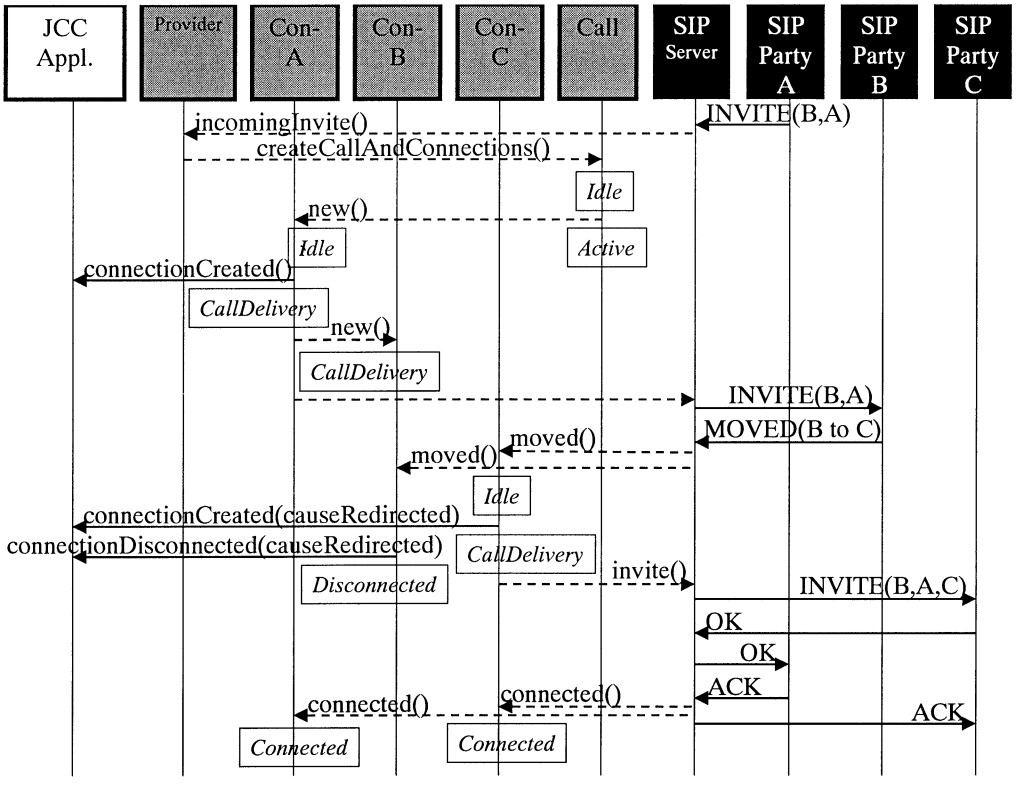

Fig. 6 Redirection monitored by application.

6. Discussion and Concluding Remarks

The aim of this paper is to demonstrate that SIP and JCC are complementary. SIP controls the actual network entities where as JCC abstracts from protocol details and aims at simplifying and accelerating service creation. We have described how a JCC API fits on top of a SIP platform. Using the message-to-event mappings and the call flows of the previous section, we have

shown that JCC can map to SIP and vice versa. The Telcordia mini Call Agent software prototype in fact implements JCC on top of a SIP platform.

The call flows in the previous section also demonstrate a key advantage of using JCC rather than SIP, namely the abstraction and simplicity it provides the application programmer. Thus for setting up a simple two-party call only two key JCC method invocations are necessary, and the programmer need not be concerned with the strict patterns and timing constraints of the underlying protocol. Of course, this simplicity comes at a price—some of the information conveyed by SIP is not made available to the programmer. However, for applications requiring these details the programmer can program to SIP (e.g. the JAIN SIP API) or to a richer call model API such as JCAT [4]. For most common uses however, programming to JCC is faster and less error-prone than programming directly to SIP. This is particularly the case as SIP has expanded recently to cover a very wide range of diverse applications, including handling mobility, inter-softswitch communications, the softswitch-application server interface, etc. (In fact there is a document in progress guiding developers of SIP extensions [10].)

Since the JCC platform implementation takes care of maintaining call processing state, the application is freed from the tedious, complex and repetitive details involved. Thus applications written to JCC will typically be more modular and have a smaller footprint (code size) than corresponding applications using SIP directly.

Finally, JCC is protocol agnostic; it allows to be implemented on other protocols, e.g. MGCP, INAP, etc. Documents describing these mappings are forthcoming from the JAIN Expert group.

In further work we are pursuing the relationship of JCC to other efforts such as IETF SPIRITS [8], richer call models such as JCAT [4], and service creation environments and markup languages [11].

Acknowledgments

We thank Mr. Satoshi Shiraishi, Mr. Kenichiro Matsumoto and Mr. Hisayoshi Inamori for their kind invitation to contribute this article. We thank Gary Levin, Phil Ber, and Simon Tsang of Telcordia for useful discussions and information. Finally, we thank the team members of JAIN Japan for their ongoing efforts in the JCC-SIP mapping.

References

[1] F. Anjum, F. Caruso, R. Jain, P. Missier, and A. Zordan, "CitiTime: A system for rapid creation of portable next-generation telephony services," Computer Networks, vol.35, pp.579–595, 2001.
[2] S. Roberts, Essential JTAPI, 555, Prentice Hall, 1999.
[3] The Parlay Group, See http://www.parlay.org
[4] Sun Microsystems, "JAIN Java coordination and transaction (JCAT) API Java specification request (JSR) 122," 2001. See http://jcp.org/jsr/detail/122.jsp
[5] Telcordia Technologies, Inc., "JAIN @ Telcordia," JAIN Reference Implementations download site, 2001. See http://www.argreenhouse.com/JAINRefCode/
[6] British Telecommunications, plc., R. Stretch, 3GPP TSG_CN5 (Open Service Architecture—OSA). See http://www.3gpp.org/ftp/TSG_CN/WG5_OSA/TSGN5_09_Helsinki/Docs/N5-010016_parlayandsip.zip
[7] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session initiation protocol, Request for Comments: 2543," Internet Engineering Task Force (IETF) Network Working Group, March 1999.
[8] Internet Engineering Task Force (IETF) Working Group Charters, SPIRITS Working Group, 2001. See http://www.ietf.org/html.charters/spirits-charter.html
[9] Sun Microsystems, "JAIN session initiation protocol (SIP) API Java Specification Request (JSR) 32," 2001. See http://jcp.org/jsr/detail/32.jsp
[10] J. Rosenberg and H. Schulzrinne, "Guidelines for authors of extensions," draft-ietf-sip-guidelines-01.txt (work in progress), March 2000.
[11] Sun Microsystems, "JAIN service creation environment (SCE) API Java Specification Request (JSR) 100," 2001. See http://jcp.org/jsr/detail/100.jsp
[12] Sun Microsystems, "JAIN Java call control (JCC) API Java Specification Request (JSR) 21," 2001. See http://jcp.org/jsr/detail/21.jsp
[13] R. Jain and F. Anjum, "Java call control," in Java in Telecommunications, ed. T. Jepsen, Wiley, 2001.
[14] R. Jain, F. Anjum, P. Missier, and S. Shastry, "Java call control, coordination and transactions," IEEE Commun., vol.38, no.8, pp.106–114, Jan. 2000.

Ravi Jain received the Ph.D. in Computer Science from the University of Texas at Austin in 1992. Currently he is Director of the Middleware and Mobile Applications Research Group at Applied Research, Telcordia Technologies. His interests include programmability, middleware and applications for Next Generation Networks, mobile Internet access and applications, and mobile and wireless networking. Jain has numerous publications in these areas and several issued and pending patents. He is also Edit Lead for the JAIN expert group defining Java Call Control (JCC) 1.0 and JCAT 1.0. Jain is a member of the Upsilon Pi Epsilon and Phi Kappa Phi honorary societies, a senior member of IEEE, and a member of ACM. He can be reached at rjain@telcordia.com.

John-Luc Bakker is a Research Scientist in the Middleware and Mobile Applications Research Group at Applied Research, Telcordia Technologies, in Morristown, NJ, USA. He holds an M.S. degree in programming aspects of distributed and parallel computing from the Delft University of Technology, The Netherlands. Mr. Bakker published several papers in the area of component based service creation and advanced network architectures. Currently, he is co-Edit Lead for JAIN Service Creation Environment (SCE) expert group, and expert in Parlay services and framework. He can be reached at jbakker@telcordia.com.

Farooq Anjum is a Research Scientist at Telcordia in the Middleware and Mobile Applications research group. He is currently a co-PI in a DARPA project related to the design of compensating middleware for intrusion tolerance. As part of this project Anjum is studying different techniques to provide intrusion tolerance in middleware packages. He is also active in several projects and has several papers related to investigations into the use of software agent technology, application layer multicasting, personal communications networks and Bluetooth. Anjum is also an expert in the Java APIs for Advanced Integrated Networks (JAIN) standards group. Farooq joined Telcordia after completing a Ph.D. in Electrical and Computer Engineering from the University of Maryland at College Park in 1999. He can be reached at fanjum@telcordia.com.