UNIVERSITÀ DEGLI STUDI FIRENZE

FACULTY OF ENGINEERING
DEPARTMENT OF INFORMATION ENGINEERING

Ph.D. in INFORMATICS, SYSTEMS AND TELECOMMUNICATIONS
CYCLE XXVII

CURRICULUM: TELEMATICS AND INFORMATION SOCIETY
COORDINATOR: PROF. LUIGI CHISCI

Approaches for the service composition and description towards the Web-Telecom convergence

ING-INF/03

Ph.D. Student	*Tutors*	*Coordinator*
Terence Ambra	Prof. Alessandro Fantechi	Prof. Luigi Chisci
	Prof. Dino Giuli	
	Dr. Federica Paganelli	

Years 2012/2014

Acknowledgements

I wish to thank my tutors Prof. Dino Giuli and Prof. Alessandro Fantechi for giving me the opportunity to perform this exciting thesis by encouraging my research and for allowing me to grow professionally and personally. I would also like to thank Ing. Federica Paganelli, for her advice and continuous help in this long thesis work. An affectionate embrace goes to all the people of the laboratory *"Radar and Telecommunications"* with the which I have established a beautiful friendship.

Finally, a special thanks to my parents for the countless sacrifices and efforts made in this long university path.

Firenze, 31 Dicembre 2014

Terence Ambra

Abstract

Several approaches are currently being discussed for the convergence of Web and Telecommunication services. For instance, research and industry stakeholders have recently proposed Web-based APIs to control real-time communication among SIP User Agents. The IETF and W3C standardization bodies are investigating how web browsers should evolve to natively support communication services. In this perspective, the design of novel mechanisms for the exchange of signaling messages and possible interworking between Web-based and SIP-based systems is a hot topic of research. Indeed, the discussion is still ongoing on how differences between REpresentational State Transfer (REST) and Session Initiation Protocol (SIP) models should be coped with. This issue is made more difficult by the lack of rigorous modeling of RESTful systems. In this PhD thesis we discuss how we applied a REST-oriented methodology to design a set of REST APIs for communication services (e.g. a voice call and presence service). The contribution of this work is threefold. Firstly, we formalize the call resource behavior through a Finite State Machine representation which accounts for the SIP specifications and for REST constraints. Secondly, we simulate the service expected behavior and its interworking with SIP User Agents through a tool for the analysis of communicating state machines. Thirdly, we present the implementation details of a web application prototype and evaluate its functional correctness and performance. This prototype supports three mechanisms for handling asynchronous notifications (i.e., WebSocket, Long Polling and HTTP Streaming).

Contents

List of Figures

List of Tables

Introduction

The Web is shifting from a document-centric paradigm to an increasingly interactive and collaborative form providing information sharing and real-time communication. Indeed, the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) standardization bodies are defining recently WebRTC API and protocol specifications [Bergkvist et al., 2014] to allow the native support of voice and video communications by web browsers.

In the telecommunication domain, the research and industry communities have defined several web-based APIs to expose on the Web multiparty or peer-to-peer communication services provided by legacy telecommunications platforms, such as systems based on the Session Initiation Protocol (SIP) [Rosenberg et al., 2002a]. In this context, the REpresentational State Transfer (REST) design style [Fielding, 2000] is considered a best practice for building distributed hypermedia systems and APIs oriented to web. REST principles have been applied to design communication services in standardization efforts, such as in the RESTful bindings for Parlay X Web Services [OMA, 2012] and the OneAPI REST interfaces [GSMA, 2009], as well as in several research works, such as those discussed in the survey by Belqasmi et al. [2011].

Among telecommunications legacy frameworks, the architectures based on SIP protocol definitely play a major role, as argued by Amirante et al. [2013]. The interworking between the emerging browser-enabled systems and SIP-based ones is thus a hot issue of research [Amirante et al., 2013]. Actually, the convergence of HTTP and SIP domains is not straightforward since these protocols rely on different principles. In fact, SIP is a stateful and peer-oriented protocol, while HTTP is stateless and based on the client-server model [Bond et al., 2009, Islam and Gregoire, 2013]. Several authors have worked on this topic and have discussed the design of APIs based on REST principles for the convergence of Web-centric and Telecom-centric services [Li and Chou, 2010, Davids et al., 2011, Griffin and Flanagan, 2011a]. However, the potential impact of these related works is weakened by the general lack of rigorous modeling of REST principles and related RESTful systems. As argued by Zuzak et al. [2011] this is causing a widespread misunderstanding of REST concepts and a resulting difficulty in fully taking advantage of REST benefits (e.g., scalability, interoperability and simplicity).

To address these limitations, in this work we present a set of REST APIs purposely conceived for communication services (e.g., call and presence service) that interworks with SIP-based systems. By leveraging a resource-oriented service design methodology, our original contribution is threefold. Firstly, we model the resource behavior through a Finite State Machine representation which accounts for the SIP specifications of a call session setup and possible error conditions and for REST constraints. Secondly, we simulate the behavior of the RESTful call service and its interworking with SIP User Agents by adopting a tool for the analysis of communicating state machines. Thirdly, we present a web application prototype that implements the REST APIs according to the proposed specifications. The prototype offers a RESTful real-time communication service accessible to web browsers that supports the interworking with SIP User Agents. We also discuss three alternative implementations for handling asynchronous notification to web clients: the first based on the WebSocket protocol [Fette and Melnikov, 2011], the second on Long Polling technique and the third on HTTP Streaming technique [Huang and Zhu, 2012], compared upon experimental results.

In Chapter 1 we present the main technologies that we used in this thesis work (REST architectural style, SIP protocol and WebRTC standard) and discuss related works for the convergence of web and communication services.

In chapter 2 we describe our approach to design RESTful call service based on the adoption of a state machine formalism and a tool for the simulation of the service expected behavior and interworking with SIP-based systems.

In Chapter 3 we describe the implementation of our RESTful call service: in particular we show the technologies chose for this project, and we describe the packages, classes and client-side scripts developed to implement this service.

In Chapter 4 we describes the call service web application functioning and related functional test.

Finally, in Chapter 5 we describe the tests carried out to analyze the performance of the RESTful service and obtained results.

Part I

State of the Art

Chapter 1

Context of the work

The evolution towards the Next Generation Network (NGN) based on All-IP architecture aims to achieve the convergence of fixed and mobile communications networks, voice and video services, Web services and Internet. IP Multimedia Subsystem (IMS) is an architectural model for telecommunications networks (Figure 1.1) designed with the intent to bring together all fixed and mobile telecommunication devices on IP-based network infrastructure that is capable of providing voice and multimedia services.

Within the NGN, the *convergence* term can be used to have different meanings (e.g., access networks, terminal and service convergence). In this thesis we focus on the service convergence, as a set of features that allows mash-up, service composition and brokerage, and constituent service components between heterogeneous domains (Telecom and Web service providers). At present the most existing solutions for the Telco and Web service composition use inflexible instruments which do not allow the creation of convergent services in easy and fast way. The presence of APIs that allow easy integration between the Web and Telecom services lead to the rapid creation of new service types that offer to users an experience of major use. In a web page we can communicate with other users that surf it. The exposure of services through Web API is therefore considered a key factor to allow cooperation between network operators, service and content providers. This chapter in-

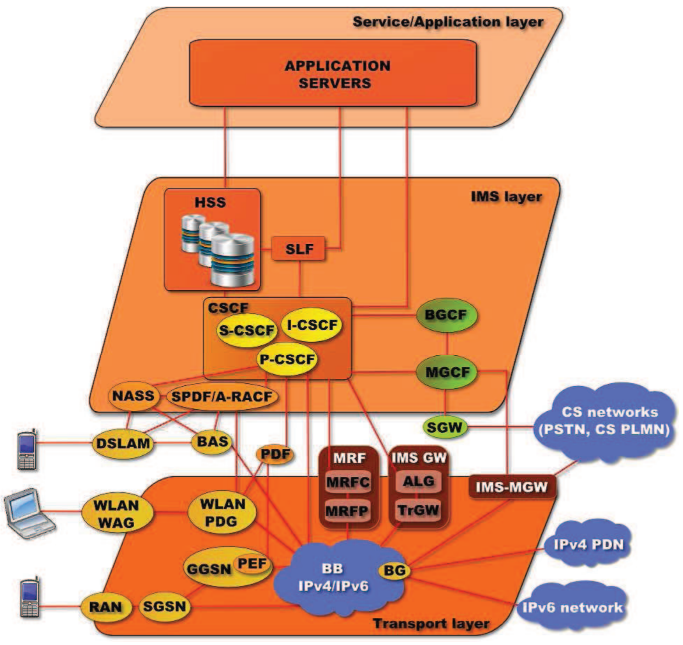


Figure 1.1: Ip Multimedia Subsystem Architecture.

troduces some technologies that can provide a simple and immediate way to create convergent services. In particular we focus on the design and development of Web-based APIs based on REST principles to expose communication services on the web and make them accessible via web browser. [Mazzi, 2013]

In section 1.1 we present REST architectural style to model and expose Web API, since it provides guidelines for developing applications coherent with the principles on the web. One of the most popular protocols for handling calls in the VoIP world is the Session Initiation Protocol (SIP), which allows the management of multimedia sessions at application level. This protocol is widely used in VoIP for the spread of broadband connections, so can count on an increasing number of users. For this reason, in section 1.2 we present it in more detail, describing the functioning and main characteristics. About real-time and web communications, in section 1.3 we describe the latest technology WebRTC whose objective is precisely to enable the connection and direct communication between two users through the use of browser. Finally, in section 1.4 we taken into account some research works that have proposed approach models for the Web and Telecom service convergence, such as exposure of telephony services via web API. For this purpose, the analyzed models use technologies seen in the development of convergent services. So doing, we can contextualize the work done within the landscape seeing to converge the internet and telecommunication world.

1.1 REST

The Representational State Transfer (REST) is an architectural style that provides a set of principles to create a web service-oriented client/server architecture. Fielding in his doctoral thesis writes: ”REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.” [Fielding, 2000]. The motivation behind the development of REST was to create a design pattern for how the Web should work, so that it could act as a guiding framework for the Web standards and designing Web services. On the REST vision, data sets and objects handled by client-server application logic are modeled as resources. Although REST is not bound to any specific protocol, in practice HTTP is widely adopted for its implementation. REST itself is not a standard but it prescribes the use of standards such as HTTP, URL, and XML/HTML/JPEG.

REST-style architectures consist of clients and servers. Clients initiate requests to servers who process these requests and return responses based on these requests. These requests and responses are built around the transfer of representations of these resources. A resource can be any coherent and meaningful concept that can be addressed, while a representation of a resource is a document that captures the intended state of a resource. Fundamentally in REST, each resource is first identified by using an URL and a new resource for every required service is created. The data returned by the service must be linked to the other data, hence making it in to a network of information unlike the Object Oriented design which encourages the encapsulation of information.

1.1.1 Principles

REST architectural style describes six constraints applied to architecture [Fielding, 2000]:

- *Client–server*, a uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

- *Stateless*, the client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition [Davids et al., 2011].

- *Cacheable* as on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

- *Layered system*, a client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.

- *Code on demand* (optional), servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript. ”Code on demand” is the only optional constraint of the REST architecture.

- *Uniform interface*, the uniform interface constraint is fundamental to the design of any REST service [Islam and Gregoire, 2013]. The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

– *Identification of resources (Addressability).* Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation, and it is the same one resource regardless. Resources are exposed by servers through URIs. Since URIs belong to a global addressing space, resources identified with URIs have a global scope.

– *Manipulation of resources through these representations.* When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource. The interaction with the resource is fully expressed with four primitives, i.e., create, read, update and delete. The constraint of uniform interface means that resources are handled through a fixed set of operations: create, read, update, delete. These operations can be mapped onto HTTP methods: GET gets the resource state; PUT sets the resource state; DELETE deletes a resource; POST extends a resource by creating a child resource.

– *Self-descriptive messages.* Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cacheability [Fielding, 2000].

– *Hypermedia as the engine of application state (HATEOAS).* Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext). Except for simple fixed entry points to the application, a client does not assume that any particular action is available for any particular resources beyond those described in representations previously received from the server.

One can characterize applications conforming to the REST constraints described in this section as ”RESTful” [Alvestrand, 2013]. If a service violates any of the required constraints, it cannot be considered RESTful.

Complying with these constraints, and thus conforming to the REST architectural style, enables any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

1.1.2 REST-oriented methodology

In this section we discuss how we applied a REST-oriented methodology to design a web API for communication services (e.g., a call service). We adopted the methodology for resource-oriented design proposed by Richardson and Ruby [2007]. According to this methodology, designers have to first figure out the dataset on which the service will operate, and split it into resources. After that, they should proceed for each resource as follows:

1. name the resource using a URI;
2. identify a subset of the uniform interface that is exposed by the resource;
3. design the representation(s) of the resource as received in a request from the client or returned in a reply;
4. analyze the typical course of events by exploring and defining how the new resource behaves during a successful execution and analyze possible error conditions.

This resource-oriented methodology uses the HTTP protocol and obviously GET, POST, PUT and DELETE methods like uniform interface. The request PUT is used to update the resource status. The PUT method, as well as DELETE, should be idempotent. The DELETE method tells the server that the resource should no longer exist. The client does not send a representation along with the request as unnecessary. The POST method is the attempt to create a new resource by an existing one. As for the PUT method, also in this case a representation of the resource is sent together with the request. The GET method allows to request a resource, so any representation is sent along with the request. This request type must not change the resource state. This resource-oriented methodology suggests an intelligent use about the response (positive and negative) appealing to the response states already defined by HTTP. For instance, a POST request for the resource creation causes a 201 ”Created” response in positive case, while a PUT or DELETE request causes a 204 ”No Content” response.

1.1.3 Asynchronicity Management

The REST style, in its conception, was heavily influenced by the HTTP protocol with which it is often implemented. The HTTP client/server nature has so conditioned the REST style that is designed with the idea of two entities: one that requires the service and that it provides. This can create problems in delivering services such NOTIFY/SUBSCRIBTION in which the change of a resource must be notified by the provider to the user. Fielding is not talking about this possibility and there is not generally a well-defined approach on how to address the problem. However some solutions exist, based on HTTP and other protocols closely linked, and are reported below:

1. GET or Periodic polling;
2. Long polling;
3. HTTP streaming;
4. WebSocket.

These solutions are treated in details in the next chapter.

1.2 SIP

The Session Initiation Protocol (SIP) is an application layer protocol used to create, modify and terminate multimedia sessions between two or more users. The first Request for Comments (RFC) was released in 1999 [Handley and Rosenberg, 1999], while the second version in 2002, RFC 3261 [Rosenberg et al., 2002a]. At the moment it is the most important because it contains the main specifications of this protocol. Standardization is done by IETF.

SIP is not the only protocol that the communicating devices will need. It is not meant to be a general purpose protocol. Purpose of SIP is just to make the communication possible, the communication itself must be achieved by another means (and possibly another protocols). Two protocols that are most often used along with SIP are RTP [Schulzrinne and Jacobson, 2003] and SDP [Handley and Jacobson, 1998]. RTP protocol is used to carry the real-time multimedia data (including audio, video, and text), the protocol makes it possible to encode and split the data into packets and transport such packets over the Internet. An another important protocol is SDP, which is used to describe and encode capabilities of the session participants. Such a description is then used to negotiate the characteristics of the session so that all the devices can participate (that includes, for example, negotiation of codecs used to encode media so all the participants will be able to decode it, negotiation of transport protocol used and so on).

SIP is independent from the underlying transport protocol: TCP, UDP, or otherwise. It is basically peer-to-peer and has intelligent endpoint and a network core which deals simple tasks. SIP presents an architectural model similar to HTTP:

- client/server architecture;
- request/response model;
- BNF textual encoding;
- codes associated with response messages.

SIP can be used to manage different service types:

- Short messaging (sms);
- IP Multimedia Messaging (MMS);
- Instant Messaging (IM);
- Terminal location;
- Presence;
- Audio call;
- Multimedia (e.g., video and audio) Conference;
- Streaming media;
- Third part call.

1.2.1 SIP URI

SIP entities are identified using SIP URI (Uniform Resource Identifier). A SIP URI has form of sip:username@domain, for instance, sip:joe@company.com. As we can see, SIP URI consists of username part and domain name part delimited by @ (at) character. SIP URIs are similar to e-mail addresses, it is, for instance, possible to use the same URI for e-mail and SIP communication, such URIs are easy to remember. [Janak, 2003]

1.2.2 SIP Network Elements

Basic SIP elements are user agents, proxies, registrars, and redirect servers. We will briefly describe them in this section. Note that the elements, as presented in this section, are often only logical entities. It is often profitable to co-locate them together, for instance, to increase the speed of processing, but that depends on a particular implementation and configuration. Figura 1.2 shows a functioning example of SIP protocol.

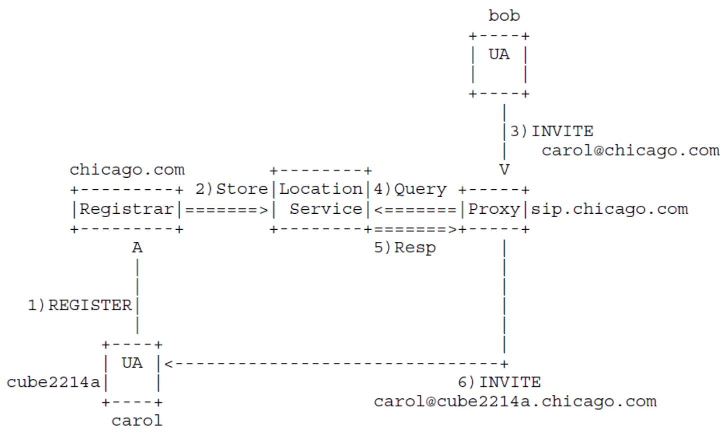


Figure 1.2: A functioning example of SIP protocol [Rosenberg et al., 2002a].

User Agent

Internet end points that use SIP to find each other and to negotiate a session characteristics are called user agents. User agents usually, but not necessarily, reside on a user's computer in form of an application; This is currently the most widely used approach, but user agents can be also cellular phones, PSTN gateways, PDAs, automated IVR systems and so on. User agents are often reffered to as User Agent Server (UAS) and User Agent Client (UAC). UAS and UAC are logical entities only, each user agent contains a UAC and UAS. UAC is the part of the user agent that sends requests and receives responses. UAS is the part of the user agent that receives requests and sends responses. Because a user agent contains both UAC and UAS, we often say that a user agent behaves like a UAC or UAS. For instance, caller's user agent behaves like UAC when it sends an INVITE requests and receives responses to the request. Callee's user agent behaves like a UAS when it receives the INVITE and sends responses. But this situation changes when the callee decides to send a BYE and terminate the session. In this case the callee's user agent (sending BYE) behaves like UAC and the caller's user agent behaves like UAS. [Janak, 2003]

Proxy server

User agents can send messages to a proxy server. Proxy servers are very important entities in the SIP infrastructure. They perform routing of a session invitations according to invitee's current location, authentication, accounting and many other important functions. The most important task of a proxy server is to route session invitations "closer" to callee. The session invitation will usually traverse a set of proxies until it finds one which knows the actual location of the callee. Such a proxy will forward the session invitation directly to the callee and the callee will then accept or decline the session invitation. There are two basic types of SIP proxy servers: stateless and stateful.

Stateless server are simple message forwarders. They forward messages independently of each other. Stateless proxies are simple, but faster than stateful proxy servers. They can be used as simple load balancers, message translators and routers.

Stateful proxies are more complex. Upon reception of a request, stateful proxies create a state and keep the state until the transaction finishes. Some transactions, especially those created by INVITE, can last quite long (until callee picks up or declines the call). Because stateful proxies must maintain the state for the duration of the transactions, their performance is limited. [Janak, 2003]

Registrar

A SIP entity that receives requests for registering and places information in the location database: IP address, port number, username and more. It is a logic element that is often placed in the same machine on which a proxy resides. Sometimes it find in dedicated machines in order to promote the network scalability.

Location Service

The Location Service typically resides on the same machine of a Registrar Server and contains a constantly updated database about the user records. It can directly locate the researched user or return the addresses of Proxy Server or other entities that may know the location. Sometimes an intermediate entity, called Redirect Server, between the Location Service and Proxy can be present, which is contacted by users as alternative to a Proxy. The Redirect Server obtains location information from the Location Database of a Registrar Server and communicates it to the user who can then re-route the request. In particular it is a UAS that generates responses 3xx (Redirection) to the requests it receives, directing the client to contact a set of alternative URI. These servers allow the proxy to direct calls to the SIP sessions on external domains.

1.2.3 Messages

Communication using SIP (often called signaling) comprises of series of messages. Messages can be transported independently by the network. Usually they are transported in a separate UDP datagram each. The model used by SIP is similar to HTTP that uses request and response. A fundamental difference is the ability to receive multiple SIP responses to a single request. In particular, a request may be associated with zero or more provisional responses (1xx) and one or more final answers [Janak, 2003]. SIP is a text-based request/response protocol. The messages have this format:

- Start-line (Request-line/Status-line), the first line identifies message type. There are two types of messages: requests and responses. If the first line of the message contains a request type, then it is a message request, otherwise if contains a response status, it is a response message. Both message types have this format:

- Message-header contains the headers of the message.
- CRLF, empty line.
- Message-body, optional field that can contain other informations.

In case of a request the first line expresses the type of request that the UA client wants to do. The SIP request messages are listed and briefly explained below:

- REGISTER, registration request to a Registrar.
- INVITE request for establishment of a session.
- ACK, confirmation of message exchange.
- CANCEL, termination request of a pending request.
- BYE, closing of a session between two users.
- OPTIONS, information request about the capabilities of the caller.

The message responses are sent by the UAS. In this case the Start-line contains a code representing the response. These codes are divided into six categories:

- Provisional (1xx), the request was received.
- Success (2xx), the request was received, accepted and processed.
- Redirection (3xx), the request needs other actions because it is satisfied.
- Client Error (4xx), the request can not be satisfied.
- Server Error (5xx), the server has failed request processing, even if valid.
- Global Failure (6xx), any server can meet the request.

The most important header fields are:

- From, URI of the sender.
- To, URI of the receiver.

- Call-ID, identifies a call between two or more participants.
- CSeq, identifies a transaction within the dialogue between two users.
- Via, identifies the protocol used for the transaction and the entities to which the response should be sent.
- Content-type, describes the content type in the message body.
- Content-length, indicates the content size in the message body.

1.2.4 Transactions

Although we have said that SIP messages are sent independently over the network, they are usually arranged into transactions by user agents and certain types of proxy servers. Therefore SIP is said to be a transactional protocol. A transaction is a sequence of SIP messages exchanged between SIP network elements. A transaction consists of one request and all responses to that request. That includes zero or more provisional responses and one or more final responses (remember that an INVITE might be answered by more than one final response when a proxy server forks the request). Figure 1.3 shows what messages belong to what transactions during a conversation of two user agents. [Janak, 2003]

1.2.5 Dialog

We have shown what transactions are, that one transaction includes INVITE and it's responses and another transaction includes BYE and it responses when a session is being torn down. But we feel that those two transactions should be somehow related—both of them belong to the same dialog. A dialog represents a peer-to-peer SIP relationship between two user agents. A dialog persists for some time and it is very important concept for user agents. Dialogs facilitate proper sequencing and routing of messages between user agents. Dialogs are identified using Call-ID, From tag, and To tag. Messages that have these three identifiers same belong to the same dialog. We have shown that CSeq header field is used to order messages, in fact it is used to order messages within a dialog. The number must be monotonically increased for each message sent within a dialog otherwise the peer will handle it as out of order request or retransmission. In fact, the CSeq number

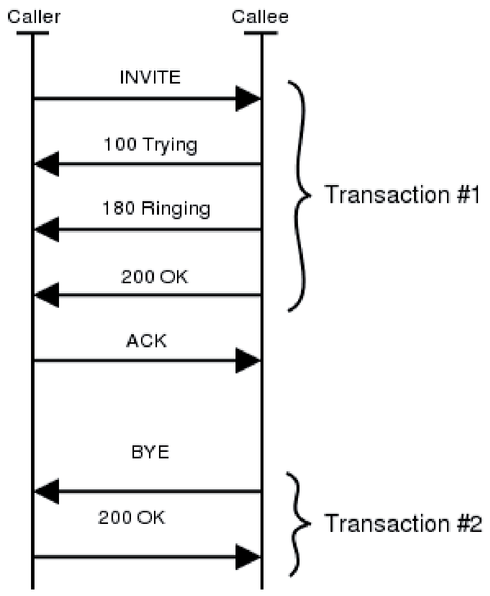


Figure 1.3: SIP Transactions

identifies a transaction within a dialog because we have said that requests and associated responses are called transaction. This means that only one transaction in each direction can be active within a dialog. One could also say that a dialog is a sequence of transactions. Figure 1.4 extends figure 1.3 to show which messages belong to the same dialog.

Some messages establish a dialog and some do not. For instance, INVITE message establishes a dialog, because it will be later followed by BYE request which will tear down the session established by the INVITE. This BYE is sent within the dialog established by the INVITE. But if a user agent sends a MESSAGE request, such a request doesn't establish any dialog. Any subsequent messages (even MESSAGE) will be sent independently of the previous one.

Call-ID is call identifier. It must be a unique string that identifies a call. A call consists of one or more dialogs. Multiple user agents may respond to a request when a proxy along the path forks the request. Each user agent that sends a 2xx establishes a separate dialog with the caller. All such dialogs are part of the same call and have the same Call-ID. From tag is generated by the caller and it uniquely identifies the dialog in the caller's user agent.

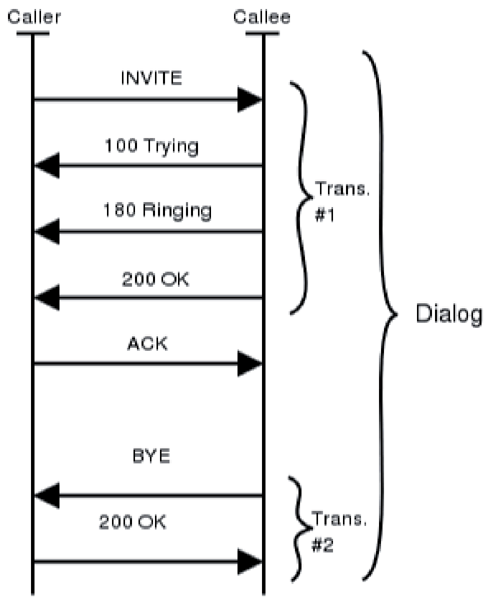


Figure 1.4: SIP Dialog

To tag is generated by a callee and it uniquely identifies, just like From tag, the dialog in the callee's user agent. This hierarchical dialog identifier is necessary because a single call invitation can create several dialogs and caller must be able to distinguish them. [Janak, 2003]

1.2.6 Typical SIP Scenarios

This section gives a brief overview of typical SIP scenarios that usually make up the SIP traffic.

Registration

Users must register themselves with a registrar to be reachable by other users. A registration comprises a REGISTER message followed by a 200 OK sent by registrar if the registration was successful. Registrations are usually authorized so a 407 reply can appear if the user didn't provide valid credentials. Figure 1.5 shows an example of registration and 1.6 its corresponding SIP REGISTER message.

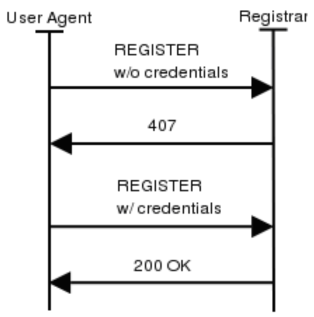


Figure 1.5: Example of SIP registration.

```
REGISTER sips:ss2.biloxi.example.com SIP/2.0
Via: SIP/2.0/TLS
client.biloxi.example.com:5061;branch=z9hG4bKnashds7
Max-Forwards: 70
From: Bob <sips:bob@biloxi.example.com>;tag=a73kszlfl
To: Bob <sips:bob@biloxi.example.com>
Call-ID: 1j9FpLxk3uxtm8tn@biloxi.example.com
CSeq: 1 REGISTER
Contact: <sips:bob@client.biloxi.example.com>
Content-Length: 0
```

Figure 1.6: SIP REGISTER Message.

Session Invitation

A session invitation consists of one INVITE request which is usually sent to a proxy. The proxy sends immediately a ”100 Trying” message reply to stop retransmissions and forwards the request further. All provisional responses generated by callee are sent back to the caller. The response is generated when callee’s phone starts ringing. Figure 1.9 shows a ”180 RINGING” message. A ”200 OK” message is generated once the callee picks up the phone and it is retransmitted by the callee’s user agent until it receives an ”ACK” message from the caller, as shown in Figure 1.10. The session is established at this point. Figure 1.7 shows an example of session invitation.

In order to establish a call two users have to exchange data about the protocols and encodings supported. Usually this is done through the use of Session Description Protocol (SDP) [Handley and Jacobson, 1998], which provides a standard representation for the description of the above information. A first description, called ”offer”, is typically sent in the body of the INVITE message and is generated by the AUC, as shown in Figure 1.8. This allows the called to create a response with a description of its ability, called ”answer”, which is typically sent in the body of the ”200 OK” message, as

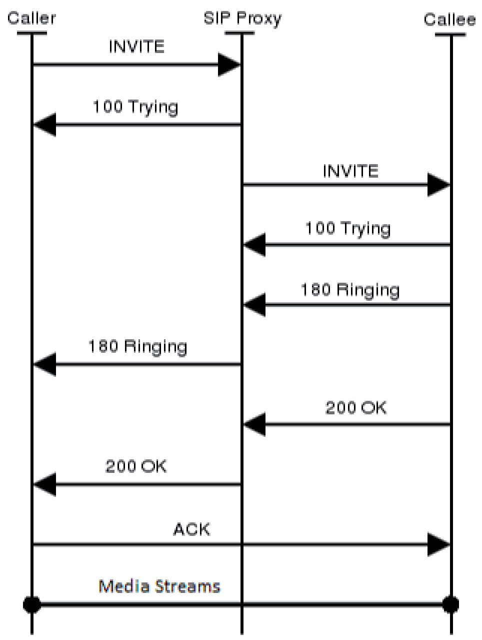


Figure 1.7: Example of SIP session invitation.

shown in Figure 1.11. In case of negative response, the sending of ”answer” is completely superfluous. Once both participants have ”offer” and ”answer”, a media channel can be established between the two UA for the communication between their.

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP
client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob sip:bob@biloxi.example.com
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:alice@client.atlanta.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 151

v=0
o=alice 2890844526 2890844526 IN IP4 client.atlanta.example.com
s=-
c=IN IP4 192.0.2.101
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Figure 1.8: SIP INVITE Message.

```
SIP/2.0 180 Ringing
Via: SIP/2.0/TCP
client.atlanta.example.com:5060;branch=z9hG4bK74bf9
;received=192.0.2.101
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:bob@client.biloxi.example.com;transport=tcp>
Content-Length: 0
```

Figure 1.9: SIP 180 RINGING Message.

```
ACK sip:bob@client.biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP
client.atlanta.example.com:5060;branch=z9hG4bK74bd5
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 ACK
Content-Length: 0
```

Figure 1.10: SIP ACK Message.

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
client.atlanta.example.com:5060;branch=z9hG4bK74bf9
;received=192.0.2.101
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:bob@client.biloxi.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 147

v=0
o=bob 2890844527 2890844527 IN IP4 client.biloxi.example.com
s=-
c=IN IP4 192.0.2.201
t=0 0
m=audio 3456 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Figure 1.11: SIP 200 OK Message.

Session Termination

Session termination is accomplished by sending a BYE request within dialog established bye INVITE. Party wishing to tear down a session sends a BYE request to the other party involved in the session. The other party sends a 200 OK response to confirm the BYE and the session is terminated. The transaction 2 of Figure 1.4 shows an example of session termination.

1.2.7 INVITE Client Transaction

The INVITE transaction consists of a three-way handshake, as shown in Figure 1.12. The client transaction sends an INVITE, the server transaction sends responses, and the client transaction sends an ACK. The server transaction can send additional 1xx responses, which are not transmitted reliably by the server transaction. Eventually, the server transaction decides to send a final response. For each final response that is received at the client transaction, the client transaction sends an ACK, the purpose of which is to quench retransmissions of the response. [Rosenberg et al., 2002a]

1.2.8 INVITE Server Transaction

The state diagram for the INVITE server transaction is shown in Figure 1.13. When a server transaction is constructed for a request, it enters the ”Proceeding” state. The server transaction must generate a 100 (Trying) response unless it knows that the TU will generate a provisional or final response within 200 ms, in which case it MAY generate a 100 (Trying) response. If, while in the ”Proceeding” state, the TU passes a 2xx response to the server transaction, the server transaction must pass this response to the transport layer for transmission. While in the ”Proceeding” state, if the TU passes a response with status code from 300 to 699 to the server transaction, the response MUST be passed to the transport layer for transmission, and the state machine must enter the ”Completed” state. [Rosenberg et al., 2002a]

We described only the INVITE Client and Server Transactions because these scenarios are very important in this thesis, while the Non-INVITE Client and Server Transactions are in RFC 3261, [Rosenberg et al., 2002a].

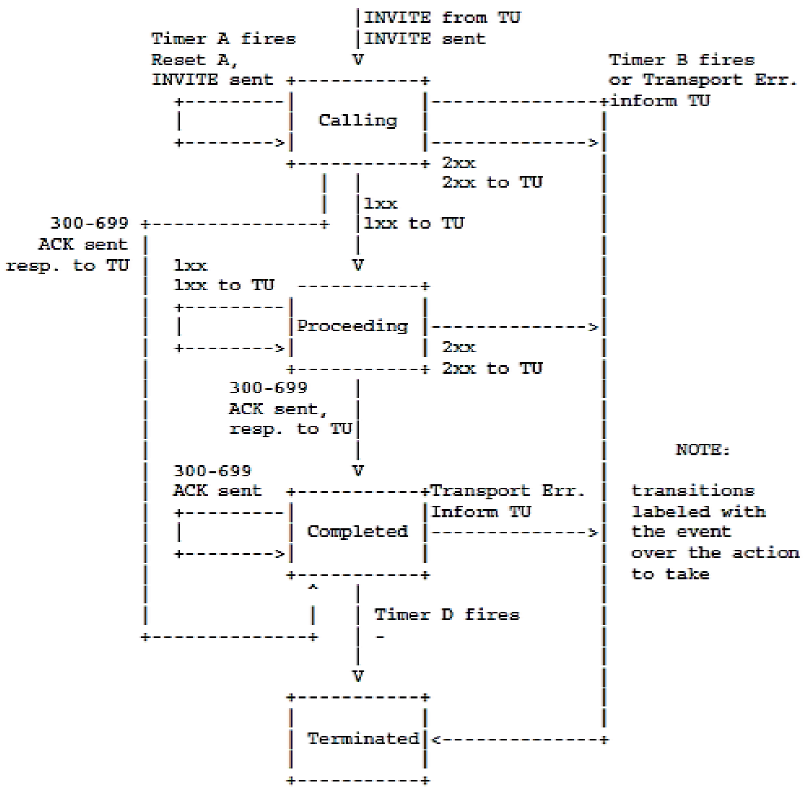


Figure 1.12: INVITE Client Transaction [Rosenberg et al., 2002a].

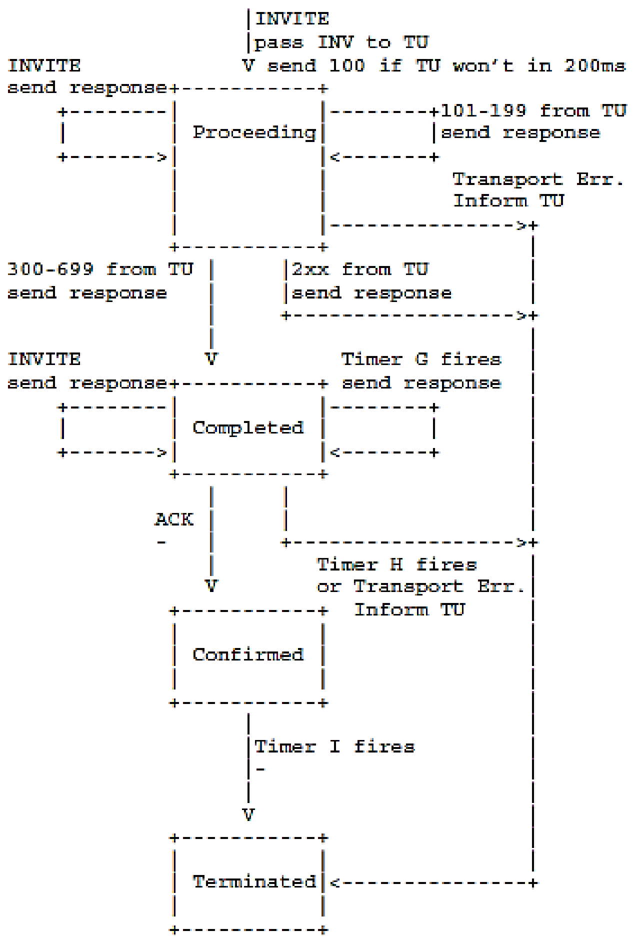


Figure 1.13: INVITE Server Transaction [Rosenberg et al., 2002a].

1.3 WebRTC

The last years have seen an increasing use of web applications to provide various service types with the ability to implement more complete interfaces. In some cases these solutions needed plugin that have to be downloaded and installed separately. HTML5 has as objective for the programmer the increase of available tools, which correspond to an increase of services that can be offered to the end user without the use of external plugins.

WebRTC is a free open source project born in 2011 and currently supported by Google, Mozilla and Opera. The objective of WebRTC is to enable the browser to realize audio/video conference and sharing files, using HTML5 and Javascript API, without the user has to install external plugins or make use of dedicated applications. APIs are currently still very young and not fully functional. The browsers that implement these features and can actually make audio/video browser-to-browser calls are Google Chrome and Mozilla Firefox [Bergkvist et al., 2014]. In order to achieve this objective, WebRTC uses multiple technologies defined by several standardization groups:

- A suite of protocols developed by the group RTCWEB of IETF for real-time communication between applications that can be run by a browser [Richardson and Ruby, 2007].
- API for JavaScript language defined by W3C [Fielding, 2000], making it possible to send and receive media data between two browsers or devices that implement the appropriate set of real-time protocols.
- API for accessing to local media devices developed by Media Capture Task Force.

The RTC capabilities located within the browser allow the communication through the suite of protocols defined by IETF and are exposed through the API defined by W3C.

1.3.1 IETF Protocol Specification

The process of communication establishment between two browsers can be explained by observing Figure 1.14. The two browsers communicate via two paths. The first path is used for the signaling phase and uses a web server to convey messages, whose content allows the creation of the second path dedicated to the direct communication between the browsers. The second path must comply with the specifications of the RTCWEB protocol suite. The process is similar to that seen in the SIP world: SIP User Agents in place of browsers, Proxy in place of Web Server, and the signaling occurs through the use of the SIP protocol and not HTTP or WebSocket. The specification [Richardson and Ruby, 2007] highlights as the media negotiation should use descriptions that follow the SDP syntax so that it is possible to build a gateway for the signaling between SIP and RTCWEB. This makes possible to communicate with future SIP devices that support ICE, RTP and SDP.

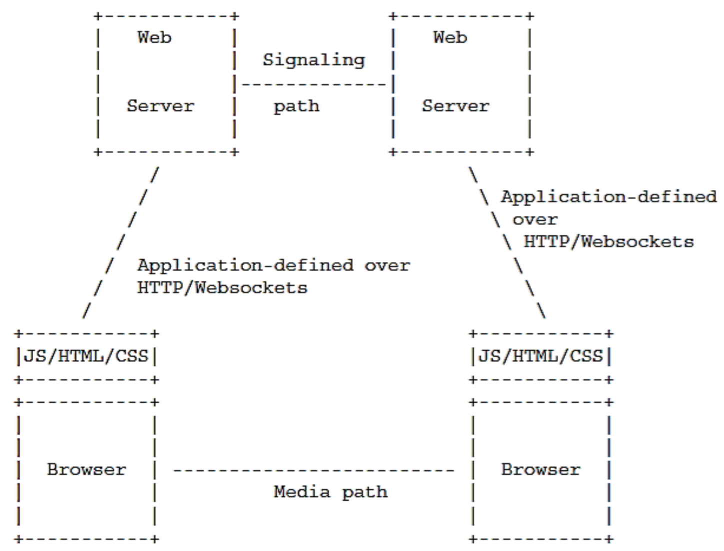

Web Server
Signaling path
Web Server
Application-defined over HTTP/Websockets
Application-defined over HTTP/Websockets
JS/HTML/CSS
JS/HTML/CSS
Browser
Media path
Browser

Figure 1.14: Communication establishment between two browsers.

1.3.2 W3C API JavaScript Specification

WebRTC is a recent technology and, for this reason, the JavaScript APIs are subject to continuous updates accompanied by related documentation. In this work we consider the last Working Draft of 4 July 2014 [Bergkvist et al., 2014] although different browsers can implement slightly different API. These differences do not affect the main concepts. APIs are divided into two parts:

- Network Stream APIs.
- Peer-to-peer connections.

The Network Stream APIs (called also getUserMedia) allow to access to multimedia resources of the computer. Within these APIS the MediaStream concept is defined, namely an interface that represents a data audio/video stream type. This interface can be extended to represent a stream from or sent to a remote node. Each MediaStream can be composed of more MediaStreamTrack as shown in Figure 1.15. A MediaStreamTrack represents the data flow coming from a device (e.g., webcam, microphone).

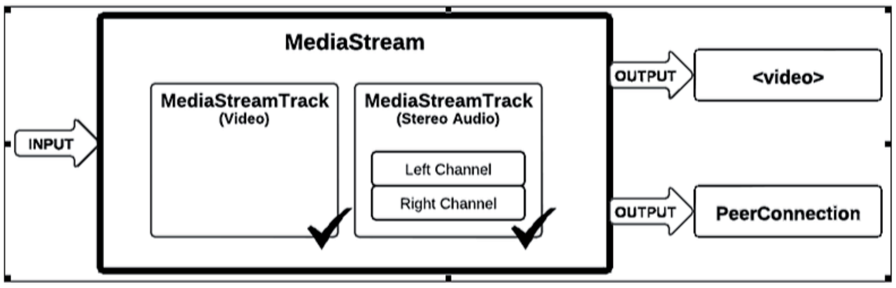


Figure 1.15: MediaStream and MediaStreamTrack.

The peer-to-peer connections concern the communication between two browsers. In particular, the RTCPeerConnection class allows two users to communicate directly from browser to browser. This communication is coordinated via a not-specified signaling method although one usually used is illustrated in Figure 1.14, which uses HTTP messages and a WebSocket chanel between Browser and Web Server. The creation of a RTCPeerConnection object provide to pass any parameters for crossing NAT via Session Traversal Utilities for NAT" (STUN) or Traversal Using Relays around server NAT (TURN) server. The most important fields of the RTCPeerConnection class are:

- iceState, indicates the status of ICE agent,

- readyState, indicates the RTC connection status,
- localDescription, contains the SDP of local media,
- remoteDescription, contains the SDP of remote media,
- localStreams, contains an array of local streams,
- remoteStreams, contains an array of remote stream.

Moreover, methods for object the management are defined. The create-Offer and createAnswer methods permit the creation of SDP to be sent to the caller or callee. For a discussion we refer to [Bergkvist et al., 2014].

1.4 Related Work

The increasing need of making capabilities of an operator's network accessible and invokable by applications of external consumers has driven the recent technological evolution in the telecommunication domain. To this purpose, the service-oriented principles [Erl, 2007] have inspired the Service Delivery Platforms specifications exposing the telecom capabilities via open APIs in order to enable enhanced and flexible service provision and composition.

In this context, several standard specifications regarding the exposure of telecom services have been recently specified. The ITU-T has defined the NGN Open Service Environment (OSE) that offers standard APIs to access and orchestrate heterogeneous Next Generation Network services to the application providers [ITU, 2008]. The Open Mobile Alliance has published specifications about an Open Service Environment and related Service Enablers [OMA, 2009]. These open specifications define how the functional capabilities have to designed, deployed, composed, and executed over convergent networks [Brenner and Unmehopa, 2008].

More recently, the IEEE Standard Association has approved the specifications regarding the Next Generation Service Overlay Network (NGSON) functional architecture [NGSON Working Group, 2011, Lee and Kang, 2012]. The IEEE NGSON architecture defines functions related to the service and transport in order to support context-aware, dynamically adaptive, and self-organizing networks. NGSON is expected to operate on the top of different underlying networks such as the IP Multimedia Subsystem (IMS), Next Generation Networks (NGN), peer-to-peer (P2P) overlays, and the Web.

Menkens and Wuertinger [2011] discuss the move in the Telecommunication industry towards the service-oriented infrastructures and the actions made by telecom service providers to make their capabilities accessible by third party developers. They also highlight major obstacles towards the development of Web/Telecom convergent applications:

1. available specifications for telecom service environments define how telecommunication features can be exposed to third party developers, but they do not provide any concept or paradigm for supporting the developers in the composition of telecommunication services with web services;

2. telecommunications specifications, such as IP Multimedia Subsystem (IMS) [Camarillo and García-Martín, 2006] and Session Initiation Protocol (SIP) [Rosenberg et al., 2002a], are not supported by default by widely adopted platforms for mobile devices;

3. application developers typically adopt Internet, web protocols and data formats (e.g., HTTP, XML [Bray et al., 2004] and the JavaScript Object Notation (JSON) [Crockford, 2002]).

More specifically, the convergence of Web and SIP-based services is considered difficult to achieve, since HTTP and SIP protocols rely on different principles [Bond et al., 2009, Islam and Gregoire, 2013]:

- typical use of SIP is stateful, while HTTP is stateless;

- SIP is peer-oriented, while HTTP is based on the client-server paradigm.

1.4.1 Web APIs for Telecom services

In order to effectively support third party application developers, some standard specifications for the telecom service exposure based on Web have been defined [Mulligan, 2009, Belqasmi et al., 2011]. Web-based interfaces may be distinguished into those that comply with Web Service (WS) specifications and those that comply with REST guidelines.

The Open Mobile Alliance has defined a web service framework called OMA Web Services Enabler [OMA, 2006]. The Parlay group, which is a standardization body that works in collaboration with OMA, Third Generation Partnership Program (3GPP) and European Telecommunications Standards Institute (ETSI), has defined the Parlay X specifications [3GPP, 2009]. Parlay X is a set of Web Service APIs for accessing a wide range of telecom network capabilities (e.g., third party call control, call notification, short messaging, and payment). Nonetheless, Mulligan [2009] argued that these APIs present some limitations as they do not allow the developer to handle the service data model, although handle reasonably well the session establishment.

More recently, several standardization efforts have been focused on RESTful APIs specifications for making the telecommunication services more easily accessible by third-party web-application developers [Belqasmi et al., 2011]. The Open Mobile Alliance (OMA) has released the specifications regarding the RESTful bindings for Parlay X Web Services in 2012 [OMA, 2012]. The currently available version (version 2.0) includes simple no-session services such as short and multimedia messaging, payment and location services, and accessory features for call services.

Group Special Mobile Association (GSMA) has published OneAPI [GSMA, 2009]. It provides REST APIs enabling applications to exploit mobile network capabilities, i.e. call control, messaging, authentication, payments and location-finding across multi-operator domains.

The IETF Centralized Conferencing Manipulation Protocol (CCMP) specification [Barnes et al., 2012] includes a possible mapping between CCMP and REST architectural style. These REST APIs can be used for manipulating XML documents that contain the information characterizing a specified conference instance.

Several research works have investigated the web service adoption for exposing telecom capabilities [Chou et al., 2008, Griffin and Pesch, 2007]. Recently, researchers have increasingly focused their efforts on RESTful services, rather than on WS ones, since RESTful services are deemed more lightweight and close to web-application programming models. Belqasmi et al. [2012] made a comparison between WS and RESTful multimedia conference services and concluded that RESTful services showed better performance. Similar results have been found by AlShahwan and Moessner [2010].

Fu et al. [2010] presented an early feasibility prototype for a REST-based service architecture in order to bridge the presence service across heterogeneous domains. Moriya and Akahani [2010] conducted an experiment with human participants for investigating the productivity of web-telecom applications with Parlay X and with a software development kit (SDK) that they developed in order to easy the use of Parlay APIs. They found two major problems:

1. the programmers may not know the call session state since the SOAP/HTTP interface makes the stateless and synchronous interaction;

2. in analogous way, the programmers may apply a procedural style, while they disregard the event-driven (i.e., asynchronous) nature of telecommunication services.

The handling of session-based capabilities (e.g., a call between two end users) is discussed in several works. Lozano et al. [2008] proposed a set of REST APIs for exposing session-based IMS capabilities where the asynchronous notification is handled through HTTP polling technique. Davids et al. [2011] discussed different options in order to allow voice and video communications on the Web. They proposed a RESTful API over HTTP, where the asynchronous notification is realized through long-lived HTTP technique. Nicolas et al. [2011] proposed an approach for the convergence of telecom and web services that exploited the WebSocket protocol. However, the design of REST APIs was not discussed in detail and the message flow was described only for presence and location services. Griffin and Flanagan [2011a] applied a resource-oriented design methodology for defining a call control interface that can be consumed by browser-based applications. They toke as reference a simple call model adapted from the Computer Supported Telecommunications (CSTA) industry standard. The authors also addressed the problem of asynchronous events delivery to web browsers in another work [Griffin and Flanagan, 2010].

1.4.2 Web applications for real-time Communication

Finally, we mention the ongoing standardization efforts by the IETF and the W3C in order to enable direct and interactive communication between browsers. The IETF and W3C are defining respectively the RTCWeb protocol [Alvestrand, 2013] and WebRTC APIs [Bergkvist et al., 2014] to set up a media channel between web browsers, while the choice of a signaling mechanism is left to the application developers. An open issue is the interworking between legacy systems, especially SIP-based architectures, and the new up-coming solutions compliant with the new standards [Amirante et al., 2013]. Li and Zhang [2012] discussed the need of integrating a WebRTC-based solution with IP Multimedia Subsystem for providing a preliminary description of an integration solution, while Amirante et al. [2013] discussed the main technical issues entailed by the integration of SIP-based solutions with WebRTC applications for proposing a working solution for a conferencing system.

Part II

Discussion of the work

Chapter 2

RESTful Service Design

This chapter describes the steps taken to design the RESTful service. In paragraph 2.1 we motivate our contribution. In paragraph 2.2 we describe the main reference scenarios for the user who wants to use this service. In paragraph 2.3 we describe our approach to the design of web-based API for real time communications service. In paragraphs 2.4 and 2.5 we describe REST resources constituting the service, in particular in paragraph 2.4 we present the *presence* resource and in paragraph 2.5 the *call* resource. In order to design REST APIs for the *call* resource we adopt a state machine formalism for modeling the call service behavior and introduce a tool for simulating the service expected behavior and interworking with SIP-based systems. In paragraph 2.6 we describe some solutions for the asynchronicity problem using various technologies.

2.1 Motivation of our work

As argued by Belqasmi et al. [2011], the adoption of a stateless architectural style for the exposure of session-based services requires special attention. Our work basically accounts for the results achieved by Li and Chou [2010], Davids et al. [2011] and Griffin and Flanagan [2011a] who discussed benefits and issues of applying REST principles to the design of web-based real time communication services.

Similarly to the above mentioned works, but with a novel approach, in this thesis we propose a set of REST APIs for communication services (e.g., a voice call service) designed by the adoption of a resource-oriented service design methodology.

The main limitation of the above-mentioned works, as well as of most works in the REST-oriented service design, is the lack of rigorous modeling of REST principles and related RESTful systems. As argued by Zuzak et al. [2011] this fact is causing *"negative effects, such as confusion in understanding REST concepts, misuse of terminology and ignorance of benefits of the REST style"*. In order to overcome this limitation, the original contribution of this work is threefold:

1. we model the *call* resource behavior through a Finite State Machine representation which accounts for the SIP specifications of a call session setup and for REST constraints;

2. we adopt a tool for the analysis of communicating state machines in order to simulate the behavior of the service and its interworking with SIP User Agents;

3. we discuss the implementation of a web application prototype that exposes these REST APIs and we evaluate its compliance with the specifications with the help of the communicating state machines analysis tool in some significant sample scenarios. The prototype supports three mechanisms for the delivery of asynchronous notifications to web browsers (the first based on WebSocket, the second on Long Polling and the third on HTTP Streaming).

Our REST-based design and implementation approach is also compliant with the Hypermedia as The Engine Of Application State (HATEOAS) constraint. Although this is one of the main REST constraints, it is often disregarded [Liskin et al., 2011].

The conceptual model of the proposed call service is depicted in Fig. 2.1 and Fig. 2.2.

Fig. 2.1 shows a call setup between two REST clients (e.g., web browsers). This model presents two REST clients and an intermediary component, called REST call service, that exposes Web APIs for the exchange of the signaling messages required for the call session setup. The media path does not necessarily require an intermediary component, unless additional processing is required (e.g., transcoding).

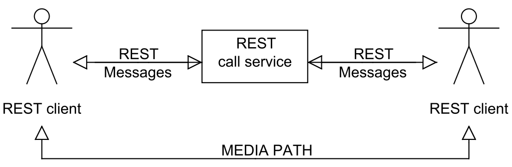


Figure 2.1: Call setup between two REST clients (web browsers).

Fig. 2.2 shows a call setup between a REST client and a SIP User Agent. This model presents a REST client, a SIP User Agent and an intermediary component that is required for the management of the signaling flow and the translation between the REST call service and a SIP Proxy. Similarly to the case mentioned above, the media path does necessarily require an intermediary component. However, current implementations of WebRTC specifications may require a media gateway to interwork with SIP User Agents [Amirante et al., 2013].

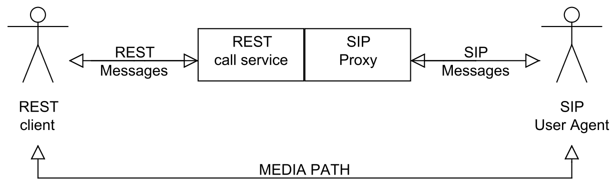


Figure 2.2: Call setup between a REST client and SIP User Agent.

2.2 Reference scenarios

Before moving on to the service design we define the use cases that the service has to manage and some simplifying assumptions:

1. all the users are registered to the same domain,
2. the system provides the SIP Proxy and Register functions for that domain.

Below, we took into account the following reference scenarios:

1. registration and deregistration service of a REST client,
2. registration and deregistration service of a SIP client,
3. call service between two REST clients,
4. call service from a REST client towards a SIP User Agent.
5. call service from a SIP User Agent towards a REST client.

2.2.1 Registration and Deregistration of a REST client

The registration of a user to the server allows to create an presence service. With this service, the users can know the users online connected via REST client and optionally also SIP client. This information permits to check whether the called user is currently online and so accessible. This action is always accompanied by a subscription to a incoming call (Figure 2.3). By subscription, a user registers to a service that will send call requests notifications. In symmetrical way, a user can deregister, be offline and unsubscribe by the notification service for incoming calls (Figure 2.4).

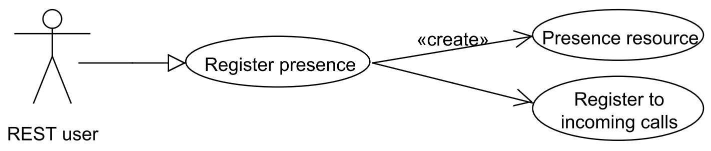


Figure 2.3: Registration to the service of a REST client

2.2.2 Registration and Deregistration of a SIP User Agent

Similarly to the previous case, also the users connected via SIP User Agent can register to the presence service. Obviously in this case the registration and deregistration are made by using the SIP protocol, in particular through the REGISTER message (Figure 2.5 and Figure 2.6). Once SIP users are

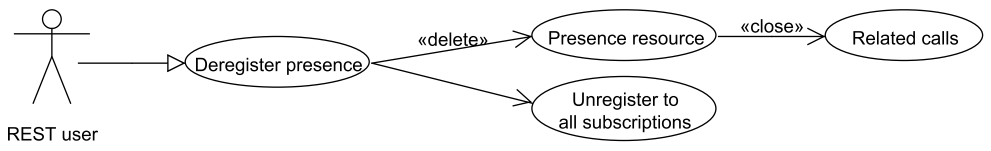


Figure 2.4: Deregistration to the service of a REST client

registered, they can be displayed in the list of users online. In contrast to the users connected via REST client, the registration to the notification service is not necessary. In fact, a SIP User Agent can operate both as client (requires the establishment of a call) and as server (receives an invitation for a call), once it knowns IP address and the port on which it listens.

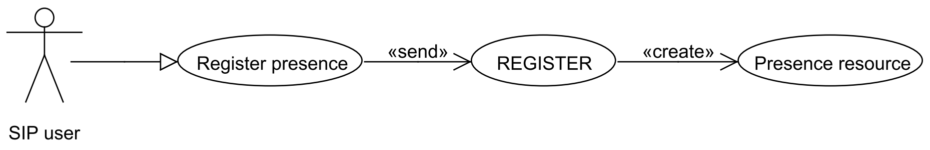


Figure 2.5: Registration to the presence service of a SIP client

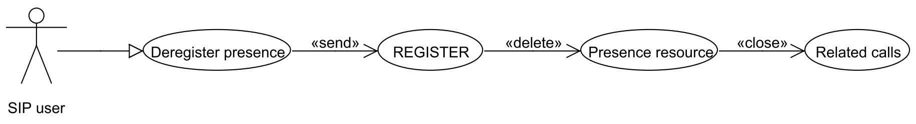


Figure 2.6: Deregistration to the presence service of a SIP client

2.2.3 Call between two REST clients

This is the first of three cases related to the service call. First, we analyze the case where a user accessing to the service via REST client (e.g., web browser) requires the establishment of a call with another user connected via REST client. In this case, the service doesn't handle the interoperability with systems based on the SIP protocol. Both users must have already signed to the presence and notification service. As shown in Figure 2.7, the presence of the called user is also occurred at the creation of the call. Moreover, the caller user must record to the call just made (Register to Call). In this way the REST client will be notified of any update on the call and will eventually perform actions on the occurrence of a new state.

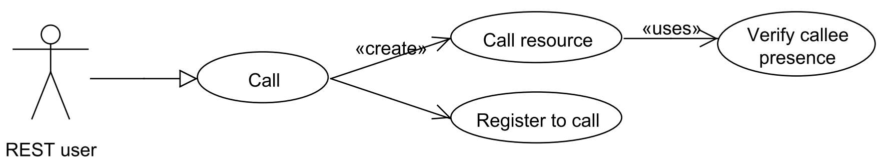


Figure 2.7: Call service between two REST users

2.2.4 REST user to SIP User Agent Call service

In this scenario an user using a REST client wants to establish a call with an user using a SIP client. As mentioned above, it is necessary that the REST client is registered while it is optional for the SIP user. The creation of the SIP interface that sends the INVITE message follows the SIP protocol specifications, as shown in Figure 2.8.

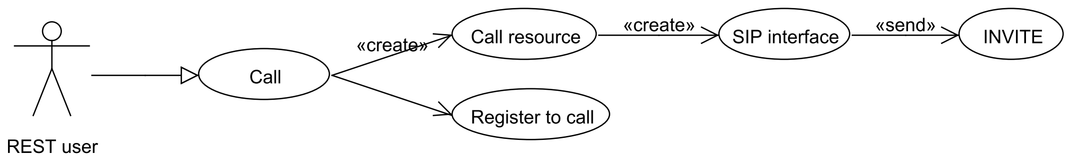


Figure 2.8: Call service from a REST user towards a SIP User Agent

2.2.5 SIP User Agent to REST user Call service

In this last scenario an user using a SIP User Agent wants to establish a call with an user connected via a REST client. In this case the SIP User Agent sends messages to the SIP interface of the call service. At this point the service checks if the callee user is a REST user recorded and available to the call (i.e., online status) and then continues by sending to the REST client a notification message about the incoming call (Figure 2.9).

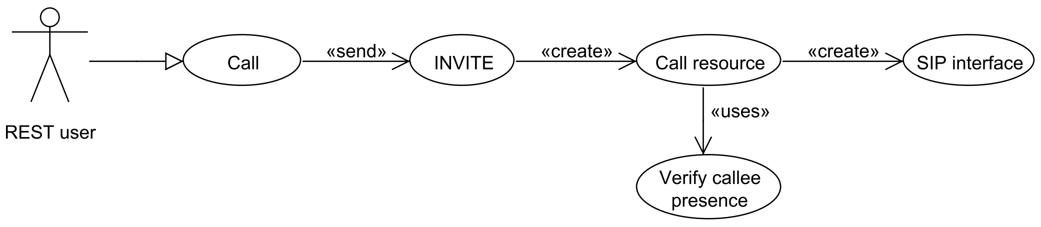


Figure 2.9: Call service from a SIP User Agent towards a REST user

2.3 Resource-oriented Design

As mentioned in Section 2.1, the aim of this thesis is the specification of a APIs set for a communication service based on REST principles. In previous chapter we described the REST principles, and after the service design that follows the Resource Oriented Architecture (ROA) specification. The advantages already described are also reflected in this service: REST constraints (unique and addressable resources, uniform interface, absence of state in the communication between client and server) give the possibility to choose freely architecture to use and to scale to system level. We adopted the methodology for resource-oriented design proposed by Richardson and Ruby [2007]. According to this methodology, designers have to first figure out the dataset on which the service will operate, and split it into resources. After that, they should proceed for each resource as follows:

1. naming the resource using a URI;
2. identifying a subset of the uniform interface that is exposed by the resource;
3. designing the representation(s) of the resource as received in a request from the client or returned in a reply;

4. analyzing the typical course of events by exploring and defining how the new resource behaves during a successful execution and analyze possible error conditions.

The service domain consists of a list of capabilities that are made available to the web browser through RESTful web services. To represent the service domain of the above-mentioned reference scenarios, we then identified the main resources of the REST-based service communication:

1. Presence, represents a user's availability status and contact information. This resource permits to know the registered users that access to the offered services.

2. Call, represents a video or audio call between two peers. This resource contains all the information that describes the call in terms of signaling and media traffic and call state.

In the following sections we describe in detail the design of two resources by ROA style.

2.4 Presence resourse

The *presence* resource is responsible for storing and distributing presence information of the connected users. For the SIP protocol extensions exist, such as SIMPLE [Rosenberg et al., 2002b], which allow to implement this service type. In this thesis we consider a simplifying configuration assumptions where the main purpose of the Presence service is to determine if a user can be reached by call service or not. In the next sub-paragraphs the service is defined through the steps outlined in the architecture ROA.

2.4.1 Assigning names to resources

Each resource is identified through a URI. According to the REST guidelines, URI fragments should contain nouns (e.g., presence), rather than verbs (e.g., registerpresence). In this work the *presences* resource is identified through the `http://{servername}/presences` URI; analogously, the identifier of a *presence* resource is `http://{servername}/presences/{presence_id}`.

2.4.2 Uniform interface

The constraint of uniform interface means that resources are handled through a fixed set of operations: create, read, update, delete (CRUD). These operations can be mapped onto HTTP methods: GET gets the resource state; PUT sets the resource state; DELETE deletes a resource; POST extends a resource by creating a child resource. This section describes the operations associated to the *presence* resource. These operations are summarized in Table 2.1. The first column shows the resource URIs, the second defines the HTTP methods that must be invoked to perform the specific operation. These actions are identified by an URI and a HTTP method. The third and fourth column indicate if the message body of HTTP request and response is empty, respectively. The fifth column provides a description of the HTTP method. The Table 2.1 shows the XML representation of the resource because it is the format that is actually used. The resource can be expose in other formats (e.g., HTML, JSON). Moreover, not all operations require to the client to send data to the server or vice versa. In particular in read operations the server sends data as response to the client while in creation and updating operations the client sends data to the server. Now we describe the individual operations on the *presences* and *presence* resources:

1. Creation of a *presence* resource, an user register to the registration service by creating a resource Presence. This operation is done by sending a POST request on the URI: `http://{servername}/presences`. The server receives in the client request all the information regarding the new resource and responds with a 201 ”Created” status code. In the response header there is a Location field that contains the URI of the new resource created.

2. Reading of all the *presence* resources, the first read operation is performed by using the GET method on the URI: `http://{servername}/presences`. This operation allows to get all the existing *presence* resources and then to know which users are currently online. If the operation is successful, the server sends 200 status code as response.

3. Reading of a specific *presence* resource, the second read operation requires the URI of a specific resource: `http://{servername}/presences/{presence_id}`. As in the previous case the positive response has 200 code.

4. Updating of a *presence* resource, the update operation is done by invoking the PUT method on the URI: `http://{servername}/presences/{presence_id}`. This operation permits to change the status of the *presence* resource. This request contains the new values about Presence in the message body. If the operation is successful, the server returns 204 ”No Content” code status in order to specify that the the resource has been overwritten with the new values, but the response doesn’t return any data.

5. Cancellation of a *presence* resource, the deletion operation of a single resource is done by the DELETE method on the URI: `http://{servername}/presences/{presence_id}`. This method doesn’t require the information sending by the client and doesn’t return any data. For this reason If the operation is successful, the server returns 204 ”No Content” code status as for the PUT method. We presented cases of correct behavior. The errors in the request message, or server-side processing, are also handled using HTTP status codes. For instance, when an user want to perform operations on the not-existing *presence* resource by using a URI is not assigned, the server returns 404 ”Not found” status code.

Table 2.1: REST APIS for *presence* resource management

Resource URIs	HTTP Method	Request Message Body	Response Message Body	Description
`/presences`	GET	No	Yes	Retrieve a list of presences
`/presences`	POST	Yes	Yes	Create a new *presence* resource
`/presences/{presence_id}`	PUT	Yes	Yes	Modify the *presence* resource state
`/presences/{presence_id}`	DELETE	No	No	Delete the *presence* resource
`/presences/{presence_id}`	GET	No	Yes	Retrieve the *presence* resource

2.4.3 Resource representation

According to Fielding [2000] *”REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components*”. At each interaction step, a representation may indicate the current state of the requested resource, the desired state for the requested resource, or the value of some other resources (e.g. a representation of some error conditions).

The *presence* resource representation contains the following data fields:

- *uri*, indicates the user SIP URI (e,g., sip:alice@server.it). There are not two *presence* resources with the same URI and therefore a check must be inspected at the time of registration.
- *status*, user status (i.e., online, busy, away), indicates the availability of the user.
- *SipUA*, boolean value explicits if the user is connected via SIP client (true) or via REST Client (false). This information is important because the presence of a SIP client in a call requires that the service sends SIP messages. To this end, the establishment of the call matches the instantiation of an object (SIP message sender, as we shall see in the following paragraphs) in order to manage the sending of SIP messages to the SIP client.

2.5 Call resource

The *call* resource is the main resource of our work and the design is more complex than the *presence* resource. Important insights for the definition of the *call* resource were taken from [Griffin and Flanagan, 2011b]. In detail we describe two resources:

- The *calls* resource represents the list of calls handled by the system, including the calls that have been disconnected but whose details are available in the call history. Maintaining the call details after disconnection may serve for providing end users with the history of calls and details for service billing or statistics. The policy for the maintenance of the call details has to be properly configured in the system in order to minimize the overhead due to the storage of call details. Most interestingly, the *calls* resource also offers a factory method to instantiate new calls and retrieve existing calls, as explained in subsection 2.5.2.
- The *call* resource represents a video or audio call between two peers. This resource contains all the information that describes the call in terms of signaling and media traffic and call state.

2.5.1 Assigning names to resources

Each resource is identified through a URI. According to the REST guidelines, URI fragments should contain nouns (e.g. call), rather than verbs (e.g., makecall). In this work the *calls* resource is identified through the `http://{servername}/calls` URI; analogously, the identifier of a *call* resource is `http://{servername}/calls/{call_id}`.

2.5.2 Uniform interface

This section describes the operations associated to the *call* resource. These operations are summarized in Table 5.2. The first column shows the resource URIs, the second defines the HTTP methods that must be invoked to perform the specific operation. These actions are identified by an URI and a HTTP method. The third and fourth column indicate if the message body of HTTP request and response is empty, respectively. The fifth column provides a description of the HTTP method. As already seen for the *presence* resource, we used the XML representation of the resource because it is the format that is actually used. The resource can also be expose in other formats (e.g., HTML, JSON).

The following operations are taken in part from a more complex model performed in [Griffin and Flanagan, 2011b]. This article considers other scenarios such as call forwarding, the retention (i.e., during a call, a user receives a second call and puts the call on hold to answer to the new call) and the conference service, but it doesn't specify the notification management in detail as instead we do.

- Creating of a *call* resource, a POST request on the `http://{servername}/calls` URI requests the creation of a new *call* resource and triggers the establishment of the call between the requesting peer and a destination peer specified in the body of the request. The returned response contains the identifier of the newly created resource (i.e. `/calls/{call_id}`). The server receives in the client request all the information regarding the new resource and responds with a 201 "Created" status code.

- Reading of all the *call* resources, the first read operation is performed by using the GET method on the URI: `http://{servername}/calls`. This operation allows to get all the existing *call* resources and then to know the list of calls handled by the system, included the disconnected calls. If the operation is successful, the server sends 200 status code as response.

- Reading of a specific *call* resource, the second read operation requires the URI of a specific resource: `http://{servername}/calls/{call_id}`. As in the previous case the positive response has 200 code.
- Subscription to the events of interest, the first subscription operation allows to receive notifications for each call where the subscriber is involved. It is particularly useful to alert a user on an incoming call. This operation is done by sending a GET request on the URI: `http://{servername}/calls/live`.
- Subscription to the notifications for a specific call, the second subscription operation allows the user to subscribe to a specific call and then receiving notifications to each call updating. This operation is done by sending a GET request on the URI: `http://{servername}/calls/{call_id}/live`. In the following paragraphs we will go into detail with regard to this operation because it requires a more comprehensive and it has a key role in the service behavior.
- Updating of a *call* resource, the update operation is done by invoking the PUT method on the URI: `http://{servername}/calls/{call_id}`. This request contains the new values about Call in the message body. If the operation is successful, the server returns 204 ”No Content” code status in order to specify that the the resource has been overwritten with the new values.
- Cancellation of a *call* resource, the deletion operation of a single resource is done by the DELETE method on the URI: `http://{servername}/calls/{call_id}`. If the cancellation operation is successful, the response returns the 204 ”No Content” status code.

We presented only the cases of corrected behavior. Possible error conditions in the request message, or server-side processing, are handled using the HTTP status codes. For instance, if an user attempts to update the call to an incorrect state, the response to the PUT method has a 406 ”Not Acceptable” status code. Moreover, when an user want to perform operations on the not-existing *call* resource by using a URI is not assigned, the server returns 404 ”Not found” status code.

Table 5.2 shows the operations that can be invoked on the *calls* and *call* resources. GET and POST methods can be invoked on the *calls* resource, while the *call* resource exposes the PUT, GET and DELETE methods.

Table 2.2: REST APIS for *call* resource management

Resource URIs	HTTP Method	Request Message Body	Response Message Body	Description
`/calls`	GET	No	Yes	Retrieve a list of calls
`/calls`	POST	Yes	Yes	Create a new *call* resource
`/calls/{call_id}`	PUT	Yes	Yes	Modify the *call* resource state
`/calls/{call_id}`	DELETE	No	No	Delete the *call* resource
`/calls/{call_id}`	PUT	Yes	Yes	Modify the *call* resource state
`/calls/live`	GET	No	Yes	Subscribe to the events of interest
`/calls/{call_id}/live`	GET	No	Yes	Subscribe to the notifications for a specific Call

2.5.3 Resource representation

The *call* resource representation contains the following data fields:

- *to*, indicates the caller, identified by a URI;
- *from*, indicates the callee, identified by a URI;
- *state*, indicates the call state;
- *offer*, contains the session description, specified according to the Session Description Protocol (SDP) [Handley and Jacobson, 1998], that the caller sends to the callee to request the establishment of a call [Rosenberg and Schulzrinne, 2002].
- *answer*, contains the session description that the callee sends to caller in response to an offer for negotiating the media session establishment.

More specifically, SDP is the protocol used to describe the parameters of media streams used in multimedia sessions and thus it can be used to negotiate the establishment of a media session between two or more peers. SDP messages usually include the following information:

- *Session information*: indicates the session name and purpose, and time in which the session is active.
- *Media information*: indicates the type of media (e.g., video and audio), the transport protocol (e.g. RTP), the media format (e.g. H.261 video and MPEG video), and other transport information (e.g., ports and IP addresses).

For instance, when a caller invokes a POST request on the `/calls` URI to create a new *call* resource, it passes in the request payload a resource representation containing the *to*, *from*, and *offer* data fields. The *answer* data field is provided by the callee when it accepts the call in the *call* resource representation conveyed through the appropriate PUT request, as explained below.

2.5.4 Finite-state machine Model

We modeled the behavior of the *call* resource through a finite-state machine (FSM) representation, as shown in Fig. 2.10.

In order to adapt the implementation of REST-oriented design principles to the main requirements of the real-time communication service to be provisioned, we took as reference the call setup model defined in the SIP standard and specified in terms of INVITE client and server transactions [Rosenberg et al., 2002a]. For the sake of conciseness, hereafter we limit the description of the *call* resource behavior to the case of a successful call and we only analyze some possible error conditions.

In our proposed *call* resource state machine, transitions are fired by REST invocations sent by user agents (i.e. the caller and the callee). When a transition is fired, a corresponding notification action (notify) is performed to inform the other peer that the resource is now in a new state and new transitions are permitted, according to the REST HATEOAS constraint. The handling of asynchronous notifications is a main requirement for real-time communication service design (for instance to notify a peer of an incoming call), but it is a challenge for HTTP-based implementation of REST services [Griffin and Flanagan, 2011a]. For this reason, the next chapter provides details on how we implemented the asynchronous notification delivery to web browsers.

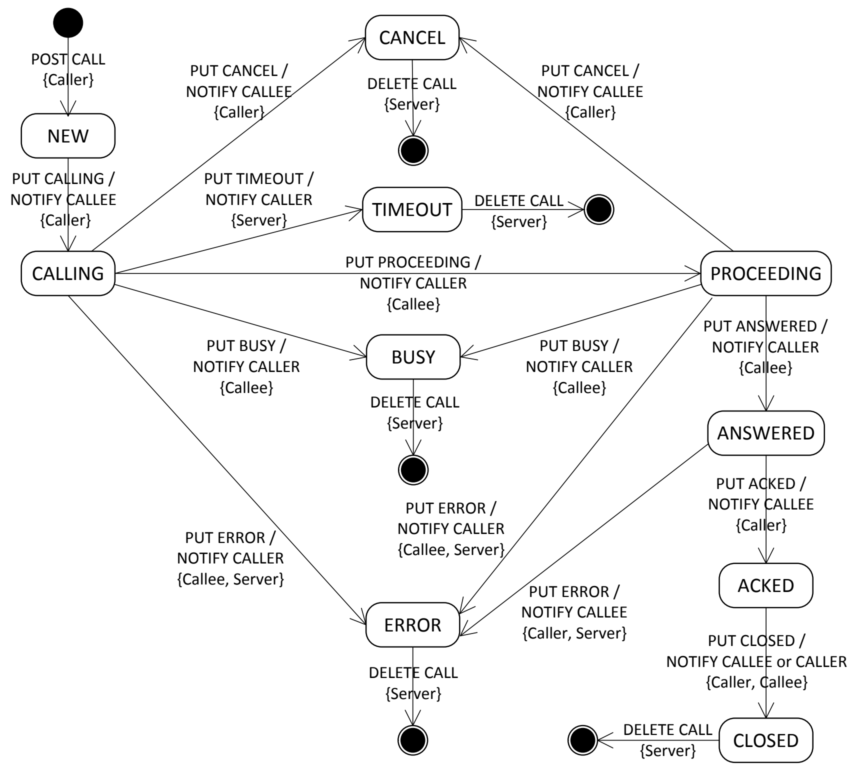

POST CALL {Caller}
NEW
PUT CALLING / NOTIFY CALLEE {Caller}
CALLING
PUT CANCEL / NOTIFY CALLEE {Caller}
CANCEL
DELETE CALL {Server}
PUT CANCEL / NOTIFY CALLEE {Caller}
PUT TIMEOUT / NOTIFY CALLER {Server}
TIMEOUT
DELETE CALL {Server}
PUT PROCEEDING / NOTIFY CALLER {Callee}
PROCEEDING
PUT BUSY / NOTIFY CALLER {Callee}
BUSY
PUT BUSY / NOTIFY CALLER {Callee}
DELETE CALL {Server}
PUT ANSWERED / NOTIFY CALLER {Callee}
ANSWERED
PUT ACKED / NOTIFY CALLEE {Caller}
ACKED
PUT CLOSED / NOTIFY CALLEE or CALLER {Caller, Callee}
CLOSED
DELETE CALL {Server}
PUT ERROR / NOTIFY CALLER {Callee, Server}
PUT ERROR / NOTIFY CALLER {Callee, Server}
PUT ERROR / NOTIFY CALLEE {Caller, Server}
ERROR
DELETE CALL {Server}

Figure 2.10: Finite-state machine of the *call* resource.

In detail, the FSM model shown in Fig. 2.10 represents the state evolution of the *call* resource for the call session setup between two user agents (i.e. the caller and the callee). The subjects that can trigger a transition by a HTTP request are reported in the diagram between brackets.

The *call* resource states are enumerated hereafter:

1. *New*, indicates a newly instantiated *call* resource. When a *call* resource is created the value of the status field is set to NEW. In this state, the call is not actually started. In order to start the call, the caller does the only possible transition leading the *call* resource to the CALLING state and updates the offer field with the SDP value of the caller. This transition causes:

   (a) The Recording of caller to the call, REGISTER CALL.

   (b) The Sending of a SIP INVITE message containing the offer, if the callee user is a SIP client.

   If the SIP client starts the call, it sends an INVITE message to the REST-based service. The service creates the *call* resource and immediately put it in CALLING state by updating the offer field with the SDP received from the SIP client in the INVITE message. The creation of the call and updating to the state CALLING is then notified to the REST client. The non-symmetric behavior is related to the desire to integrate SIP specification in the service. This leads to cases where a SIP message triggers a status change, and other times when a status change emerges the sending a SIP message.

2. *Calling*, indicates an initiated call. In this state, the callee receives the notification of an incoming call. If this does not happen, after a specified period of time, the service switches to the TIMEOUT status. In a other case, the caller wants to close the call and updates the *call* resource status to CANCEL. Both these behaviors are also provided in the SIP specifications. The caller, once sent the INVITE message, waits for a specified period of time, usually 180 seconds, after which it closes the call. These cases depend by the caller. However, there are other cases which depend by the callee. For instance, the call is put into BUSY state; this corresponds in the SIP specifications to send a SIP 480 response message, or more generically into ERROR state,

which corresponds to all the other messages of negative response that can be sent by the UAS. Finally, there is the possibility that the call updates the call to the PROCEEDING state, corresponding to sending a SIP 100 ”Trying” or 180 ”Ringing” response message.

3. *Timeout*, indicates that the call failed due to out of time. If the *call* resource remains in the CALLING or PROCEEDING state for a longer time of a specified time interval, the service updates the call status to TIMEOUT. Once in this state, the caller must cancel the registration to the call, UNREGISTER CALL, and ends the media channel, HANG-UP. If the call is made by a REST client towards a SIP User Agent the transition to this state is dictated by RESTtoSIP Gateway which, not receiving answers to the INVITE message for a specified period, establishes the TIMEOUT state. Alternatively a REST client may provide a timeout that performs this transition in a time defined by the programmer or by the user in case of greater customization.

4. *Cancel*, indicates that the call failed due to the caller-side call cancellation. This status indicates that the caller want to close a previously open session. As for TIMEOUT state, also in this case the actions UNREGISTER CALL and HANG-UP are performed by the caller.

5. *Busy*, indicates that the call failed due to busy callee. The callee sets the state to BUSY if it does not want to establish the call. Of course, the caller has to de-register and close any process of establishing a connection at media level. To notify the other peer, the playback of a tone is also provided for expliciting the event withe a sound.

6. *Error*, indicate that the call failed due to callee-side request errors events. This status is present for the management of all the error responses that can be generated by the SIP protocol to an SIP INVITE request. The caller has to unregister and close any process of establishing a connection at media level.

7. *Proceeding*, indicates a call in progress. In this state we have two notification actions of the call: PLAY proceedingTone and PLAY ringingTone. The execution of one of the two actions is usually associated with the emission of an acoustic signal, accompanied if necessary by one visual, that notifies this state to the caller and callee. For the caller

called this state means that the callee received the call and is deciding how responding. For the callee, instead, this state indicates that the caller is waiting for the response. At this point the transitions can verified that lead to states CANCEL, by the caller, or BUSY, ERROR, ANSWERED by the callee.

8. *Answered*, indicates that the callee accepted the call. Moreover other implicit actions are planned by the callee:

   (a) Recording to the call to receive notifications.

   (b) Recovery of the offer.

   (c) Upgrading of the answer field, after the creation of its SDP.

   If the caller is a UAC which sent previously a INVITE message, when the service updates the resource status to ANSWERED, it sends also a SIP 200 message of positive response in which the body contains the SDP answer. Once the caller is notified of this status change, can get the SDP of the callee. At this point, both are able to establish a media channel taking advantage of the exchanged session information.

9. *Acked*, indicates that the caller confirmed the call. Once the connection was established, the caller updates the resource status to ACKED. If the caller is a REST Client, it sends a PUT request for overwriting the status of the *call* resource to ACKED. Instead, if the caller is a SIP client sends an ACK message which will be after translated in a PUT request to update the *call* resource status to Acked.

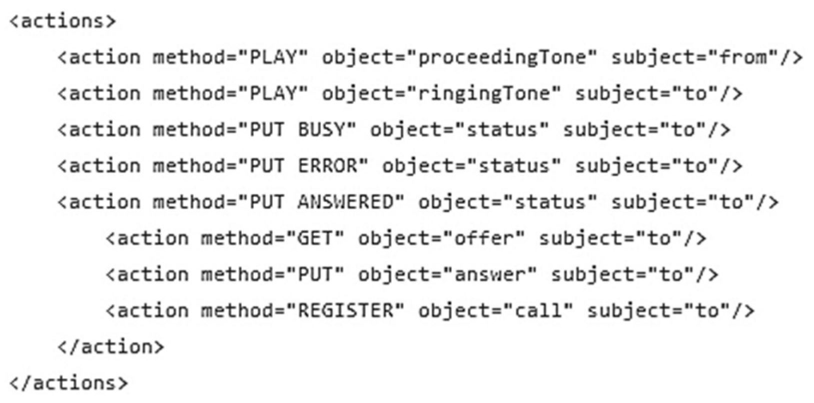
10. *Closed*, indicates that the call is terminated. Once the call is established, then it can also end. In the SIP specifications this is done by sending a BYE message, while in the case of a REST Client is done by a PUT request. This end action can be done either by the caller and callee. As for all the states that precede the call closure, even in this case we have the deregistration and the call closure at media level extended to both users. Moreover the call closure is always followed by the deletion of the resource through the DELETE method. This transaction was never made by the client, but is done automatically by the service.

Starting from the initial pseudo-state the caller performs a POST request to trigger the creation of a *call* resource (NEW state). The next intermediate transitions are all triggered by a PUT request which is sent by the caller/callee for updating the resource state; this event is always followed by a notification action to inform the other peer about the state change. For instance, when the callee accepts a call session, it updates the *call* resource state to ANSWERED through a PUT request containing the *answer* session description. This state change is notified to the caller. At this point the caller updates the *call* resource state to ACKED through a PUT request and this change is notified to the callee. The final transitions that occur in case of failed or closed call, are triggered by the server by means of DELETE operation which deletes the *call* resource. Note that the states *Timeout*, *Cancel*, *Busy*, *Error* and *Closed* can be considered as equivalent, since they all lead to termination after a DELETE operation, and therefore could be merged in a single final state. We have maintained them separate for clarity.

While in SIP User Agent implementations, notification messages contain information strictly related to the call session evolution and next permitted transitions are encoded in the SIP UA implementation logic, in our approach notification messages also contain the reference to the next permitted transitions. We made this choice to fulfill the REST HATEOAS constraint. According to this constraint, an application evolves through subsequent transitions of resources from one state to another. Through resource representations delivered to clients, the system can model and advertise permitted transitions [Parastatidis et al., 2010]. Client applications can thus decide which possible forward steps can be activated based on their specific application goals and/or through end users' actions. The state of the *call* resource thus evolves according to the actions of the two peers. Although HATEOAS is considered one of the main constraints of the REST architectural style, it is often disregarded and the discussion is ongoing for clarifying and translating it into pragmatic guidelines [Liskin et al., 2011]. As discussed also in the next chapter, it is worth observing that the joint adoption of the REST uniform interface and HATEOAS constraints helps in simplifying the development of the client-side application logic, since the semantics of the REST uniform interface operations is defined and permitted invocations are advertised at each step by including hypermedia links in the responses.

2.5.5 Actions for the resource navigation

Previous paragraphs shown the importance of the concept ”hypermedia as the engine of application state”. In service that we designed, we decided to add to the resource a systematic description of the actions that can be taken by the client, depending on the value of the status field. In this way the service user has available a guide to use the *call* resource in order to automate the behavior of the clients by associating to each action a instruction sequence. The actions are listed inside the ”actions” element (for the XML format). For every action there is an ”action” element. Figure 2.11 shows an example of the actions listed in a response along with the *call* resource at PROCEEDING status.



```
<actions>
    <action method="PLAY" object="proceedingTone" subject="from"/>
    <action method="PLAY" object="ringingTone" subject="to"/>
    <action method="PUT BUSY" object="status" subject="to"/>
    <action method="PUT ERROR" object="status" subject="to"/>
    <action method="PUT ANSWERED" object="status" subject="to"/>
        <action method="GET" object="offer" subject="to"/>
        <action method="PUT" object="answer" subject="to"/>
        <action method="REGISTER" object="call" subject="to"/>
    </action>
</actions>
```

Figure 2.11: An example of the actions listed in a response along with the *call* resource at PROCEEDING status.

As shown in Figure 2.11, every action has three attributes:

1. *Subject*, identifies who must make that action. Possible values are ”to”, ”from” and ”both”, to indicate the callee, caller or both, respectively.

2. *Object*, indicates the object of the action. It can be a field of the call, the call itself or another.

3. *Method*, indicates the type of action that should be performed. Among the types of actions there are also HTTP methods. For instance, if the action object is a resource field, then the actions might be GET or PUT methods.

An important case is that of an action that changes the resource status field. The actions of this type guide the call evolution. For instance, if we observe the action that has as method PUT ANSWERED, shown in Figure 2.11,we see that the PUT method is composed by PUT method and a value that indicates the state in which the call can go. In this way, several options in the which the call state can evolve, are given to the client. An action can enclose inside other actions.

2.5.6 UML on the fly Model Checker

In order to simulate the behavior of the RESTful service and its interworking with legacy SIP User Agents, we used the UML on the fly Model Checker (UMC), which is an integrated tool for the construction, the exploration, the analysis and the verification of the dynamic behavior of UML models described as a set of communicating state machines [Mazzanti, 2009, ter Beek et al., 2009]. UMC allows to model a system as a collection of interacting UML state machines, and offers also simulation and model-checking capabilities for verifying the satisfaction of a given set of requirements.

2.5.7 Interworking with SIP

The FSM representation shown in Fig. 2.10 can also model a call between a REST client and a SIP User Agent. The interworking is realized by introducing a proxy component that implements the notification action into SIP messages delivered to the SIP UA and translates the SIP messages sent back by the SIP UA into corresponding REST invocations. This proxy is composed of two modules:

1. SIPMessageSender, implements the notification action according to the SIP specifications.
2. SIPMessageReceiver, translates the SIP messages sent by the SIP UA into REST invocations.

We used the modeling and simulation capabilities of the UMC tool to represent our system as a set of communicating state machines and, then, simulate their behavior. To this purpose, we defined the following state machines: the *call* resource (shown in Fig. 2.10), the SIPMessageReceiver, the SIPMessageSender, the REST client and the SIP User Agent state machines (described hereafter and shown in Figs. 2.12, 2.13, 2.14 and 2.15, respectively).

Fig. 2.12 shows the SIPMessageSender FSM representation. This component is in the Standby state, under resting conditions. From this state, if one of the depicted transitions is activated the machine enters the Executed state. After that, the action related to this transition is performed, the machine returns back to the standby state by a default trigger (*timeout*). All the transitions to the Executed state are triggered by a NOTIFY request event produced by the *call* resource state machine. The transition is enabled upon the satisfaction of a guard condition. As shown in Fig. 2.12 the guard conditions refer to the type of event to be notified (e.g., the newly entered *call* resource state). Each event is followed by a notification action to the SIP User Agent (caller/callee) via a proper SIP message. For instance, if the newly entered *call* resource state is Calling (i.e. PUT CALLING guard condition), the notification message is translated into a SIP INVITE message delivered to the SIP UA.

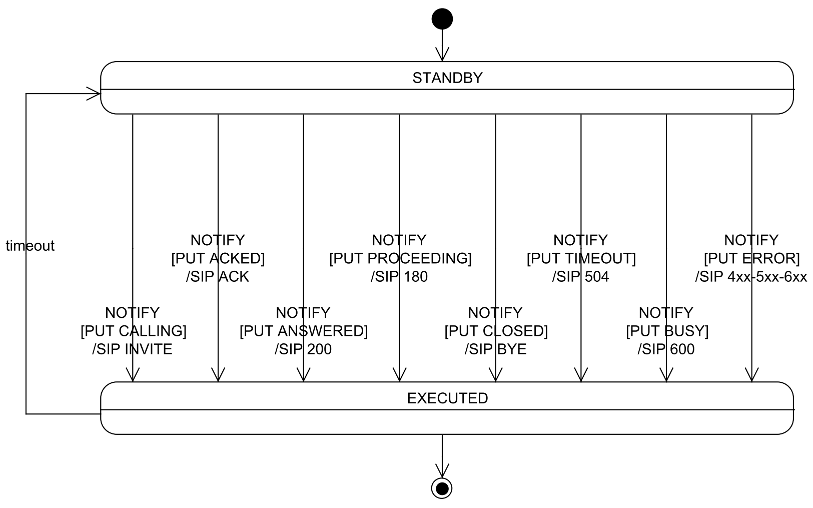

STANDBY
timeout
NOTIFY [PUT ACKED] /SIP ACK
NOTIFY [PUT PROCEEDING] /SIP 180
NOTIFY [PUT TIMEOUT] /SIP 504
NOTIFY [PUT ERROR] /SIP 4xx-5xx-6xx
NOTIFY [PUT CALLING] /SIP INVITE
NOTIFY [PUT ANSWERED] /SIP 200
NOTIFY [PUT CLOSED] /SIP BYE
NOTIFY [PUT BUSY] /SIP 600
EXECUTED

Figure 2.12: Finite-state machine of the SIPMessageSender component.

Fig. 2.13 models the behavior of the SIPMessageReceiver component through a FSM representation. This state machine is similar to the previous one, since it includes only the Standby and Executed states. The event that activates all the transitions to the Executed state is the reception of a SIP Message (SIP_MESSAGE_IN), except a case of reception of a HTTP Message (HTTP_MESSAGE_IN), as shown in Fig. 2.13. The transition is enabled upon the satisfaction of a guard condition. According to the type of SIP message received, the proper REST invocation on the *call* resource is performed. For instance, if the transition is activated by the reception of a provisional response sent by the SIP callee (e.g., 180 Proceeding), a PUT Proceeding operation is invoked on the *call* resource. It is worth noticing that when the SIP UA acts as the caller, the SIPMessageReceiver handles the *call* resource creation triggered by the reception of the SIP INVITE message from the SIP UA by sending a POST request. According to the FSM in Fig. 2.10, upon the reception of a HTTP 201 message sent by the *call* resource component, a PUT Calling request is invoked.

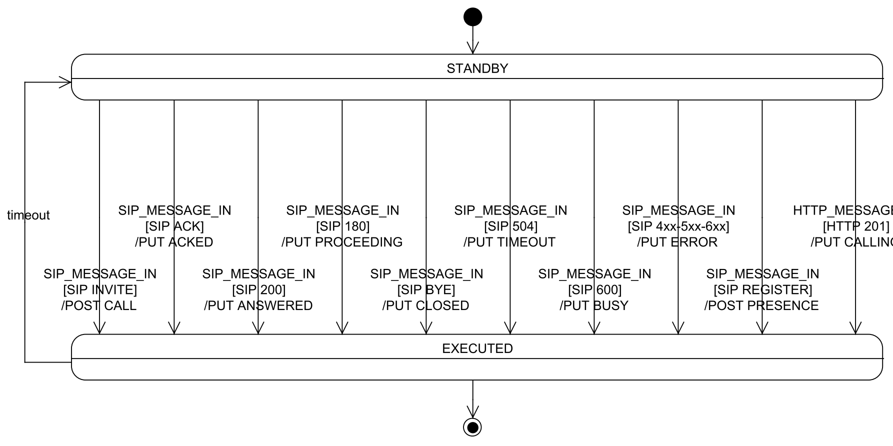

STANDBY
timeout
SIP_MESSAGE_IN [SIP INVITE] /POST CALL
SIP_MESSAGE_IN [SIP ACK] /PUT ACKED
SIP_MESSAGE_IN [SIP 200] /PUT ANSWERED
SIP_MESSAGE_IN [SIP 180] /PUT PROCEEDING
SIP_MESSAGE_IN [SIP BYE] /PUT CLOSED
SIP_MESSAGE_IN [SIP 504] /PUT TIMEOUT
SIP_MESSAGE_IN [SIP 600] /PUT BUSY
SIP_MESSAGE_IN [SIP 4xx-5xx-6xx] /PUT ERROR
SIP_MESSAGE_IN [SIP REGISTER] /POST PRESENCE
HTTP_MESSAGE_IN [HTTP 201] /PUT CALLING
EXECUTED

Figure 2.13: Finite-state machine of the SIPMessageReceiver component.

For the sake of conciseness, hereafter we limit the description of the REST client and SIP User Agent behavior to the case they act as the caller and the callee, respectively, and vice versa, in a scenario of a successful call. Fig. 2.14a shows the REST Client FSM when it acts as the caller, and Fig. 2.14b the REST Client FSM when it acts as the callee. These FSM models have been designed taking into account the transitions and actions of the *call* resource state machine (described in Fig. 2.10), with which the REST Clients must interact. Fig. 2.15a and Fig. 2.15b show the SIP UA Client and Server FSMs, which are based on the SIP INVITE client and server transaction, respectively [Rosenberg et al., 2002a]. As mentioned above, the models shown in Figs. 2.14 and 2.15 are a simplified version of the actual behavior, since they focus on the case of a successful call.

The model of the overall system is shown in Fig. 2.16 by using a UML Component Diagram representation, which describes how a system is split

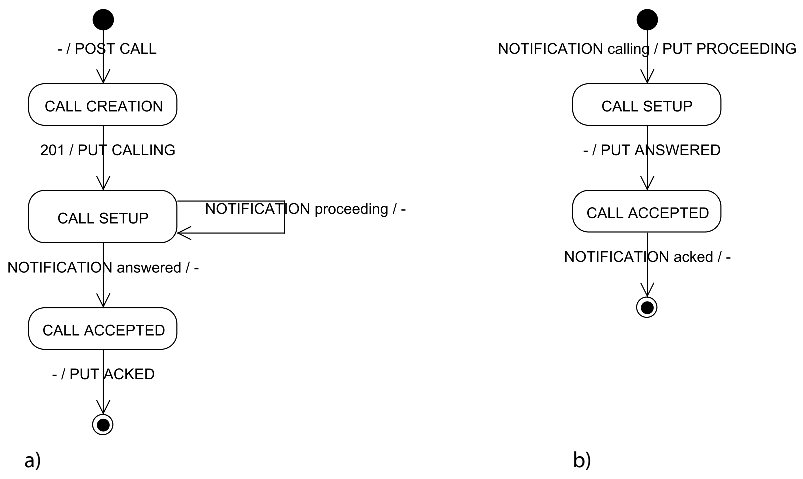


Figure 2.14: Finite-state machines of the REST Client component for a scenario of successful call: a) REST Client acting as the caller, and b) REST Client acting as the callee.

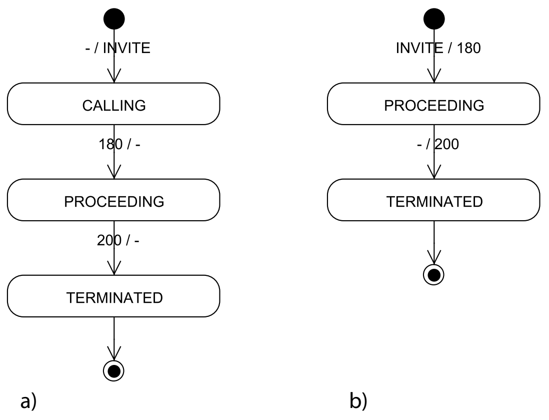


Figure 2.15: Finite-state machines of the SIP UA Client and Server component for a scenario of successful call: a) SIP UA Client acting as the caller, and b) SIP UA Server acting as the callee.

up into components and the dependencies among these components.

A UMC model is specified by providing a set of class declarations, a set of objects instantiations, and a set of abstraction rules. The classes define the structure and dynamic behavior of the objects which compose the system. Thus, each component shown in Fig. 2.16 is an object instance, which is exposed as a state machine.

According to the UML Component Diagram notation, Fig. 2.16 shows the event-based operations exposed by each FSM class interface and the dependency of other classes on these interfaces. For instance, the *call* resource class requires the NOTIFY operation exposed by the SIPMessageSender class.

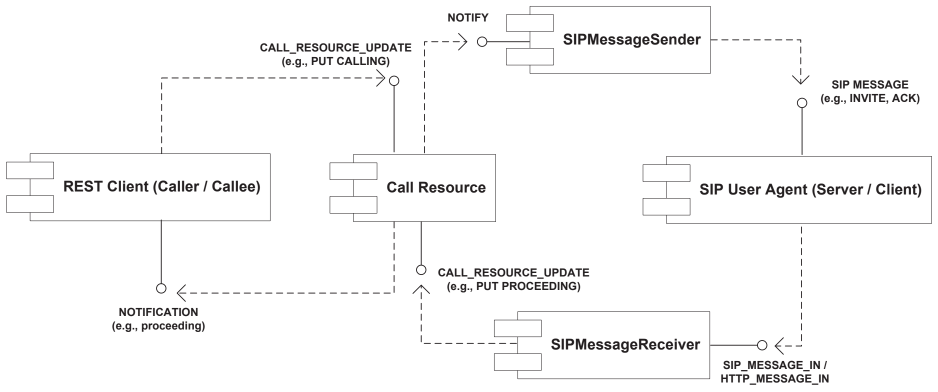

NOTIFY
SIPMessageSender
CALL_RESOURCE_UPDATE (e.g., PUT CALLING)
SIP MESSAGE (e.g., INVITE, ACK)
REST Client (Caller / Callee)
Call Resource
SIP User Agent (Server / Client)
CALL_RESOURCE_UPDATE (e.g., PUT PROCEEDING)
NOTIFICATION (e.g., proceeding)
SIPMessageReceiver
SIP_MESSAGE_IN / HTTP_MESSAGE_IN

Figure 2.16: Component diagram for the communicating state machines model for a call between a REST client acting as the caller and a SIP User Agent acting as the callee, and vice versa.

2.6 Solutions for asynchronicity in HTTP

The HTTP protocol is a request/response model. For this reason, the server does not initiate a connection with a client or send a response that is not explicitly requested. The server can not send asynchronous events to the client [Loreto et al., 2011]. However, this possibility has become increasingly important for least two purposes, [Java.net, 2011]:

1. Decoupling the processing of a request from the request reception service, so as to free threads in order used them in new requests.

2. Supporting non-blocking requests for the client.

Taking as reference point this latter aim, in this section we analyze the various ways in which you can handle this type of request. We analyzes the tools and technologies provided by HTTP or other protocols that can be applied in contexts such as REST. The final objective of this discussion is the implementation of the registration service, as seen in the previous sections, to allow a client to receive notifications about updating one or more resources. In his thesis [Fielding, 2000] Fielding does not indicate any specific method to be followed for the management of such occurrence. However, after some methods have been proposed for the management of asynchronous requests and more generally of the sending of asynchronous messages from the server to the clients. As seen in the state of the art, many articles [Li and Chou, 2010], [Islam and Gregoire, 2013], [Belqasmi et al., 2012], [Griffin and Flanagan, 2011a] and [Davids et al., 2011] explain how to implement a service call, even with SIP protocol, through the use of REST architecture. Some, however, use REST only for the call creation but continue to use the SIP client for the effective communication, as in [Belqasmi et al., 2012]. Others move the communication on the web using browser plugin, [Davids et al., 2011]. Others give suggestions on how to approach the creation of communication services via the web, but do not show an actual implementation, [Griffin and Flanagan, 2011a] [Li and Chou, 2010]. In general, however, there is not a strong stance on how the asynchronous notifications to the clients should be handle in REST. For this reason, in the following paragraph we propose the solutions widely used, for some of these we will offer an implementation in Chapter 3 and in the following chapters we will test their actual behavior and usability.

2.6.1 Periodic GET (polling)

Polling is one of the methods used to observe the resource state. It consists substantially in the resource query at regular intervals to verify the change presence (Figure 2.17). It is often used with the purpose of decoupling the resource processing from the processing request. Often, to conserve bandwidth, the resource is associated with an auxiliary resource indicating the progress of the processing procedure, as suggested in [Thijssen, 2011] and [Fielding, 2008]. The use of periodic queries has as main problem the identification of the time interval between a query and the other, which must be set so as to satisfy the needs of the project.

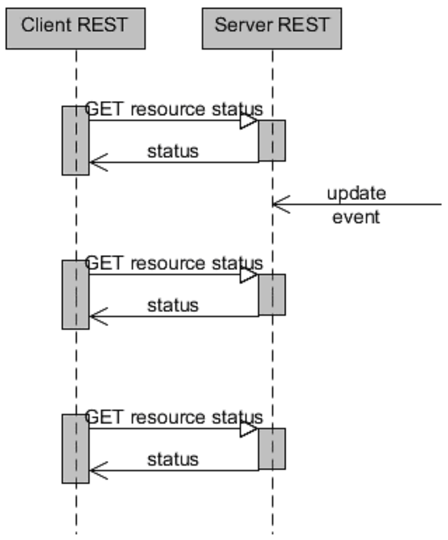


Figure 2.17: Polling from a REST client and a server that provides a REST service.

In the present case the notification times should be short, basically smaller than a second, so even using resources with only a data to explicit the successful update, we would have problems due to:

1. Bandwidth consumption;
2. Service employment;
3. Overhead of the new TCP/IP connection established for each request.

The polling is rather well suited for services where the updates may be notified even after times longer one minute. For this reason this method will not be taken into account in the implementation phase.

2.6.2 Long polling

The traditional technique of polling sends regular requests to the server to get the data. However, if there are not new data, the server needs to send a response. Unlike this, which can be considered ”short polling”, the ”long polling” minimizes the latency for sending a message from the server to the client and decreases the resource use for processing the response and for their sending on the network. To do this we see the life cycle of an application which makes use of long polling with HTTP, as described in [Loreto et al., 2011]:

1. The client makes an initial request and waits for response.
2. The server waits to send a response until an update is available or there is a particular state or a particular time interval expires.
3. When the update is available, the server sends a full response to the client.
4. At this point the client after receiving the response, it sends a new request of long polling type. This can occur immediately or after a certain period of time.

The server then waits to send the response to the client, thus avoiding the continuous exchange of request and response in which there are not effective notifications, as shown in Figure 2.18. This avoids to wast resources in the creation of all TCP connections and HTTP request and response processing. However, the long polling still leaves the connection open between client and server. If this does not necessarily present a problem for the client, it can instead be for the server that with too many clients may congest and not be able to receive other request, Figure 2.19. In our designed service a client should always receive notifications about incoming calls; this would bring each client to have at least one type ”long polling” type connection with the server that could then handle a limited number of users. Moreover for each active call other two connections are needed for the notifications to the callee and caller. This led us to avoid using this method for managing notifications.

2.6.3 HTTP Streaming

This mechanism is a further evolution of the long polling technique. In this case, once the initial request is been made, the server never closes the connection. This is possible because a channel is created to communicate the response to the client and is not never closed, Figure 2.20. This method prevents the client to re-send the request to the server. Below we show the life cycle of an application using this method [Loreto et al., 2011]:

1. The client makes an initial request and waits for response.

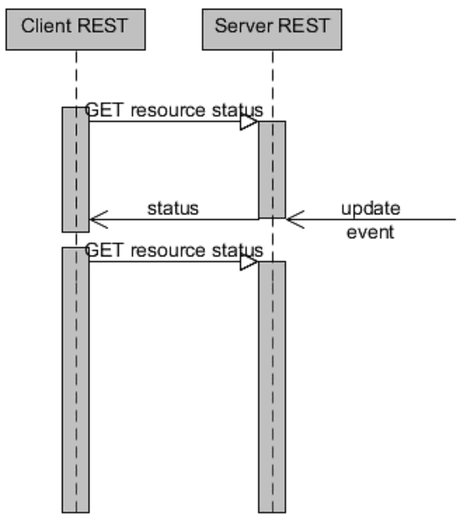


Figure 2.18: Long polling from a REST client and a server that provides a REST service.

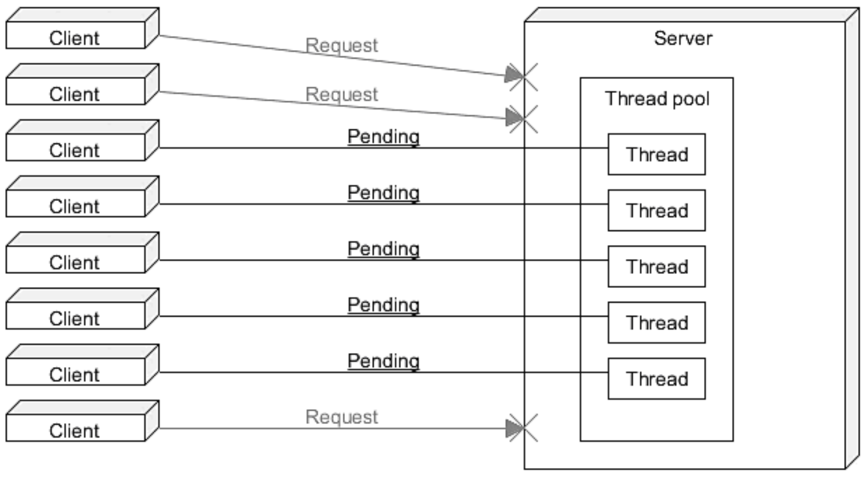


Figure 2.19: A full thread pool that can not receive other requests

2. The server waits to send a response until an update is available or there is a particular state or a particular time interval expires.

3. When the update is available, the server sends to the client as part of the response.

4. The data sent from the server does not end the connection and the server returns to step 2.

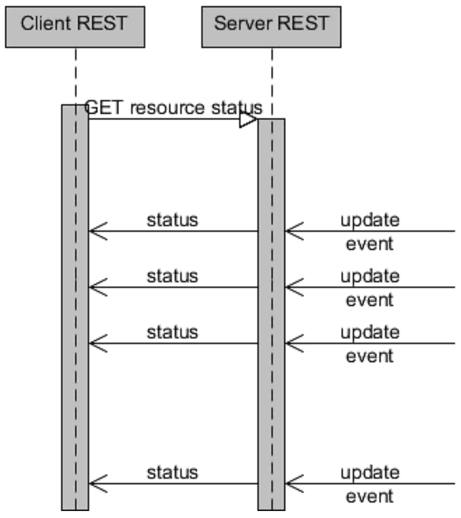


Figure 2.20: HTTP Streaming between a REST client and a server that provides a REST service.

This mechanism is therefore based on the server ability to send parts of information in the same response, without ending the request or connection. To do this we put the value of the "Transfer-Encoding" header field to "chunked". This method, as the name suggests, is used by the streaming server. However, in this way we have the following problems:

1. The intermediaries in the connection between the client and server could buffer the parts with which the response is made. In this way the client does not receive quickly the updates.

2. The management library of the request and response used to implement the client may buffer the response before sending it to the concerned entity. This happens in some browsers.

3. Same problem already descripted for the long polling technique: each client keeps always busy one of the pool threads, as shown in Figure 2.19.

2.6.4 Asynchronous Processing in Servlets

The long polling and HTTP streaming methods have the same problem: the employment of a pool thread that manages the request arrival, while waiting for an event. To this end, in the Servlet 3.0 [Specification, 2011] the asynchronous processing support was introduced. The life cycle is then:

1. The request is received and passed to the servlet.

2. The servlet processes the request parameters and its contents to determine the nature of this.

3. The servlet queues the request in waiting to be freed up resources needed for their processing.

4. The servlet returns to be free to deal with any incoming request without generating a response to the client.

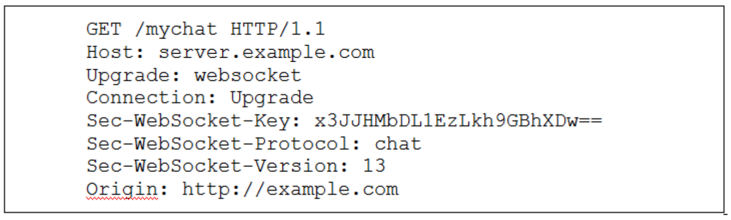
5. After a certain time, the required resources become available and so the response can be processed by an another thread and after sent.

The crucial point is 4. The servlet remains committed while waits for a response to the request, but it leaves the execution to another thread which will care then sending it to the client. In this way the two methods, long polling and HTTP streaming, become more efficient. In the next chapter we will see the their implementation.

2.6.5 WebSocket

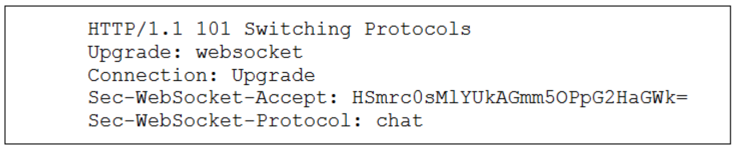
The WebSocket protocol enables two-way communication between a client and a remote host. The protocol consists of a handshake phase followed by messages sent on TCP protocol. The objective of our service is to provide a mechanism for browser applications that need a two-way communication without opening multiple HTTP connections such as polling [Fette and Melnikov, 2011]. To establish a connection, the client sends a WebSocket handshake request to the server (Figure 2.21) which sends it a response (Figure

2.22). Once the answer came back, the connection has been established and both entities are able to send and receive messages from each other, Figure 2.23.



```
GET /mychat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Figure 2.21: Request to the server for a Websocket connection.



```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

Figure 2.22: Response to the client for a Websocket connection

The use of HTTP protocol is limited to the server ability to interpret the request, and then move on WebSocket. Two URI schemes are defined: ”ws:” and ”wss:”, for not-encrypted and encrypted connections, respectively. This communication mode allows to have a full-time connection between the client and server that provides a service similar to the streaming case without all the disadvantages of the case. Although the specification is relatively recent (2011), it can be used as recent versions of popular browsers (e.g., Chrome, Explorer, Firefox) implement it.

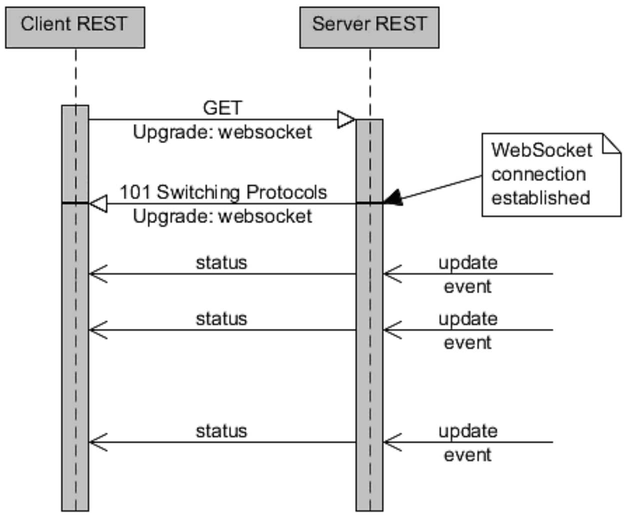


Figure 2.23: Establishment of a WebSocket channel between a REST client and a server that provides a REST service.

Chapter 3

RESTful Service Implementation

This chapter describes the steps taken to implement the RESTful service by taking into account the REST APIS and the state machine specifications described in chapter 2. This prototype offers a real-time communication service (voice and video call) between web browsers as well as web browsers and SIP user agents. The prototype also offers additional services, in particular a presence management service which is needed in order to track users availability status and contact details. In paragraph 3.1 we discuss the most important technologies that we used: Java, Jersey, AsyncContext and WebSocket, SIP servlet API, HTML 5, JavaScript and WebRTC. In paragraph 3.2 we introduce the prototype architecture and describe its components. In paragraph 3.3 we describe the main reference scenarios that we implemented for our prototype. In paragraph 3.4 we describe the server-side packages in which the project has been divided. For each package we describes the most important classes and the role they play. In Section 3.5 we describe the main client-side scripts implemented for our prototype.

3.1 Choice of Technologies

3.1.1 Java

Java is an object-oriented programming language created by Sun Microsystems. Although it was born with the aim of being used for the planning of small electronic devices, after the explosion of Internet has been successful primarily as a planning tool for developing Web applications. The latest version is Java SE 7 Edition [Lindholm et al., 2014]. The main features are:

- Object-oriented;
- Platform-independent;
- Tools and libraries for networking;
- Safe execution of code from remote sources.

Once the application has been created, the code is compiled to get the ”byte-code”; it runs on the Java platform after interpretation by the Java Virtual Machine. The most common implementation of the JVM is the Java Runtime Environment, also included in the Java Development Kit used for development. The Platform-independent feature is given by the possibility to implement the JVM for different environments.

JavaEE, Eclipse e Tomcat

The Java language defines only a part of libraries, the other part is defined by the software platform on which the program is run. For this project we used the Java 2 Enterprise Edition that provides additional features to those found in the Standard Edition. The functionalities of greater interest, in this case, are those relating to the network and to the web, such as servlets. We used as development environment ”Eclipse Java EE for Web Developers” in ”Juno Service Release 1” version. To use the servlets we need a server that implements the functionality of the servlet container, which can run web applications. We decided to use Apache Tomcat, version 7.0.29, which also includes ”Mobicents Sip Servlets 2.0.0 FINAL”.

3.1.2 Jersey

We used the Jersey library, an implementation of JAX-RS (JSR 311) [Potociar, 2009] , to build RESTful type web services [Java.net, 2011]. Its use simplifies the service development as it allows to go to explain the behaviors related to the invocation of HTTP methods on different URLs. To do this, Jersey uses a standard servlet that makes use of specially created classes for each REST resource that it wants to provide. This is supported by the use of annotations (@Path,@Produces,@Consumes,@<http method>, ...) that make services easy to implement and the code easy to manage for reading a third person who is not familiar with the servlets. Jersey supports JSON and XML, through an implementation of JAXB (Java Architecture for XML Binding). In this way the Jersey library can manage the resource sending and receiving in these formats by facilitating the binding operation to the programmer. In addition to providing classes and methods for the REST service creation, Jersey also provides classes for creating REST clients with which it is possible to consult the services created in easy and intuitive way.

3.1.3 AsyncContext e WebSocket

In chapter 2 we introduced several methods for the asynchronicity management. In our implementation we considered three options:

- Long polling via HTTP protocol;
- HTTP streaming via HTTP protocol;
- WebSocket, which uses the HTTP protocol only for the establishment of two-way channel between client and server.

As we already saw in chapter 2, to improve the performance of the first two methods we used the asynchronous mode provided by the Servlet 3.0 API implemented in Tomcat servlet container, version 7.0. Once the request has been received and verified the possibility to operate asynchronously, an AsyncContext object is created. This object allows:

- To start a thread through the Runnable interface use, which can use the methods provided by AsyncContext.
- To pair a AsyncListener where we can specify the actions to take when certain events occur:

  - starting of the asynchronous context;
  - complete processing, event originated by its method invoked on the AsyncContext object;
  - error, in case of incorrect processing;
  - timeout, if the maximum time for the request processing terminates, for instance due to the same thread blockage processing the request. The initial value can be set as desired: if is negative disables its use, as we do in our case.

- To use a stream to send the response to the client in one or more parts; this is essential for the HTTP streaming method implementation.

In our case, the asynchronicity use is not due to the long request processing time, but to the need to be able to notify a client an event. For this reason, any thread is instantiated to the AsyncContext creation. Instead, we are interested to the ability to manage the AsyncContext stream so that it can be associated with a subscription request and then retrieve it to send notifications when the resource changes. The use of WebSocket technology involves the creation of a dedicated servlet to the which a GET request is sent for making a upgrade operation to its protocol. At this point, an OutStreamInbound object is created, which is used to send information to the client that requested the connection. This object will be used to notify the client updates about the subscribed resource.

3.1.4 SIP servlet API

The most interesting properties of these APIs are [Kulkarni and Cosmadopoulos, 2008]:

- The ability to send SIP messages, allowing to make signaling. This is possible because SIP servlet API can act as UAC, UAS and proxies.

- Simplicity: containers handle "non-essential" complexity such as managing network listen points, retransmissions, CSeq, Call-ID and Via headers, routes, etc.

- Converged Applications: containers support converged applications that use multiple protocols and interfaces. In our case we managed HTTP protocol, for REST services and SIP protocol, for the communication with its clients.

- Applications composition, a request arriving at the container can be processed by multiple applications invoked according to a established order.
- SIP servlets enable the request reception by SIP clients, their processing and response sending via the servlet container, which:
  - provides network services on which request and response are received and sent;
  - manages the network listen points (IP, Transport protocol, port) on which it waits for SIP traffic;
  - decides which applications to invoke and in what order;
  - contains and manages servlets for their life cycle.

In previous chapter we talked about SIP and HTTP protocol and their similarities. These affinity are also reflected with regard to the relative servlet which derive from the same class GenericServlet [Kulkarni and Cosmadopoulos, 2008]. The main differences between the two technologies [SunMicrosystems, Inc., 2008] are reported below:

- HTTP servlets have a particular context in which they are performed (called context-root), while SIP servlets have not.
- HTTP servlet usually return HTML pages to the clients while SIP servlets are used to connect SIP hardware and to enable communication between client and server.
- SIP is a peer-to-peer protocol, as opposed to HTTP. SIP servlet can originate requests, while HTTP servlet can only send HTTP responses to requests created by the client.
- SIP servlets often act as proxies to other SIP endpoints, while HTTP servlets are typically the final endpoint for the incoming HTTP request.
- SIP servlets can generate multiple response to a particular request.
- SIP servlets can communicate asynchronously and are not obliged to respond to the incoming request.

- SIP servlets often work in collaboration with other SIP servlets to respond to specific SIP requests, while HTTP servlets typically are only responsible for the response to the HTTP request.

In this work the role of the SIP Servlet is very important because allows to interact with SIP clients receiving the request and the response from those submitted. Regarding the sending of messages to the SIP client, we used BrowserSipCallHandler class (as we will see further) that by observing the state variations of the Call resource, sends its message to the SIP client.

3.1.5 HTML 5

HTML5 is the new standard for HTML [Hickson and Hyatt, 2011] obtained from the cooperation, established in 2006, including the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). In this section we give a brief introduction to this technology, paying attention to the most important features for the web-application creation. The web evolution has led to need a new version for going meet the developers needs, since the previous 4.01 version of HTML dates from 1999. The HTML5 version defined in December 17, 2012 will be supported over time by all browsers. The base rules of this standard are:

- New features based on HTML, CSS, DOM, and JavaScript.
- Reduction of external plugins.
- Better management of errors.
- More markup designed to replace the scripting need.
- Independence from the used device type.
- The developing process plans to be visible to the public.

Some interesting features introduced by HTML 5:

- Element <canvas> for 2D drafting.
- Elements <audio> and <video> for playback of multimedia resources.
- Local storage support.

- New items for the contents: <article>, <footer>, <header>, <nav> and <section>.
- New control types for the form: calendar, date, time, email, url, search.

For more details reading [Hickson and Hyatt, 2011].

WebSocket

The WebSocket protocol was already described in the previous chapter. Here we go quickly to illustrate the ease with which HTML5 and JavaScript technologies allow to use this technique [Hickson, 2011]. The connection creation is through the WebSocket object instantiation to whose constructor the service URL is passed. To handle events related to the created connection, we can assign some callback, as outlined below:

- OnOpen, is invoked at the connection opening;
- Onerror, is invoked at the error occurrence, it returns an error object;
- OnMessage, is invoked when a message arrives, it returns a message object;
- OnClose, is invoked at connection closure.

To send the data we use the send method. This method requires that at the invocation the information to be sent is passed as field.

JavaScript

JavaScript is a scripting language [Danesh and Tatters, 1996], object-oriented and has a simple syntax and lightweight that allows to get dynamic and interactive web pages. IT is the world's most popular scripting language and widely used for creating websites. The first standardization was in 1999, the last in 2011. The main language features are reported below:

- The code is not compiled but interpreted. The interpreter is included within the browser.
- The client-side scripts allow interaction with the user with the ability to handle events such as clicking a button or entering text.

- The script can communicate asynchronously with the server. The technique called AJAX allows to send and receive data from the server asynchronously in background. In this way it is possible to reload page individual parts with a gain, both at the performance level that use experience by the user.

- Ability to manipulate the HTML page contents (both for reading and writing).

- Syntax similar to that Java with the possibility of using the most common constructs: if, while, switch, etc.

3.1.6 WebRTC

This technology has been discussed in the state of the art; here we do an overview of the W3C APIs use and their implementation in browsers [Bergkvist et al., 2014]. Regarding the implementation, browsers currently support mostly WebRTC are Mozilla Firefox and Google Chrome. Mozilla divides the development phases of its Firefox browser in Beta, Aurora and Nightly. At present, the version 23 is stable and the WebRTC API are enabled by default. Google Chrome supports APIs getUserMedia from version 21 in regular way and without the need to enable any flag. Regarding the API RTCPeerConnection, which allow the connection of two browsers directly, these are enabled by default from the version 23. Then, starting from this Google Chrome version the WebRTC API (i.e, getUserMedia and RTCPeerConnection) use is permitted, without the need to enable any flags. At present, we use the version 38. The functionalities of these APIs have been submitted by the work teams of Chrome and Firefox browsers through the video-call establishment in February 2013. The communication has been established between a user who was using Firefox, version 21, and another who was using Chrome, version 23.

3.2 Prototype Architecture

The prototype is a web application made of the following main modules, as shown in Fig. 3.1:

- *CallService Interface*, handles the RESTful exposure of the call service to web browsers. It also offers further services, namely registration and presence update subscription services.
- *CallService Logic*, contains the application and persistence logic that implements the call and presence management services.
- *Notification Manager*, is responsible to notify web browsers of events they subscribed to (e.g., incoming calls, state changes in a call setup).
- *REST-SIP Gateway*, handles the interworking with SIP User Agents (i.e., it allows to establish a call between a web browser and a SIP User Agent).
- *Client-side Logic*, consists in a set of JavaScript codes which are executed by the web browser for handling the signaling message exchange and the media channel establishment.

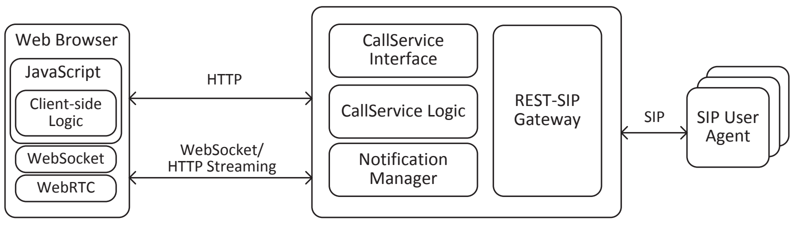


Figure 3.1: Functional architecture of the prototype.

3.2.1 Call Service Interface and Logic details

This Java-based web application has been deployed on an Apache Tomcat 7.0 servlet container. The implementation of the CallService Interface is based on Jersey, a Java-based framework for developing RESTful Web Services serving as Reference Implementation of JAX-RS specifications. The CallService Logic includes the Call and Presence classes, which represent the *call* and *presence* resources, and the classes that handle the connection with the database for the data persistence.

Event notifications towards the REST Client and SIP UA are handled by the Notification Manager and REST-SIP Gateway, respectively.

3.2.2 Notification Manager

The Notification Manager implements the observer design pattern. It listens for the updates of *call* and *presence* resources and notifies registered clients.

We chose to implement three alternative solutions for handling asynchronous notifications:

- the first solution uses the WebSocket Protocol, which provides web browsers with a basic bidirectional channel for message exchange over TCP [Fette and Melnikov, 2011],
- the second solution is based on the Long Polling mechanism and the asynchronous processing of HTTP requests provided by application containers implementing the Servlet 3.0 specifications [Juneau, 2013].
- the third solution is based on the HTTP Streaming mechanism and the asynchronous processing of HTTP requests (as in the previous case).

Implementation details for these mechanisms are reported in Section 3.4.7, while their comparative evaluation is discussed in Chapter 5.

3.2.3 REST-SIP Gateway

The REST-SIP Gateway is made of two main components, called SIPMessageSender and SIPMessageReceiver, which implement the FSMs in Fig. 2.12 and Fig. 2.13, respectively. The SIPMessageSender handles the delivery of notification messages directed to SIP User Agents. More specifically, it listens for notification messages directed to SIP UAs and translates them in the appropriate format and transport protocol, according to the SIP specifications. The SIPMessageReceiver handles the communication in the opposite direction. It receives messages originating from SIP User Agents and parses and translates them into proper actions (e.g. the corresponding REST invocation on the call resource). Both components have been developed according to the Sip Servlet programming model and have been deployed in the Mobicents Sip Servlets platform [Ivanov, 2008].

3.2.4 Client-side logic

This component is made of JavaScript files that are processed by web clients for handling the exchange of signaling messages with the server and establish the media channel with the other peer. This prototype works with web browsers that support the WebRTC API and the WebSocket protocol [Bergkvist et al., 2014].

These scripts handle the interaction with the user, the invocation of REST methods and the handling of notifications sent by the server. The establishment of the media channel relies on the WebRTC API, namely the *getUserMedia* function, which allows a web browser to access the camera and microphone resources, and *PeerConnection*, which sets up a direct channel with another browser for the transport of media data.

The call setup is handled by a set of JavaScript functions, which execute basic actions, such as playing the ringing tone, interpreting the media channel description received by the callee (offer) and preparing the answer message in order to negotiate the peer connection setup. The execution flow of these actions is triggered by two type of events: user-generated events and notifications pushed by the server. As mentioned above, the server notifies the client when the resources of interest change their state. The notification messages contains the representation of the resource and the list of permitted transitions. This information is translated into a set of actions that can be executed automatically by the web browser or upon a user-generated event. For instance, when a server notifies an incoming call, it sends a message to the callee that indicates the current state of the resource (Calling) and the list of permitted next transitions (i.e., the transitions to the Proceeding, Busy or Error states). The Proceeding state is associated to a set of locally executable actions, such as playing the ringing tone to alert the end user.

Through this mechanism our call service implementation aims at satisfying the REST HATEOAS constraint. The adoption of this constraint has the advantage of promoting the decoupling of the client and server logic, thus easing the maintenance of the client logic if the server-side logic changes, while guaranteeing that the client behaves coherently with the application state machine.

3.3 Prototype Scenarios

We chose to implement this prototype in Java, using the Eclipse IDE. This section gives a description of the scenarios that we implemented for our prototype.

3.3.1 Registration and Deregistration

Before accessing the call service, REST clients and SIP User Agents have to register their presence information.

Fig. 3.2 shows the presence registration of a REST client. It directly sends a POST request on the `/presences` URI to trigger the creation or update (if already existing) of the *presence* resource for that user. Then, it subscribes to the events of interest and creates a notification channel, as explained in section 3.2.2. The notification is assigned to an **Observer* object dedicated to a specific REST client in order to notify it the incoming calls. Implementation details for subscription and notification actions are reported in Section 3.4.7.

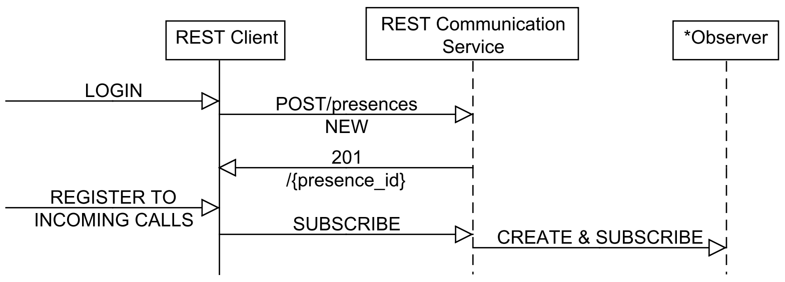


Figure 3.2: Presence registration of a REST client.

Fig. 3.3 shows the presence registration of a SIP client. A SIP UA sends a SIP REGISTER message to the REST-SIP Gateway to provide the server with the contact details needed for the delivery of the events of interest, such as incoming calls. The SIP REGISTER message is translated into a POST request on the `/presences` URI to create the corresponding resource instance. There are two differences compared to the previous case: i) the REST communication service is not directly involved in the presence registration and, ii) absence of registration to incoming calls because it is not necessary.

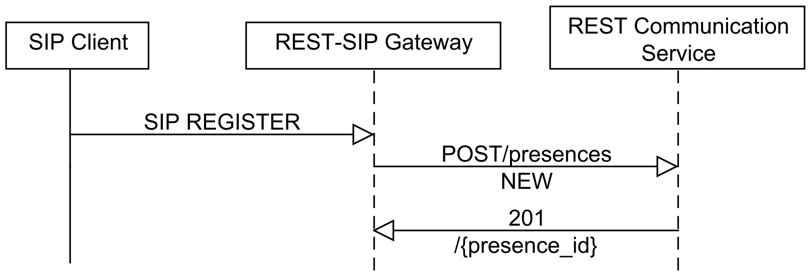


Figure 3.3: Presence registration of a SIP client.

Fig. 3.4 shows the presence deregistration of a REST client. It directly sends a DELETE request on the `/{presence_id}` URI to trigger the cancellation of the *presence* resource for that user. In addition to the resource cancellation, all objects used to send notifications related to the Client are also deleted.

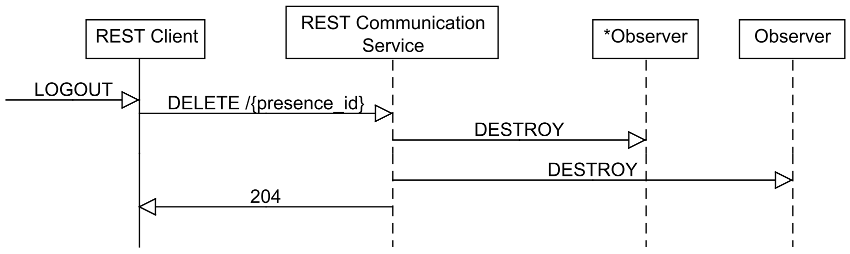


Figure 3.4: Presence deregistration of a REST client.

Fig. 3.5 shows the presence deregistration of a SIP client. A SIP UA sends a SIP REGISTER message to the REST-SIP Gateway to trigger the cancellation of the *presence* resource for that user. The SIP REGISTER message is translated into a DELETE request on the `/{presence_id}` URI to cancel the corresponding resource instance.

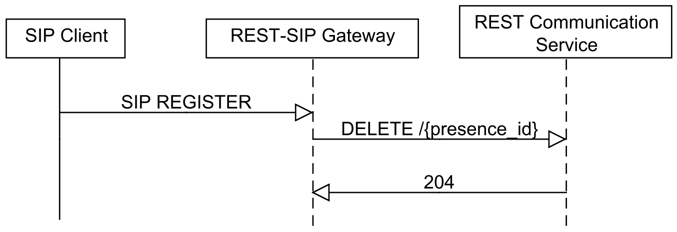


Figure 3.5: Presence deregistration of a SIP client.

3.3.2 Successful Call setup

In this section we show three examples of the test scenarios that we performed to verify that the implemented prototype behavior is coherent with the specifications formalized through the communicating state machines seen in the previous chapter.

Fig. 3.6 shows the message flow for a successful call session setup between two web browsers mediated by our web application prototype. First, the caller sends a call session setup request through a POST request on the `/calls` URI and subsequently subscribes to this resource so as to receive the state updates. This is done by creating an Observer object which has the task of monitoring the resource and notify changes to the client. Once the caller has requested the call establishment by updating the call status to CALLING, this change will be notified to the *Observer instance of the callee. The task of this observer is to monitor the changes of all calls in order notify to the callee the changes about its incoming calls. The notification message contains also the offer session description, i.e. the set of media streams and codecs the caller wishes to use, along with the IP addresses and ports the caller would like to use to receive the media [Rosenberg and Schulzrinne, 2002]. For the sake of conciseness, we don't show the use of the ICE protocol [Rosenberg, 2010] for NAT traversal, which is recommended in the WebRTC specifications [Bergkvist et al., 2014]. The callee updates the call status to PROCEEDING through a PUT request and, locally, plays the ringing tone to alert the end user. This status can persist for some seconds and is notified to the user at the caller side by playing a default beep. When the end user accepts the call, the callee performs the following actions:

- subscription to the call by the creation of its observer;

- it parses the session offer and generates the answer;
- it requests a transition of the call resource to the ANSWERED status through a PUT request carrying the answer.

When the caller receives the notification message, it parses the answer to establish the media session according to the negotiated parameters and, finally, updates the call resource to the ACKED status. Now the call has been established and the end users can talk to each other.

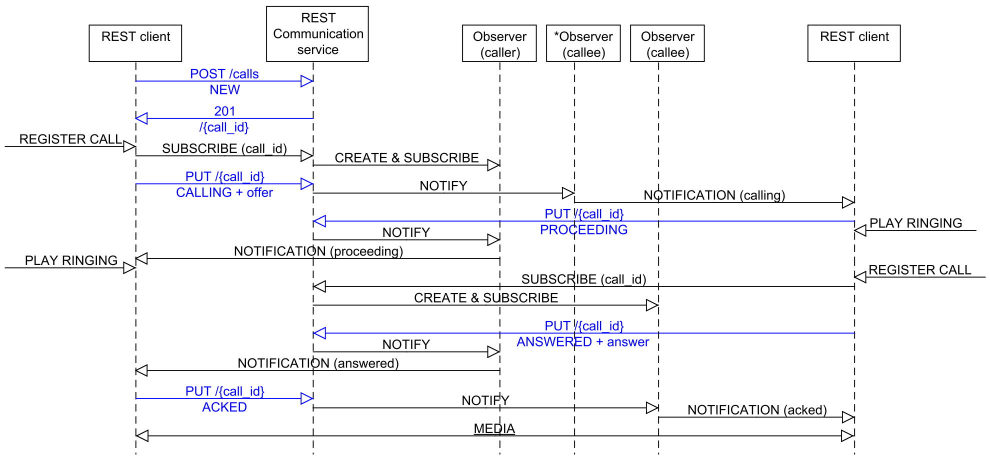


Figure 3.6: Call setup between two web browsers.

Fig. 3.7 and Fig. 3.8 show an analogous message flow for a successful call setup between a web browser acting as the caller and a SIP User Agent acting as the callee and vice versa. The web browser interacts with the server components as in the previous scenario. The interaction with the SIP User Agent is handled by the REST-SIP Gateway (i.e. the SIPMessageSender and SIPMessageReceiver components) in compliance with the design specifications described in the chapter 2.

It is worth observing that the interactions shown in the previous sequence diagram are coherent with the evolution chart of the communicating state machines generated by the UMC tool for a successful call setup. Fig. 3.9, Fig. 3.10 and Fig. 3.11 show an excerpt of the chart representing the evolution of the state machines from the first POST request to a PUT PROCEEDING invocation for the three reference scenarios (for the sake of conciseness, we have omitted the subscription messages).

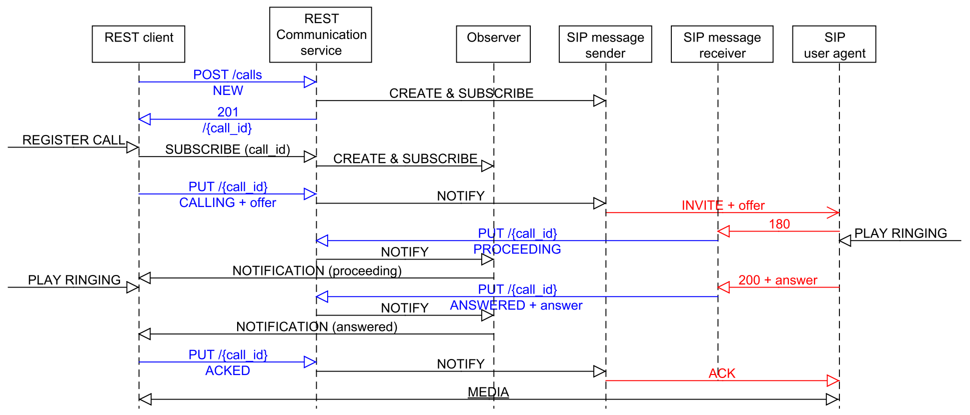


Figure 3.7: Call setup between a web browser (caller) and a SIP User Agent (callee).

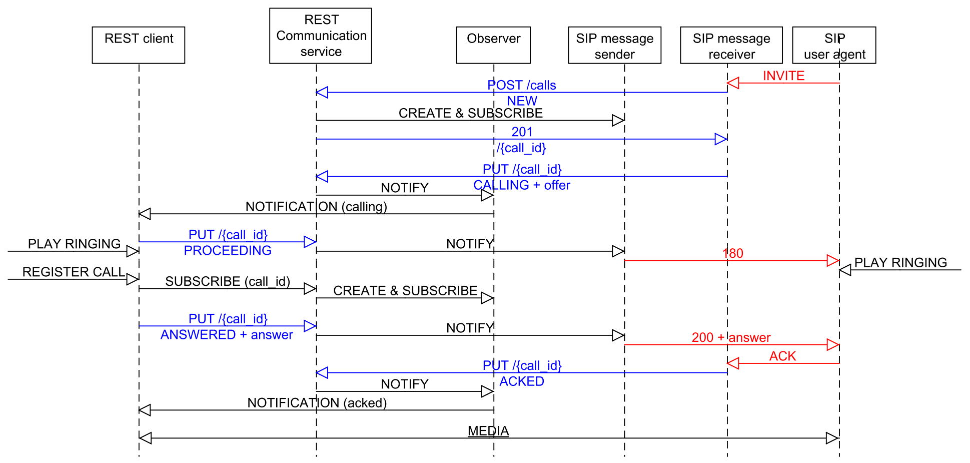


Figure 3.8: Call setup between a SIP User Agent (caller) and a web browser (callee).

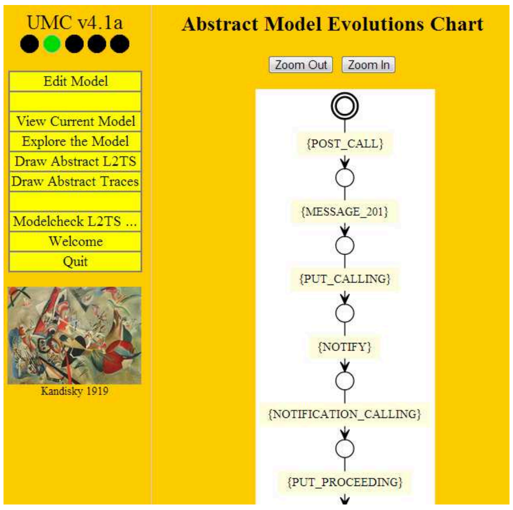


Figure 3.9: Excerpt of the FSM evolution chart generated by the UMC tool for a call setup between two web browsers.

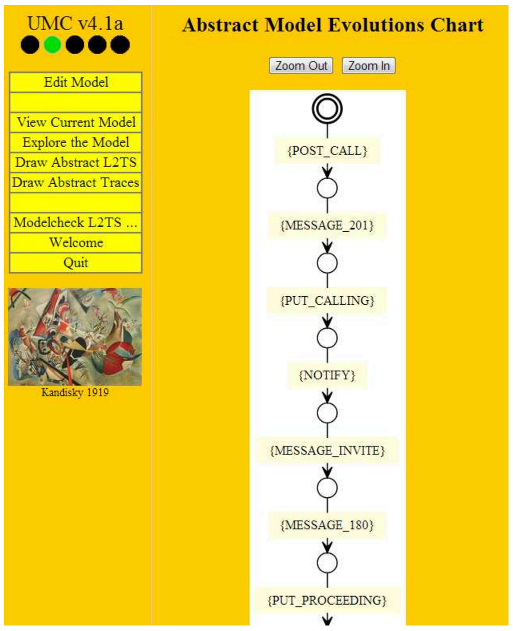


Figure 3.10: Excerpt of the FSM evolution chart generated by the UMC tool for a call setup between a web browser (caller) and SIP User Agent (callee).

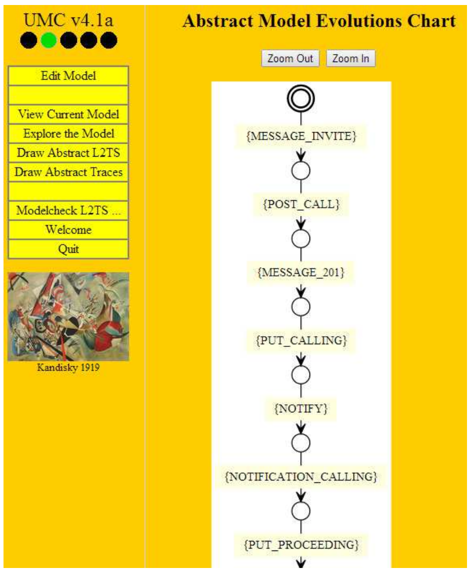


Figure 3.11: Excerpt of the FSM evolution chart generated by the UMC tool for a call setup between a SIP User Agent (caller) and a web browser (callee).

3.3.3 Unsuccessful Call setup

The call failure scenario is very similar to the successful case. The only difference is the final answer from the callee. Figure 3.12 shows an example of generic error or busy callee. The callee claims being busy or having encountered an error and then rejects the call. The answer is BUSY/ERROR. The caller must now deregister to the call and close any process of establishing a connection to media level. The service will delete the call resource.

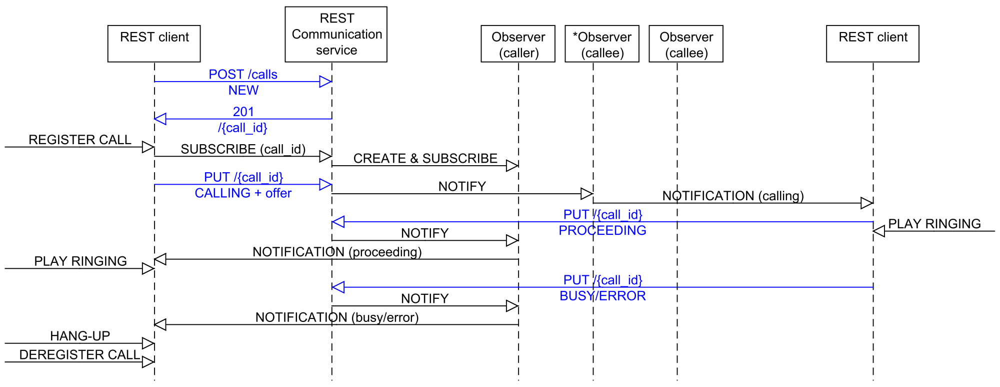


Figure 3.12: Call failure between REST clients due to the callee.

Figure 3.13 shows an example where the caller cancels the call. The process is similar to the previous ones until the PROCEEDING state. The only difference is the caller makes the decision to close the call before the callee answers.

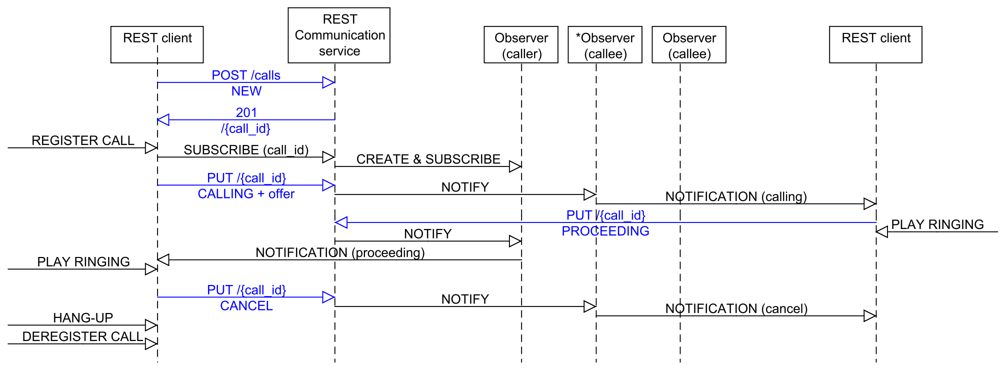


Figure 3.13: Call failure between REST clients due to the caller.

Figure 3.14 shows an example where the callee doesn't respond within the limit waiting time (timeout). In this case the state change of the call resource is made directly by the RESTful call service.

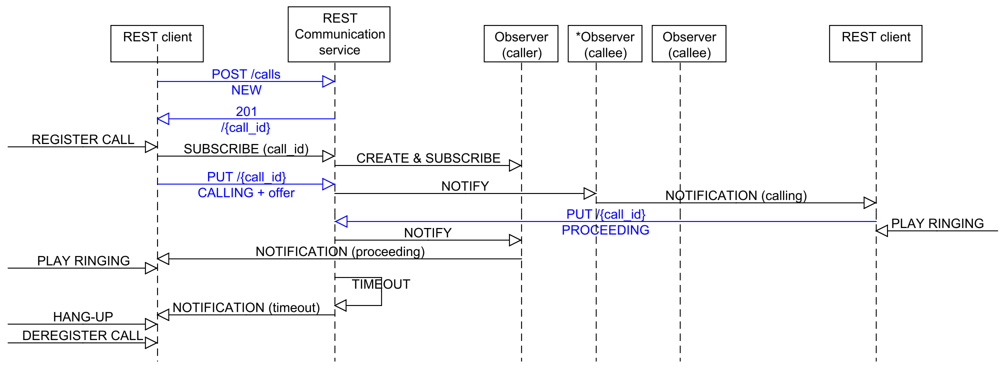


Figure 3.14: Call failure between REST clients due to the timeout.

3.3.4 Call termination

The last scenario is the call termination in which the users had correctly established a call session (Figure 3.15). When a user wants to close the session invokes the PUT operation by updating the call resource status to CLOSED. Subsequently the other user receives the update notification. Finally both users provide to close media-level peer-to-peer connection and cancelling the subscription to the call notifications. Subsequently, the service will eliminate the call resource.

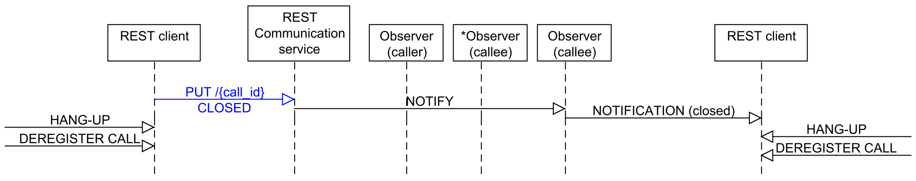


Figure 3.15: Call termination between REST clients.

3.4 Package and Class

The project was divided into several packages in order to organize better the code and to group classes according to the role they play. In Figure 3.16 we can see the main packege structure. In the next paragraphs we explain the functionality that each package has in the project.

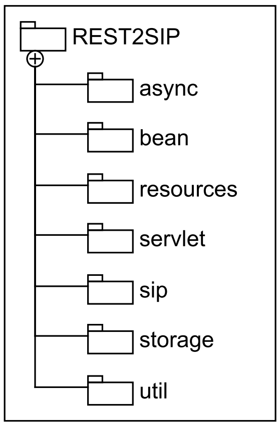


Figure 3.16: Package Structure of the project.

3.4.1 Package resources

The classes included in this package carry out the task to create REST interface of the call service (i.e., CallsResources and CallResource) and presence service (i.e., PresencesResource and PresenceResource) defined in Chapter 2, as shown in Figure 3.18. We describe in detail some functions of these classes so as to highlight the Jersey role in the REST resource design.

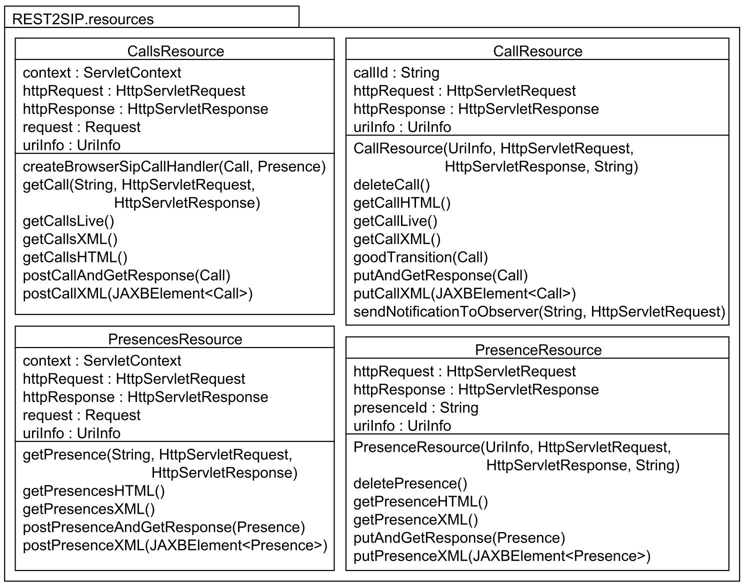

REST2SIP.resources
CallsResource
context : ServletContext
httpRequest : HttpServletRequest
httpResponse : HttpServletResponse
request : Request
uriInfo : UriInfo
createBrowserSipCallHandler(Call, Presence)
getCall(String, HttpServletRequest, HttpServletResponse)
getCallsLive()
getCallsXML()
getCallsHTML()
postCallAndGetResponse(Call)
postCallXML(JAXBElement<Call>)
CallResource
callId : String
httpRequest : HttpServletRequest
httpResponse : HttpServletResponse
uriInfo : UriInfo
CallResource(UriInfo, HttpServletRequest, HttpServletResponse, String)
deleteCall()
getCallHTML()
getCallLive()
getCallXML()
goodTransition(Call)
putAndGetResponse(Call)
putCallXML(JAXBElement<Call>)
sendNotificationToObserver(String, HttpServletRequest)
PresencesResource
context : ServletContext
httpRequest : HttpServletRequest
httpResponse : HttpServletResponse
request : Request
uriInfo : UriInfo
getPresence(String, HttpServletRequest, HttpServletResponse)
getPresencesHTML()
getPresencesXML()
postPresenceAndGetResponse(Presence)
postPresenceXML(JAXBElement<Presence>)
PresenceResource
httpRequest : HttpServletRequest
httpResponse : HttpServletResponse
presenceId : String
uriInfo : UriInfo
PresenceResource(UriInfo, HttpServletRequest, HttpServletResponse, String)
deletePresence()
getPresenceHTML()
getPresenceXML()
putAndGetResponse(Presence)
putPresenceXML(JAXBElement<Presence>)

Figure 3.17: Package Resources contains the classes that implement the REST interface.

CallsResource

The CallsResource class manages the resource identified by the URI: `http://{servername}/calls`. The most significant methods are presented in Table 3.1, which shows the annotations used by Jersey for the call resource management when HTTP requests arrive.

- The first two methods return to the client a list of existing calls. The first in XML or JSON, the second in HTML format. These formats are specified in the header ”Accept”.
- The third method is used to handle requests for subscription to incoming calls by the requesting client through the AsyncContext use.

- The fourth method creates a new call: in fact the REST corresponding method is a POST request. The client provides the new resource in XML format in the request body.
- The last method allows to require a specific resource when the URI is followed by the call identifier. Therefore, an CallResource object is created to process the request.

Table 3.1: Main methods of the CallsResource class.

Java Method	REST Method	Returned or Expected Formats
`getCallsXML()`	`@GET`	`@Produces({MediaType.APPLICATION_XML}` `MediaType.APPLICATION_JSON`
`getCallsHTML()`	`@GET`	`@Produces({MediaType.TEXT_HTML})`
`getCallsLive()`	`@GET`	
`postCallXML()`	`@POST`	`@Consumes(MediaType.APPLICATION_XML)`
`getCall`		

CallResource

This class handles the requests specifically made to a single call identified by its Id. The URI that identifies a resource of this type has a syntax like this: `http://{servername}/calls/{call_id}`. The most significant methods are presented in Table 3.2:

- The first three methods are the corrsponding version for single resource of those seen in the previous subsection.
- The fourth method updates the resource state and notifies any subscriptions.
- The last method is used to delete this resource.

PresencesResource

This class manages the resource identified by the URI: `http://{servername}/presences`. The most significant methods are presented in Table 3.3:

- The first method allows to get all the Presence resources created in XML format.

Table 3.2: Main methods of the CallResource class.

Java Method	REST Method	Returned or Expected Formats
`getCallXML()`	`@GET`	`@Produces({MediaType.APPLICATION_XML}` `MediaType.APPLICATION_JSON`
`getCallHTML()`	`@GET`	`@Produces({MediaType.TEXT_HTML})`
`getCallLive()`	`@GET`	
`putCallXML()`	`@PUT`	`@Consumes(MediaType.APPLICATION_XML)`
`deleteCall`	`@DELETE`	

- The second method allows to get all the Presence resources created in HTML format, so that presenting the web page to the user by integrating the code returned by the GET request.
- The third method allows the creation of a resource Presence.
- The last method allows to require a specific resource when the URI is followed by the identifier of a presence. Then, a PresenceResource object is created to process the request.

Table 3.3: Main methods of the PresencesResource class.

Java Method	REST Method	Returned or Expected Formats
`getPresencesXML()`	`@GET`	`@Produces({MediaType.APPLICATION_XML}` `MediaType.APPLICATION_JSON`
`getPresencesHTML()`	`@GET`	`@Produces({MediaType.TEXT_HTML})`
`postPresenceXML()`	`@POST`	`@Consumes(MediaType.APPLICATION_XML)`
`getPresence`		

PresenceResource

This class handles the requests specifically made to a single presence identified by a URI like this: `http://{servername}/presences/{presence_id}`. The most significant methods are presented in Table 3.4:

- The first method allows to obtain a specific presence resource in XML format.
- The second method allows to get a specific presence resource in HTML format.

- The third method allows to update a presence resource.
- The fourth method is used to delete a specific presence resource.

Table 3.4: Main methods of the PresenceResource class.

Java Method	REST Method	Returned or Expected Formats
`getPresenceXML()`	`@GET`	`@Produces({MediaType.APPLICATION_XML}` `MediaType.APPLICATION_JSON`
`getPresenceHTML()`	`@GET`	`@Produces({MediaType.TEXT_HTML})`
`putPresenceXML()`	`@PUT`	`@Consumes(MediaType.APPLICATION_XML)`
`deletePresence`	`@DELETE`	

3.4.2 Package bean

In this package we implemented the Call and Presence classes, together with Observable interface which both implement, as shown in Figure ??. The two classes are very important because they represent the structure of the two main project resources. The attributes of these classes represent the call and presence resource fields already seen in Chapter 2. The only new attribute within the Call class is *actions*. This does not describe a resource property but is used to send to the client the actions that can be taken on the same resource. The set and get methods are essential. The first allow the creation of a Call class instance starting from its encoding in XML. The second is used to create the XML representation from a Call object. These two actions are performed automatically through the use of the Jersey and JAXB libraries. In the Call class there are some methods used to compile the *actions* attribute: checkNextStatus and hasNextStatus, they are used to check if the new value of the status field to give is correct. In particular, these two methods are used to verify that in the state machine representation (Figure 2.10) there is a transition that leads from the current state to the new one. Observable interface indicates the only characteristic required for a resource so that it is observable and therefore subscriptions associated to it can exist.

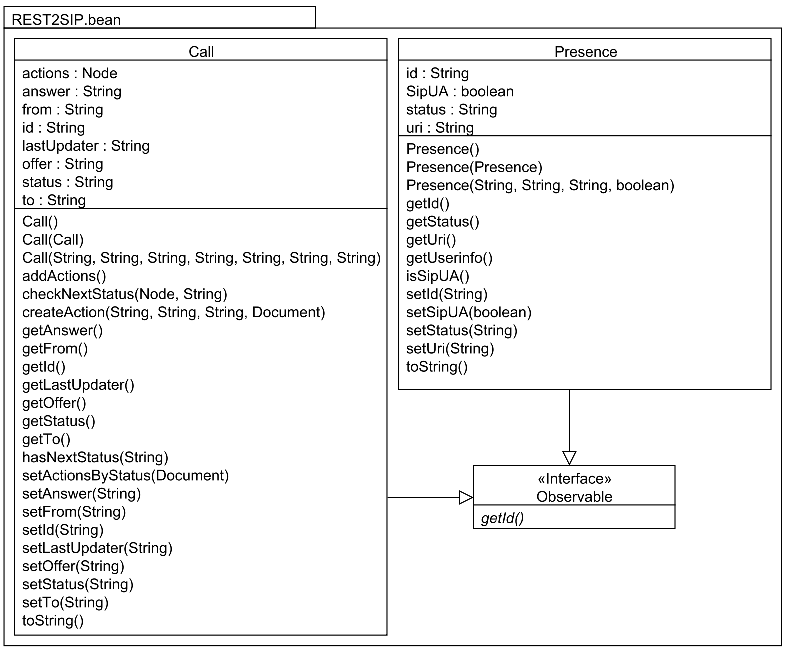


Figure 3.18: Package Bean.

3.4.3 Package servlet

This package contains the servlets used in our project and other two supporting classes. The wsServlet class is used to accept communications via WebSocket protocol to handle notifications to the client. The purpose is to establish a channel between client and server and provide the used stream to send data to the client via a OutStreamInboud instance. This class extends StreamInbound class and therefore implements the methods shown in Figure 3.19. These methods are used to handle any data sent from the client to the server via WebSocket. In our case we use only the connection created for communication from server to client, so we don't create code for this purpose. The REST2SIPServlet has a more complex role because is responsible for receiving all request and response from the SIP client. This servlet defines the methods invoked depending on the received message type. For instance, the doRegister method is called for the REGISTER message management, doInvite for INVITE message management. A REST client is implemented within some methods, through classes provided by Jersey, which makes use of methods proposed by the service to translate SIP messages in

HTTP requests. For instance, the sending of a SIP REGISTER message, depending on the header values, corresponds to the creation, modification or deletion of a resource Presence via POST, PUT or DELETE HTTP method, respectively.

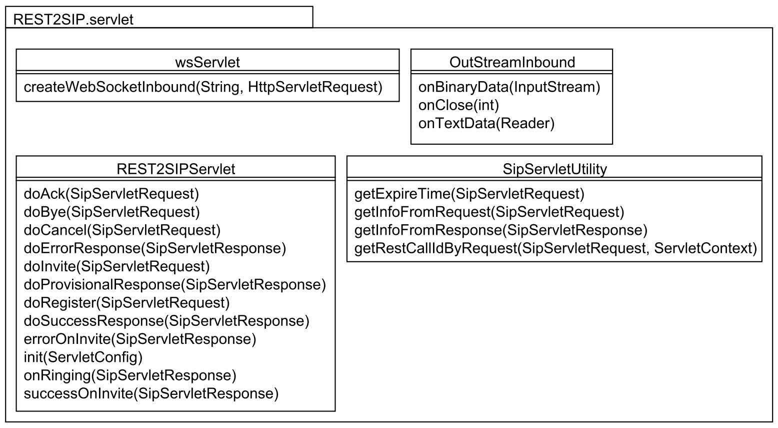

REST2SIP.servlet
wsServlet
createWebSocketInbound(String, HttpServletRequest)
OutStreamInbound
onBinaryData(InputStream)
onClose(int)
onTextData(Reader)
REST2SIPServlet
doAck(SipServletRequest)
doBye(SipServletRequest)
doCancel(SipServletRequest)
doErrorResponse(SipServletResponse)
doInvite(SipServletRequest)
doProvisionalResponse(SipServletResponse)
doRegister(SipServletRequest)
doSuccessResponse(SipServletResponse)
errorOnInvite(SipServletResponse)
init(ServletConfig)
onRinging(SipServletResponse)
successOnInvite(SipServletResponse)
SipServletUtility
getExpireTime(SipServletRequest)
getInfoFromRequest(SipServletRequest)
getInfoFromResponse(SipServletResponse)
getRestCallIdByRequest(SipServletRequest, ServletContext)

Figure 3.19: Package Servlet.

3.4.4 Package sip

This package presents only the BrowserSipCallHandler class that implements the Observer interface presented in section 3.4.7 and plays the important role of sending messages to the SIP client, as shown in Figure 3.20. The creation of an instance is closely linked to a call involving a SIP client and can be considered a subscription, as can be verified by observing Figure 3.21. Whenever a change is made to the resource Call, whose Id matches the CallID value, the sendNotification method is called. This checks if there are messages to be sent by considering the state in which the call is and the role played by the SIP client, caller or callee. The sequence diagram of the notification istance is shown in Figure 3.22.

3.4.5 Package storage

This package contains classes that instantiate objects responsible for resource storing, PresenceStore and CallStore, and also Call resource observers (Figure 3.23). All three elements implement the Singleton pattern which as-

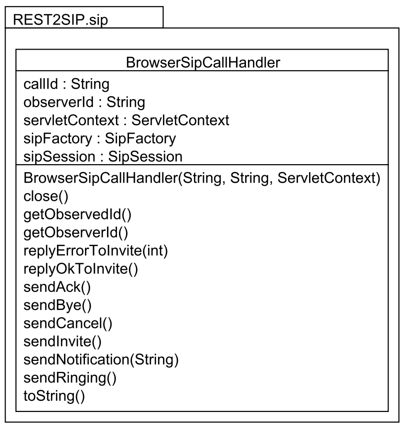


Figure 3.20: Package Sip.

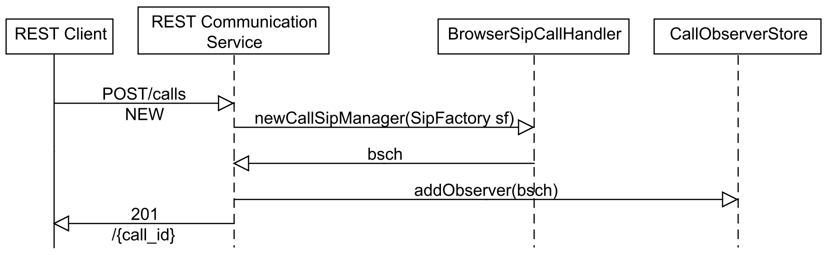


Figure 3.21: Subscription of a BrowserSipCallHandler instance to a call with SIP UA.

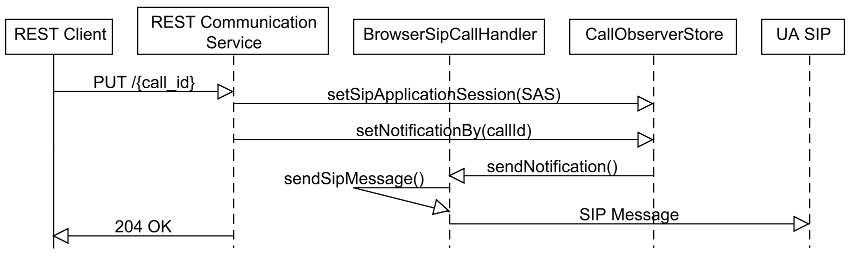


Figure 3.22: Notification of a BrowserSipCallHandler instance to a call with SIP UA.

sures for each storage the existence of one instance that can be recovered easily. The store attribute is in PresenceStore and CallStore and is a Map that connects its resource to each Id. Among the methods we have getFreeId(), used to find an available Id by which to identify a possible resource to create. In addition, in PresenceStore there is a getPresenceBy method, which permits to recover the resource Id starting from the *uri* field value. This avoids to create a new Presence resource if it already exists for the received URI. The class CallObserverStore presents methods for the subscription managing to the various resources. Among the methods there are the ones to recover the Observer depending on the observed object, getObserversByObservedId and getObserversByObserverId, in addition to those to record and delete one or more Observer. We define the sendNotificationBy method, which is called for the resource updating and, in turn, calls the sendNotification method on Observer instances registered to a particular call.

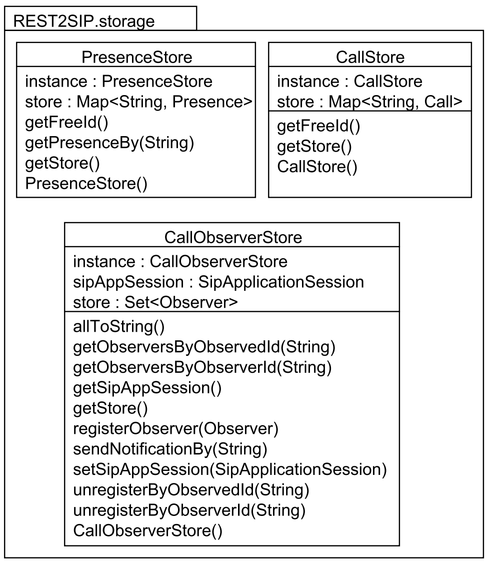

REST2SIP.storage
PresenceStore
instance : PresenceStore
store : Map<String, Presence>
getFreeId()
getPresenceBy(String)
getStore()
PresenceStore()
CallStore
instance : CallStore
store : Map<String, Call>
getFreeId()
getStore()
CallStore()
CallObserverStore
instance : CallObserverStore
sipAppSession : SipApplicationSession
store : Set<Observer>
allToString()
getObserversByObservedId(String)
getObserversByObserverId(String)
getSipAppSession()
getStore()
registerObserver(Observer)
sendNotificationBy(String)
setSipAppSession(SipApplicationSession)
unregisterByObservedId(String)
unregisterByObserverId(String)
CallObserverStore()

Figure 3.23: Package Storage.

3.4.6 Package util

The classes in this package are for ”utility”, namely they perform auxiliary functions for other classes. The RESTapi class presents methods that allow the SIP Servlet REST2SIPservlet to execute request to the REST service.

The other two TypeProduceCall and TypeProducePresence classes provide methods for creating XML content or other format that is used by RESTapi class methods (Figure 3.24).

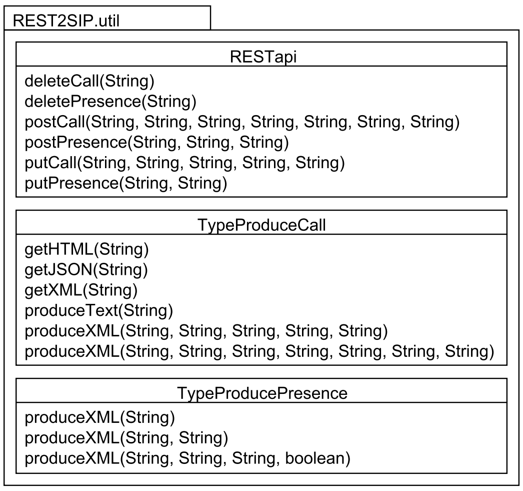

REST2SIP.util
RESTapi
deleteCall(String)
deletePresence(String)
postCall(String, String, String, String, String, String, String)
postPresence(String, String, String)
putCall(String, String, String, String, String)
putPresence(String, String)
TypeProduceCall
getHTML(String)
getJSON(String)
getXML(String)
produceText(String)
produceXML(String, String, String, String, String)
produceXML(String, String, String, String, String, String, String)
TypeProducePresence
produceXML(String)
produceXML(String, String)
produceXML(String, String, String, boolean)

Figure 3.24: Package Util.

3.4.7 Package async

This package contains the classes used to manage individual subscriptions: CallACResponder, CallACWriter and CallWSWriter. All three classes implement the same Observer interface, as shown in Figure 3.25. The Observer interface presents four important methods that must be implemented (toString is used for debugging reason):

- *sendNotification*, is used to send the change notification to the registered client;
- *getObservedId*, returns the identifier of the observed resource;
- *getObserverId*, returns the client identifier observing the resource, typically the Id of its presence;
- *close*, is called before the subscription cancellation and allows to close the stream with the client.

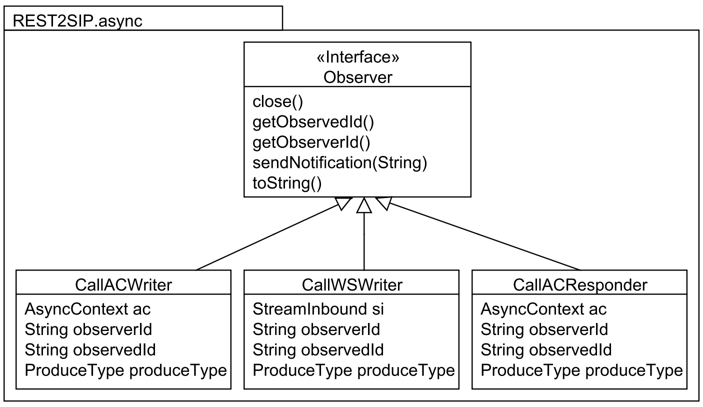


Figure 3.25: Package Async.

The Observer interface is also implemented by the BrowserSipCallHandler class, already previously described. All three the classes have the same set of attributes:

- *AsyncContext* or *StreamInbound*, is used to convey messages to the client;
- *observerId*, indicates the string that identifies the Presence resource of its subscriber;
- *observedId*, indicatesthe string that identifies the subscribed resource;
- *ProduceType*, indicates the format type with which we want to receive the notification.

CallACResponder

This class allows to make a subscription following the "long polling" principles, already defined in the chapter 2. In order to make this subscription we used a GET method with suffix "/live", where a parameter specifies the subscription type (Figure 3.26). At this point, a AsyncContext is created and its stream is used to send the response to the client. When a notification must be sent, the sendNotification method is called to send its data to the client, closing the asynchronous context and response. At this point the CallACResponder object, and therefore the subscription representing, is canceled and the client has to again send a subscription request to continue receiving notifications (Figure 3.27).

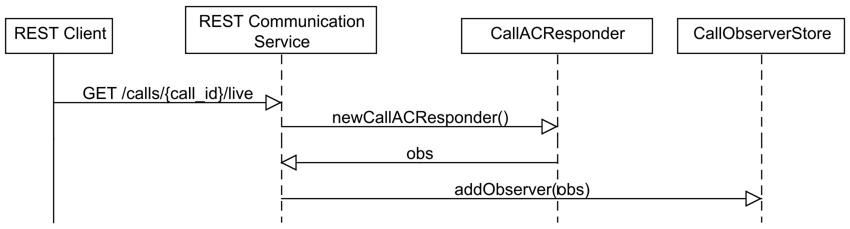


Figure 3.26: Subscription of a CallACResponder object.

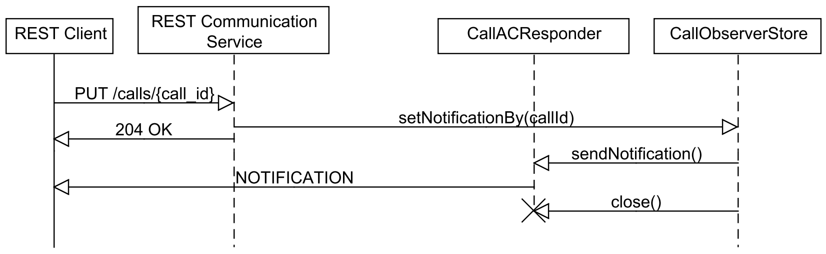


Figure 3.27: Notification of a CallACResponder object.

CallACWriter

This class implements the HTTP streaming technique, already defined in the chapter 2, with the aid of the asynchronous context that improves its performance. This solution, applied to the call context between a web client and a SIP client, is also presented in [Dureulle, 2008]. Similar to the CallACResponder case, in order to subscribe a REST client must send a GET request with suffix ”/live” with a parameter that specifies the type, Figure 3.28. Unlike the previous case, for each notification the subscription should not be made, Figure 3.29. The sent notifications queue one after the other in partial response that the client receives. The AsyncContext is closed once the subscription is canceled.

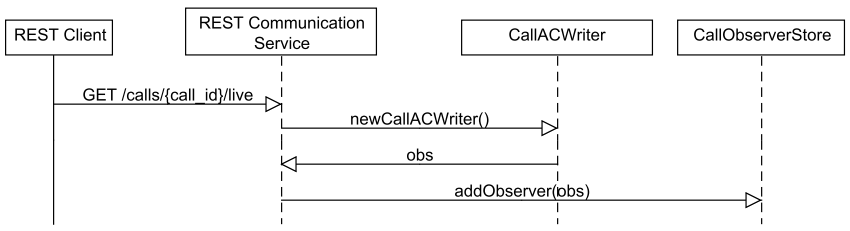


Figure 3.28: Subscription of a CallACWriter object.

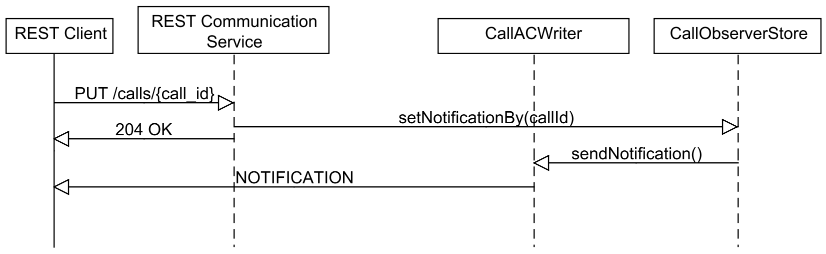


Figure 3.29: Notification of a CallACWriter object.

CallWSWriter

This class uses the WebSocket protocol to obtain a channel between client and server with which the first can notify the updates to the subscribed resource. Its behavior is very similar to that CallACWriter but differs in some essential points:

- subscription modality, involves a specific servlet and not the servlet that deals with the REST service;
- response entity absence, as WebSocket is not a request/response protocol;
- the sent notifications don't queue but are retrieved from the stream.

Figure 3.30 shows the sequence diagram of the subscription. The GET message is used to make the upgrade to the WebSocket protocol. During its processing the CallWSWriter instance is created and added to the store. Figure 3.31 shows the notification process that is similar to the CallACWriter case. The subscription cancellation requires the connection deletion and deleting the CallWSWriter object.

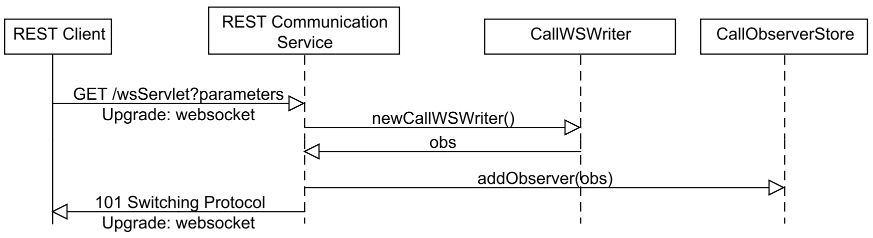


Figure 3.30: Subscription of a CallWSWriter object.

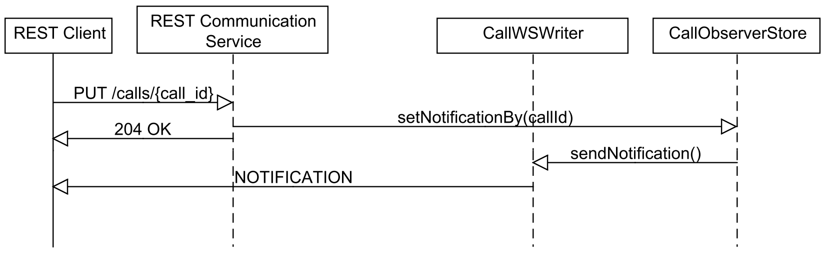


Figure 3.31: Notification of a CallWSWriter object.

3.5 Client-side Script

The web application consists of several JavaScript files. In addition to those specially designed for the application XMLWriter.js and jquery.js have been used. The first allows to create easily XML files through the use of special functions [Hickson, 2011]. The second, jQuery, is a popular JavaScript library that simplifies the use of HTML pages, event handling, animation and interaction between Ajax and Web applications. With jQuery, we can change the approach for writing JavaScript code [AA.VV., 2005]. In the following sub-paragraphs we describe scripts created for this Web application.

3.5.1 interfaceOperationAll.js

This script contains the main functions concerning the management of the graphical interface and events. Some functions related to the arrival of asynchronous events and callbacks are reported below:

- *onSDPReady*, is invoked by the caller or callee when these know the parameters to be exchanged to establish a media session (SDP). The method returns the description within a variable. If the caller invokes this callback, then this event corresponds to the sending of a PUT request that changes the resource state to CALLING Call and inserts a value for the "offer" field. If instead the callee invokes this callback, this event corresponds to send a PUT request that changes the call status to ANSWERED and enter a value for the "'answer" field.

- *onPeerConnection*, is invoked when the connection is established between the two peers. Upon the occurrence of this event the remote stream is started within a video tag.

Functions related to user input are:

- *setPresence*, method invoked when the user logs in. It creates a presence resource by sending a POST request. When resource creation is confirmed, the user makes the recording to incoming calls as already described in chapter 2.
- *startCall*, method invoked when the user wants to initiate a call. After retrieving the called party the call resource is created by sending a POST request. Upon the creation confirmation the user subscribes to the call, to receive future update notifications, and initiated the creation of the "offer" SDP.
- *cancelCall*, method invoked by the caller when it decides to stop call establishment request. This method sets the resource status to CANCEL by sending a PUT request.
- *answerCall*, method invoked by the callee to answer to the call. Once the user has subscribed to the incoming call, the creation of the "answer" SDP is initiated and is followed by the invocation of the onSDPReady callback.
- *busyCall*, method invoked by the callee if the user decides to report the caller that does not want to establish the session. The call is put on BUSY state through a PUT request.
- *closeCall*, method invoked by one of the two parties to end the call. The Web application sends a PUT request that sets the resource to CLOSED state and ends the connection previously established between the peers.

- *execActions*, this method allows to parse the ”actions” field returned together to the call resource (as already described in chapter 2). In this way we can identify actions that can be performed either by the caller that callee.

3.5.2 presenceAPI.js

The presenceAPI script allows sending request to the Presence service. The methods are:

- *getAllPresence*, allows to recover all the Presence sending a GET request to the URI: `http://{servername}/presences`.
- *getPresence*, allows to retrieve a specific presence by sending a GET request to the URI: `http://{servername}/presences/{presence_id}`.
- *postPresence*, allows to create a Presence resource by sending a POST request to the URI: `http://{servername}/presences`.
- *putPresence*, allows to update a specific presence by sending a PUT request to the URI: `http://{servername}/presences/{presence_id}`.
- *deletePresence*, allows to delete a specific presence by sending a DELETE request to the URI: `http://{servername}/presences/{presence_id}`.

3.5.3 callAPI.js e Call.js

The callAPI script allows sending request to the call service. The methods are:

- *getCall*, allows to retrieve a specific call by sending a GET request to the URI: `http://{servername}/calls/{call_id}`.
- *postCall*, allows to create a call resource by sending a POST request to the URI: `http://{servername}/calls`.
- *putCall*, allows to update a specific call by sending a PUT request to the URI: `http://{servername}/calls/{call_id}`.

The Call script makes easy to access the information in the resource returned by the server. Once the resource is returned in XML format, this description is used to create an Call object instance to retrieve the field values of the resource by the get methods.

3.5.4 registrationAll.js

This script collects all functions relating to the subscription both of a call is incoming calls. The functions are:

- *registerToCall*, is used to subscribe to a call, or more generally to incoming call. This function takes as input:
  - *asyncType*, indicates the method used to get the notification: WebSocket, AsyncContextWriter or AsyncContextResponder;
  - *CallID*, indicates the call Id to which to subscribe or the '*' character to subscribe to all incoming calls;
  - *produceType*, indicates the format in which we want to receive the resource upon receipt of a notice (e.g., XML, HTML, etc.);
  - *presenceId*, indicates the presence Id of the subscriber;
  - *onRegistered*, indicates the callback function that must be called at the time of registration;
  - *onNotification*, indicates the callback function that is invoked upon receipt of a notification. Each time this is called, receives the latest version of the resource in the format specified by produceType parameter.
- *onIncomingCallRegistered*, the callback function invoked for each subscription to incoming calls.
- *onIncomingCallNotification*, the callback function used with the notification of an incoming call. In the specific case where the user is already engaged in another call, this callback report it to the caller by sending a PUT request that sets the value of the status field to BUSY.
- *onCallRegistered*, the callback function used when the subscription registration to a specific call is happened.

- *onCallNotification*, the callback function used to notify the call state change to which the subscription was made. This function invokes an another function which analyzes the back resource to take lately the appropriate actions, as already described in chapter 2.

Subscription methodologies

There are three ways to subscribe to calls. In 3.4.7 we described server-side implementation differences, in this section client-side implementation differences. As first case we analyzed the "WebSocketWriter" method for which we create the connection passing to the WebSocket class constructor the URL: `"Ws://{servername}:<port>/wsServlet?<parameters>"` where:

- *servername*, indicates the URL or IP address of the server providing the service;
- *port*, indicates the port to which the server provides the service;
- *parameters*, indicates the parameters to be passed to the server already described above: asyncType, CallID, produceType and presenceId.

Once the WebSocket instance is created, we define the functions relating to certain events:

- *OnOpen*, the onCallRegistered or onIncomingCallRegistered function is assigned;
- *onMessage*, the onCallNotification or onIncomingCallNotification function is assigned.

In the "AsyncContextWriter" case, we must build an HTTP request through a XMLHttpRequest instance. The URL will be sent to the interest resource to which the "/live?<parameters>" suffix is added. The parameters to be passed to the server are always those already described in the "WebSocketWriter" case. We define then the onreadystatechange event that identifies every time the behavior of state change in the request. For every change we evaluate the readyState attribute value which can be:

- 0 = uninitialized;
- 1 = open;

- 2 = request sent;
- 3 = response reception;
- 4 = response received.

The state 3 indicates the presence of a new information sent by the server, therefore, we assign the onCallNotification or onIncomingCallNotification function at this case. At this point we can send the request. Lastly we described the "AsyncContextResponder" case. This method is similar to the above. Only difference is the state in which the onCallNotification or onIncomingCallNotification function is called is 4. This method expects at each notification the request is terminated. This behavior leads the Web-application having to subscribe to every notification.

3.5.5 WebRTC.js

This script deals with the acquisition of audio and video streams and the creation of peer-to-peer connection between caller and callee. The acquisition of audio and video streams is possible by the use of the API getUserMedia included in WebRTC standard [Bergkvist et al., 2014]. After the user has given consent on the web page, the application will be able to access audio and video capture devices of the machine on which the browser is running. The peer-to-peer connection allows the channel creation between two clients through the use of RTCPeerConnection objects. Once the RTCPeerConnection instance got the audio and video stream, the SDP "offer" is created with the which the "offer" field of the call resource will be updated. The callee reads the "offer" value and creates, as for the caller, the SDP "answer", namely the descriptor related to its audio and video streams. This is used to update the "answer" field of the call resource. When the two RTCPeerConnection instances have set the descriptor value of the remote client, then can finally establish the communication.

Chapter 4

Web application Functioning

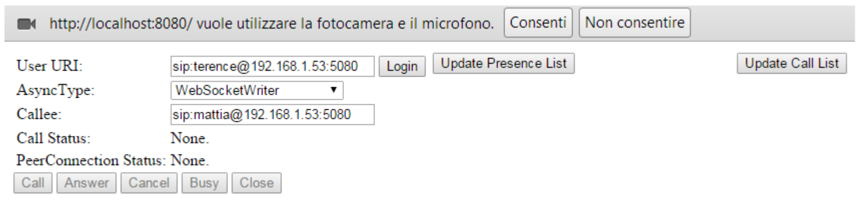
This chapter describes the call service web application functioning and related functional test. In paragraph 4.1 we describe the web application interface and its functioning. In paragraph 4.2 we describe the functional test related to main reference scenarios defined in the previous chapter.

4.1 Web application Interface

When an user inserts the call service URL, the web application interface is returned (Figure 4.1) that requires to the user the authorization to use the camera and microphone. This is due to the using of the API getUserMedia. Once the user gave its consent, he can login. To do this, the user must enter a valid SIP URI, choose a method for managing subscriptions and then press the Login button that invokes the setPresence function. At this point, the web application enables the ability to initiate a call specifying the callee SIP URI.

The interface presents other fields, as shown in Figure 4.2:

- *Call Status*, indicates the status value of the call resource. Moreover informs the user if the *offer* and *answer* fields have been set and the last user who updated the call resource.



http://localhost:8080/ vuole utilizzare la fotocamera e il microfono. Consenti Non consentire

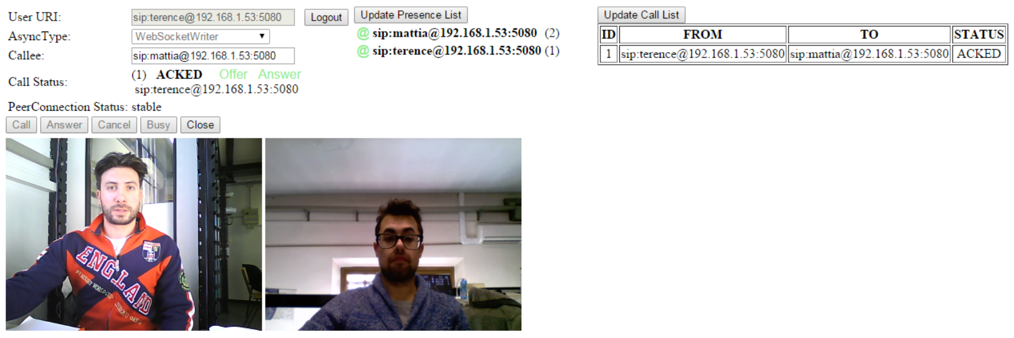
User URI:	sip:terence@192.168.1.53:5080
AsyncType:	WebSocketWriter
Callee:	sip:mattia@192.168.1.53:5080
Call Status:	None.
PeerConnection Status:	None.

Figure 4.1: Web application Interface.

- *PeerConnection Status*, indicates the peer-to-peer connection status.
- *Presence Update List*, indicates all the Presence resources created and then all users currently connected to the service.

Below the interface contains the buttons which recall the functions already seen in chapter 3 for interacting with the web application.

- *Call*, allows a user to initiate the call by invoking the startCall function;
- *Answer*, allows the callee to answer an incoming call by invoking the answerCall function;
- *Cancel*, allows the caller to cancel the call request by invoking the cancelCall function;
- *Busy*, allows the callee to not answer to the call by invoking the busy-Call function;
- *Close*, allows the caller and callee to close properly an established call by invoking the closeCall function.



User URI:	sip:terence@192.168.1.53:5080
AsyncType:	WebSocketWriter
Callee:	sip:mattia@192.168.1.53:5080
Call Status:	(1) ACKED Offer Answer sip:terence@192.168.1.53:5080
PeerConnection Status:	stable

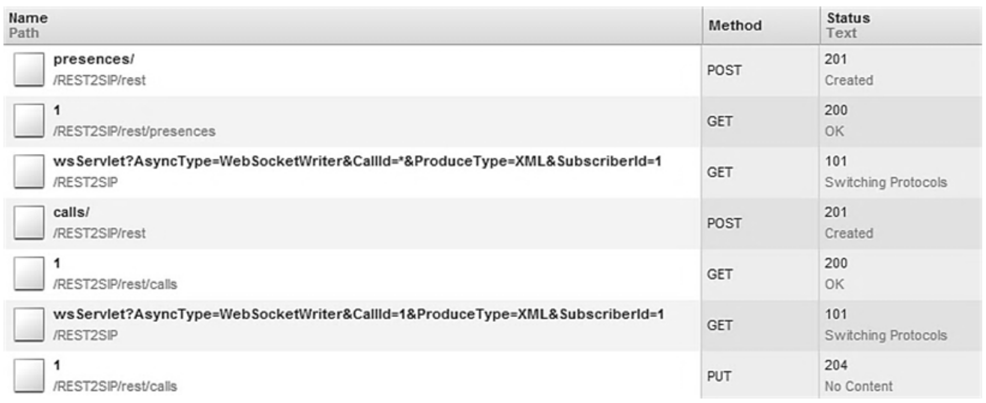
@ sip:mattia@192.168.1.53:5080 (2)
@ sip:terence@192.168.1.53:5080 (1)

ID	FROM	TO	STATUS
1	sip:terence@192.168.1.53:5080	sip:mattia@192.168.1.53:5080	ACKED

Figure 4.2: Call service web application functioning.

Once the call is initiated, the buttons described above and the camera activate. Subsequently, when the callee accepts the call, appears also the video stream from the callee user. Symmetrically, when the callee accepts the call, two videos will be shown to him: one relating to its camera and one to the stream coming from the caller user. Figure 4.2 shows in chronological order the flow of HTTP requests sent by the Web-application and monitored by Google Chrome:

- the creation of the presence resource and the response 201 that contains the Id of the newly created resource;
- the recovery of the presence resource due to the Id knowledge;
- the subscription to incoming calls through the use of WebSocket and then the protocol switching evidenced by the response 101;
- the creation of the call resource and the response 201 that contains the Id of the newly created resource;
- the recovery of the call resource based on the knowledge of the Id;
- the subscription to the created call through the use of WebSocket and then protocol switching evidenced by the response 101;
- the updating of the call resource to the state CALLING via a PUT request.



Name	Method	Status
Path		Text
presences/ /REST2SIP/rest	POST	201 Created
1 /REST2SIP/rest/presences	GET	200 OK
wsServlet?AsyncType=WebSocketWriter&CallId=*&ProduceType=XML&SubscriberId=1 /REST2SIP	GET	101 Switching Protocols
calls/ /REST2SIP/rest	POST	201 Created
1 /REST2SIP/rest/calls	GET	200 OK
wsServlet?AsyncType=WebSocketWriter&CallId=1&ProduceType=XML&SubscriberId=1 /REST2SIP	GET	101 Switching Protocols
1 /REST2SIP/rest/calls	PUT	204 No Content

Figure 4.3: Flow of HTTP requests sent by the Web application.

4.2 Functional Test

This section describes the functional test related to main reference scenarios defined in the previous chapter. In order to test these scenarios we chose the following testbed environment:

- Apache Tomcat version 7.0.29, the servlet container used to make available the call service.
- Google Chrome version 25.0.1364.172 m, the browser for accessing to the Web-application that implements the REST client. We chose this browser because it implements the WebRTC specifications without special settings.
- Talk Express version 4.28, used as a SIP client. This choice was made for the simplicity of software setting and implementation absence of SIP protocol extensions such as SIMPLE or other, usually very common in other clients (X-lite, Blink, Jitsi, ...). Such extensions would lead to the continuous re-sending of SIP messages to the call service due to the lack of response.

The results are the same regardless of the method used to subscribe to asynchronous notifications sent from the server to the client. The test verified successfully the following scenarios:

- the registration and deregistration to the Presence service by a REST client;
- the registration and deregistration to the Presence service by a SIP client;
- Successful call setup and closure between two REST clients;
- Unsuccessful call due to busy callee or cancel request by the caller.

The call establishment between a REST client and a SIP User Agent is not possible. As regards the signaling phase we don't encountered problems. The busy-answer by the callee and the call cancellation by the caller are managed correctly. The problem occurs in the case the callee decides to accept the call. In this case, we found an incompatibility between the codes used to audio/video level between the WebRTC standard and the SIP clients, for which the two users can not communicate. This same situation was found by other developers who have attempted to establish a call using the WebRTC API and a SIP client. In a near future probably the WebRTC API capabilities will be increased and will therefore allow the channel media establishment necessary to the communication.

Chapter 5

Performance Evaluation

Once we verified the RESTful service functioning, we performed a set of test iterations to evaluate the performance of our prototype when is used by multiple users. This chapter describes the performed tests and the obtained results to identify the differences between the three subscription methods to the asynchronous notifications. In Section 5.1 we describe the environment in which the tests were performed and the way used to simulate a larger number of clients accessing simultaneously. In section 5.2 we show the test and the obtained results related to the time measurement between the various events and the use of computer resources by our service.

5.1 Testbed Environment

We considered as test scenario the canceled-call case by the caller between a REST client and a SIP User Agent , as already seen in Figure 3.13. In order to simulate a configurable number of REST clients that request a call setup, we developed a web application allowing to configure the number of calls to be initiated and the time delay between two consecutive calls. The following test results have been obtained configuring the web application for simulating the initiation of 100 consecutive calls with 5 seconds of delay between two consecutive calls. The testbed environment included a single machine with a CPU Processor Intel Core i3 3217U 1.80 GHz, RAM 4 GB DDR3, hosting the web application prototype, a Google Chrome browser and a SIP client Express Talk (the software configuration is the same already seen in chapter 4). We chose to perform this experiment on a single machine to gather the results on delays due to processing tasks minimizing the network delays.

5.2 Performance Test

We performed a set of test iterations in order to evaluate the performance of the implemented prototype. A first experiment was aimed for evaluating the performance in terms of time delay in REST-to-REST and REST-to-SIP call scenarios.

We used the following metrics:

- the *call setup delay*, it defines the time elapsing between the call setup request (POST HTTP message) and the reception of the final response (ANSWERED call notification);
- the *subscription delay*, it defines the time between the delivery of the subscription request and the establishment of the notification channel;
- the *notification delay*, it defines the time between the occurrence of an event (e.g., the reception of a PUT request in order of change the resource state) and the reception of the corresponding notification action by the subscribed client.

The obtained results are presented hereafter. Table 5.1 compares the call setup delay delays in REST-to-REST and REST-to-SIP call scenarios. The call setup between two web browsers required approximately 50 ms, where approximately 30 ms were due to the initial phase (i.e., PUT and POST invocation). A call setup between a web browser and a SIP User Agent required about 110 ms, where approximately 60 ms were required for processing an incoming SIP message and translating it into the corresponding REST invocation. Thus, the difference between the REST-to-REST and the REST-to-SIP scenarios is essentially due to the time needed for processing the incoming SIP messages and performing the corresponding REST invocation.

These results show how the average call setup delay in our prototype is comparable with analogous measures for call setup delay in SIP environ-

Table 5.1: Call Setup delays

REST Invocation delay (ms)	REST-to-REST Call Setup delay (ms)	SIP Message Processing (ms)	REST-to-SIP Call Setup delay (ms)
30	50	60	110

ments. For instance, the study by Kellokoski et al. [2010] reports an average call setup delay about 40 ms between two SIP User Agents. Moreover, the maximum call setup delay measured in our prototype is well below the acceptable limit about 8 seconds for the call setup delay in Web/Telecom convergent environments defined by the TS 186 008-2 standard [Vingarzan et al., 2007].

Table 5.2 compares the subscription and notification delays obtained by adopting the WebSocket, Long Polling and HTTP Streaming notification approaches. The subscription delay in the three cases has been measured in the following way:

- *WebSocket* case, the delay is the client-side measured time between the delivery of the request for activating the WebSocket channel and the reception of the HTTP 101 response message.
- *HTTP Streaming* case, the delay is the time interval between the subscription request and the reception of the first HTTP response chunk.
- *Long Polling* case, the delay is the time interval between the subscription request and the reception of the HTTP response.

As shown in Table 5.2 the subscription delay with WebSocket is 7 ms on average, while the delay with HTTP Streaming and Long Polling is around 13 ms. The notification delay is around 12 ms for all three approaches.

Table 5.2: Subscription and Notification delays

	Subscription delay (ms)	Notification delay (ms)
Web Socket	7	11
HTTP Streaming	13	12
Long Polling	14	13

A second experiment was aimed at measuring the consumption of resources in terms of CPU usage. In order to obtain more reliable measurements, the browser Google Chrome and the SIP client Express Talk were located on a second machine. The two machines were connected via a 100 Mbps Ethernet/LAN. In the machine which hosts the Tomcat Application Container and the web application we used JProfiler, which is a JVM profiler that offers CPU profiling capabilities.

As in the previous experiment, we ran this experiment with a load scenario provided by the test web application configured with 100 users. First, all users send a presence registration request to the server, as we already seen in Figure 3.2, and then they initiate a call to a SIP User Agent, one after the other with a time interval of 5 seconds between two consecutive calls.

The Figures 5.1, 5.2 and 5.3 show the CPU load for the WebSocket, Long Polling and HTTP Streaming approaches, respectively. In all three cases, the CPU load has some peaks in the first time instants. This is due to the creation of presence resources which occur in the initial phase of our experiment. Then, the following peaks are due to the POST and PUT requests processing for the call setup, recurring at intervals of approximately 5 seconds.

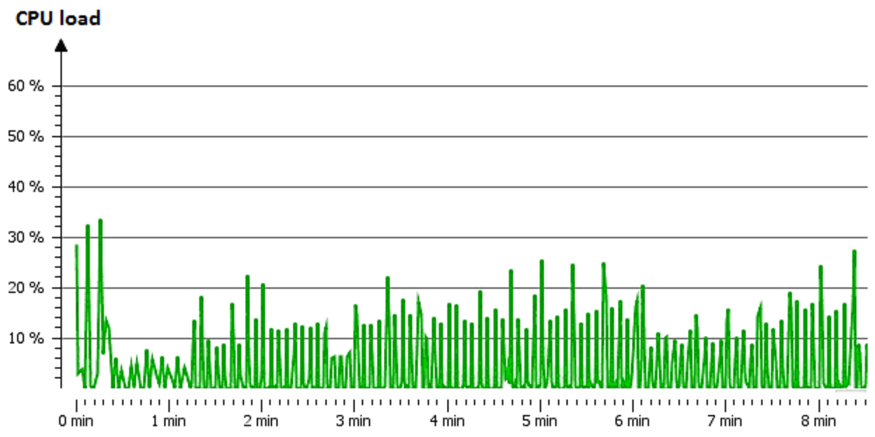


Figure 5.1: CPU usage with the Long Polling notification approach.

In the case of Long Polling (Fig. 5.1) and HTTP Streaming (Fig. 5.2), these peaks increase up to a CPU usage of 20% and then decrease to a level close to zero, while in the case of WebSocket (Fig. 5.3), the application shows a CPU usage with peaks up to 10% with a minimum CPU usage that never drops below 5%.

At actual state, the available studies on WebSocket focused on the network latency and throughput in reference scenarios characterized by continuously streamed data, as in the study proposed by Pimentel and Nickerson

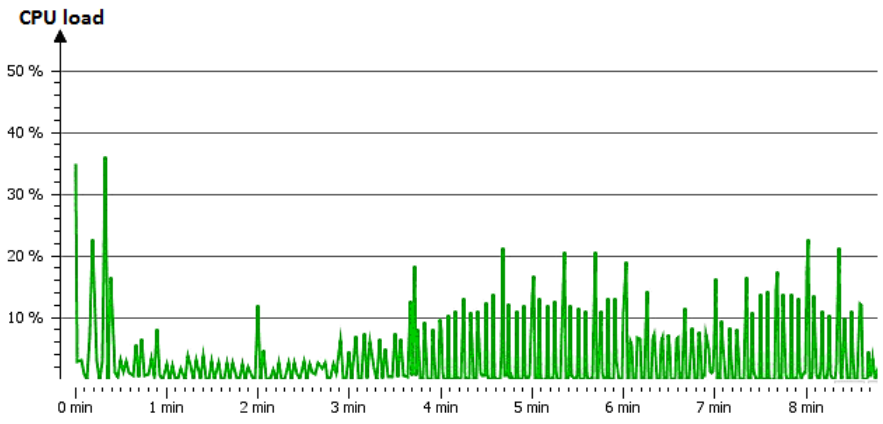


Figure 5.2: CPU usage with the HTTP Streaming notification approach.

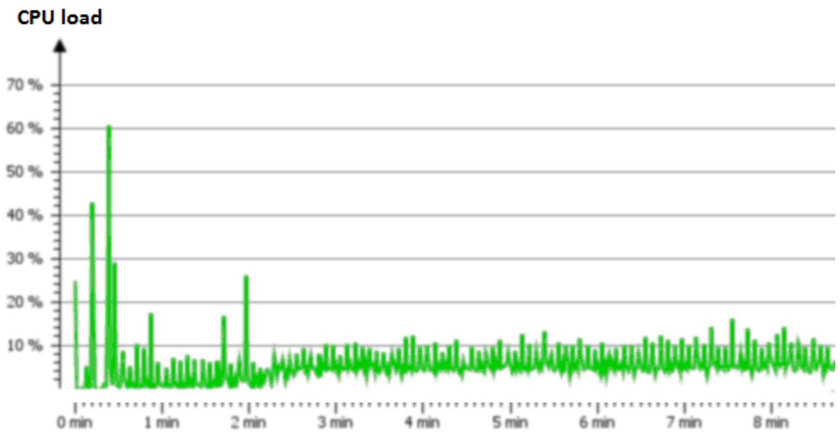


Figure 5.3: CPU usage with the WebSocket notification approach.

[2012]. Thus, it is not straightforward to compare our results with these studies.

More accurate performance analysis would be worthwhile for evaluating delays and consumption of resources in more complex workload scenarios, which we are planning to make in future works.

Conclusions

In this thesis work, we have proposed an approach for the design and implementation of a set of Web APIs based on REST principles for providing real-time communication services towards the convergence of web and Telecommunications domains.

By leveraging a resource-oriented design methodology, we defined a set of REST APIs and subsequently modeled the call resource behavior through a Finite State Machine (FSM) representation. We specified FSM states, transitions and actions with the requirement of REST/SIP interworking in mind. We adopted a FSM representation (as UML State Diagrams) in order to use the analysis and exploration capabilities offered by the UMC tool to model the RESTful service as a set of communicating state machines and simulate their behavior and interworking with client components (i.e., a REST client and a SIP User Agent).

Finally, by leveraging these design specifications, we implemented a REST-based communication service that can be invoked by any recent browser without requiring any additional code download. We took into account a scenario for a call session setup between two REST clients and a REST client and a SIP User Agent, respectively, by implementing three mechanism for handling asynchronous notifications. The WebRTC standard is been used to create peer-to-peer video and audio communication. Finally, we evaluated the functional correctness as well as the performance of this prototype implementation in terms of time delay (i.e., call setup, subscription and notification delay) and resource usage (i.e., CPU load). The functional test showed that implemented REST APIs allow to use the WebRTC standard to create audio and video communications between two users equipped with a browser.

However we verified the impossibility, with the current implementation of the WebRTC specification, to establish a call between a web browser and a SIP user agent due to an incompatibility between the codes used to audio/video level between the WebRTC standard and the SIP clients.This problem has been reported by other developers and is probably due to the youth of the WebRTC technology.

According to REST principles adopted in this thesis, the implemented REST APIs support stateless interaction constraint, namely every request from the client to the server contains all the information required for serving the request. The set of resources exposed through a uniform interface based on HTTP methods can be extended for offering other similar services (e.g., instant messaging and video conference service).

Moreover the implemented prototype adheres to the HATEOAS constraint, namely at each interaction step the client is provided with the options that are permitted at that point. In particular, at each transition the server provides to the client the instructions on the possible next steps. Therefore, the dynamic behavior of the caller and callee users is embedded in the client-side application logic, but the possible next transitions are provided by the server. This causes loose coupling between client-side and server-side implementations, while guarantees that the client behaves coherently with the state machine of the call resource. This advantage is accentuated if we compare this approach with legacy SIP User Agents, which implement SIP client and server state machines defined in the SIP specifications.

Real-time communication services require the delivery of asynchronous notifications. To fulfill this issue, that is not clearly handled in the REST architectural style, we implemented and compared through performance testing three alternative solutions: the first solution is based on the WebSocket protocol, the second and third one on Long Polling and HTTP Streaming technique, respectively.

Future work includes to extend the proposed approach and related implementation to expose more complex services (e.g., a video conference service). Moreover, we will investigate solutions for REST service publication and discovery in order to ease the dynamic specification and realization of Web and Telecommunication composite services. The use of the state machine formalism allowed us to exploit the analysis and exploration capabilities offered by the UMC tool to evaluate the compliance of the implemented prototype behavior with the communicating state machine model. By relying on these results, an another future work could be to extend this study towards formal specification of RESTful services:

1. We will define a set of formal properties that express desirable attributes of our model, i.e. general properties (e.g., deadlock absence), and specific properties of RESTful systems (e.g., resource connectedness) [Chakrabarti and Rodriquez, 2010].

2. We will use the model-checking capabilities offered by the UMC tool for automated property verification.

UMC is not currently equipped with other typical capabilities of model-based design tools, such as the generation of automatic code and automatic test from the model. Regarding to UML tool, these two capabilities can be found in the commercial IBM Rhapsody tool, which on the other hand does not provide model checking capabilities) [IBM Rhapsody, 2013]. The combined use of UMC and Rhapsody tools could be used to further explore the possibilities offered by both towards the rigorous design of web-based communication signaling and interworking.

Bibliography

3GPP. Open Service Access (OSA); Parlay X web services; Part 1: Common. 3GPP TS 29.199-01, 2009.

AA.VV. Writing XML using JavaScript, 2005. URL `http://www.codeproject.com/Articles/12504/Writing-XML-using-JavaScript`.

F. AlShahwan and K. Moessner. Providing SOAP Web Services and RESTful Web Services from Mobile Hosts. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 174–179, 2010.

H. Alvestrand. Real Time Protocols for Browser-based Applications. Internet-Draft, IETF, September 2013. URL `http://tools.ietf.org/html/draft-ietf-rtcweb-overview-08`.

Alessandro Amirante, Tobia Castaldi, Lorenzo Miniero, and Simon Pietro Romano. On the seamless interaction between webRTC browsers and SIP-based conferencing systems. *Communications Magazine, IEEE*, 51(4):42–47, 2013.

M. Barnes, C. Boulton, S. Romano, and H. Schulzrinne. Centralized Conferencing Manipulation Protocol. RFC 6503, 2012. URL `http://tools.ietf.org/html/rfc6503`.

F. Belqasmi, R. Glitho, and Chunyan Fu. Restful web services for service provisioning in next-generation networks: a survey. *Communications Magazine, IEEE*, 49(12):66–73, 2011.

F. Belqasmi, J. Singh, S.Y. Bani Melhem, and R.H. Glitho. SOAP-Based vs. RESTful Web Services: A Case Study for Multimedia Conferencing. *Internet Computing, IEEE*, 16(4):54–63, 2012. ISSN 1089-7801. doi: 10.1109/MIC.2012.62.

A. Bergkvist, C. Jennings D. C. Burnett, and A. Narayanan. WebRTC 1.0: Real-time Communication Between Browsers. W3C Working Draft, W3C, July 2014. URL `http://www.w3.org/TR/webrtc/`.

Gregory Bond, Eric Cheung, Ioannis Fikouras, and Roman Levenshteyn. Unified telecom and web services composition: problem definition and future directions. In *Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*, IPTComm '09, pages 13:1–13:12, New York, NY, USA, 2009. ACM. doi: 10.1145/1595637.1595654. URL `http://doi.acm.org/10.1145/1595637.1595654`.

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation. Technical report, W3C, 2004. URL `http://www.w3.org/TR/2004/REC-xml-20040204`.

Michael Brenner and Musa Unmehopa. *The Open Mobile Alliance: Delivering Service Enablers for Next-Generation Applications*. John Wiley & Sons, Ltd, 2008. ISBN 9780470519905.

G. Camarillo and M.A. García-Martín. *The 3G IP Multimedia Subsystem: Merging the Internet and the Cellular Worlds*. John Wiley & Sons, May 2006.

Sujit Kumar Chakrabarti and Reswin Rodriquez. Connectedness Testing of RESTful Web-services. In *Proceedings of the 3rd India Software Engineering Conference*, ISEC '10, pages 143–152, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-922-0. doi: 10.1145/1730874.1730902. URL `http://doi.acm.org/10.1145/1730874.1730902`.

Wu Chou, Li Li, and Feng Liu. Web services for communication over IP. *Communications Magazine, IEEE*, 46(3):136–143, 2008. ISSN 0163-6804. doi: 10.1109/MCOM.2008.4463784.

Douglas Crockford. Introducing JSON, 2002. URL `http://www.json.org`.

Arman Danesh and Wes Tatters. *JavaScript 1.1 developer's guide*. Sams. net, 1996.

Carol Davids, Alan Johnston, Kundan Singh, Henry Sinnreich, and Wilhelm Wimmreuter. SIP APIs for voice and video communications on the web. In *Proceedings of the 5th International Conference on Principles, Systems and Applications of IP Telecommunications*, IPTcomm '11, pages 2:1–2:7, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0975-2. doi: 10.1145/2124436.2124439. URL `http://doi.acm.org/10.1145/2124436.2124439`.

J Dureulle. Mobicents communications platform. *JavaOne presentations*, 2008.

Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. ISBN 0132344823.

I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, December 2011. URL `http://tools.ietf.org/rfc/rfc6455.txt`.

Fielding. Paper tigers and hidden dragons, 2008. URL `http://roy.gbiv.com/untangled/2008/paper-tigers-and-hidden-dragons`.

R.T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures.* PhD thesis, Architectural Styles and the Design of Network-Based Software Architecture, 2000.

Chunyan Fu, F. Belqasmi, and R. Glitho. RESTful web services for bridging presence service across technologies and domains: an early feasibility prototype. *Communications Magazine, IEEE*, 48(12):92–100, 2010. ISSN 0163-6804. doi: 10.1109/MCOM.2010.5673078.

D. Griffin and D. Pesch. A Survey on Web Services in Telecommunications. *Communications Magazine, IEEE*, 45(7):28–35, 2007. ISSN 0163-6804. doi: 10.1109/MCOM.2007.382657.

Keith Griffin and Colin Flanagan. Evaluation of Asynchronous Event Mechanisms for Browser-based Real-time Communication Integration. In Khaled Elleithy, Tarek Sobh, Magued Iskander, Vikram Kapila, Mohammad A. Karim, and Ausif Mahmood, editors, *Technological Developments in Networking, Education and Automation*, pages 461–466. Springer Netherlands, 2010. doi: 10.1007/978-90-481-9151-2_80. URL `http://dx.doi.org/10.1007/978-90-481-9151-2_80`.

Keith Griffin and Colin Flanagan. Defining a Call Control Interface for Browser-based Integrations Using Representational State Transfer. *Comput. Commun.*, 34(2):140–149, February 2011a. ISSN 0140-3664. doi: 10.1016/j.comcom.2010.03.029. URL `http://dx.doi.org/10.1016/j.comcom.2010.03.029`.

Keith Griffin and Colin Flanagan. Defining a call control interface for browser-based integrations using representational state transfer. *Computer Communications*, pages 140–149, 2011b.

GSMA. OneAPI. 3GPP TS 29.199-01, 2009.

M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, April 1998. URL `http://www.ietf.org/rfc/rfc2327.txt`.

Schooler Handley, Schulzrinne and Rosenberg. SIP: Session Initiation Protocol. RFC 2543, March 1999. URL `http://www.ietf.org/rfc/rfc2543.txt`.

Ian Hickson. The websocket api. *W3C Working Draft WD-websockets-20110929, September*, 2011.

Ian Hickson and David Hyatt. Html5: A vocabulary and associated apis for html and xhtml. *W3C Working Draft edition*, 2011.

Min Huang and Lizhe Zhu. Research for Network Fault Real-time Alarm System Based on Pushlet. In *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pages 212–215, 2012. doi: 10.1109/ICICEE.2012.63.

IBM Rhapsody, 2013. URL http://www-03.ibm.com/software/products/en/ratirhapfami.

S. Islam and J. Gregoire. Converged access of IMS and web services: A virtual client model. *Network, IEEE*, 27(1):37–44, 2013.

ITU. Y.2234 : Open service environment capabilities for NGN. Recommendation Y.2234, International Telecommunication Unit, 09 2008. URL http://www.itu.int/rec/T-REC-Y.2234-200809-I.

Ivelin Ivanov. Mobicents Communication Platform, 2008. URL http://www.mobicents.org/index.html.

Jan Janak. Sip proxy server effectiveness. *Master's Thesis, Department of Computer Science, Czech Technical University, Prague, Czech*, 2003.

Java.net, 2011. URL https://java.net/projects/jax-rs-spec/pages/AsyncServerProcessingModel/revisions/32.

Josh Juneau. New Servlet Features. In *Introducing Java EE 7*, pages 1–14. Apress, 2013. ISBN 978-1-4302-5848-3. doi: 10.1007/978-1-4302-5849-0_1. URL http://dx.doi.org/10.1007/978-1-4302-5849-0_1.

J. Kellokoski, E. Tukia, E. Wallenius, T. Hamalainen, and J. Naarmala. Call and messaging performance comparison between IMS and SIP networks. In *Internet Multimedia Services Architecture and Application(IMSAA), 2010 IEEE 4th International Conference on*, pages 1–5, 2010. doi: 10.1109/IMSAA.2010.5729396.

M Kulkarni and Y Cosmadopoulos. Sip servlet specification, version 1.1. *JSR*, 289, 2008.

Seung-Ik Lee and Shin-Gak Kang. NGSON: features, state of the art, and realization. *Communications Magazine, IEEE*, 50(1):54–61, 2012. ISSN 0163-6804. doi: 10.1109/MCOM.2012.6122533.

Li Li and Wu Chou. Design Patterns for RESTful Communication Web Services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 512–519, 2010.

Lin Li and Xiping Zhang. Research on the integration of RTCWeb technology with IP multimedia subsystem. In *Image and Signal Processing (CISP), 2012 5th International Congress on*, pages 1158–1161. IEEE, 2012.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

Olga Liskin, Leif Singer, and Kurt Schneider. Teaching Old Services New Tricks: Adding HATEOAS Support As an Afterthought. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 3–10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0623-2. doi: 10.1145/1967428.1967432. URL `http://doi.acm.org/10.1145/1967428.1967432`.

Salvatore Loreto, P Saint-Andre, S Salsano, and G Wilkins. Known issues and best practices for the use of long polling and streaming in bidirectional http. *Internet Engineering Task Force, Request for Comments*, 6202(2070-1721):32, 2011.

David Lozano, Luis A. Galindo, and Luis García. WIMS 2.0: Converging IMS and Web 2.0. Designing REST APIs for the Exposure of Session-Based IMS Capabilities. In *Proceedings of the 2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, NGMAST '08, pages 18–24, Washington, DC, USA, 2008. IEEE Computer Society. doi: 10.1109/NGMAST.2008.97. URL `http://dx.doi.org/10.1109/NGMAST.2008.97`.

Franco Mazzanti. Welcome to UMC v4.1a, 2009. URL `http://fmt.isti.cnr.it/umc/V4.1/umc.html`.

L. Mazzi. API REST per la telefonia su web e interoperabilità con il protocollo SIP, 2013.

Christian Menkens and Michael Wuertinger. From service delivery to integrated SOA based application delivery in the telecommunication industry. *J. Internet Services and Applications*, 2(2):95–111, 2011.

Takaaki Moriya and Junichi Akahani. Application programming gap between telecommunication and internet. *Comm. Mag.*, 48(8):96–102, August 2010. ISSN 0163-6804.

C.E.A. Mulligan. Open API standardization for the NGN platform. *Communications Magazine, IEEE*, 47(5):108–113, 2009. doi: 10.1109/MCOM.2009.4939285.

NGSON Working Group. IEEE Standard for the Functional Architecture of Next Generation Service Overlay Networks. IEEE Standard 1903-2011, 2011.

Gerard Nicolas, Karim Sbata, and Elie Najm. Architecting end-to-end convergence of web and Telco services. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 98–105, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0784-0. doi: 10.1145/2095536.2095555. URL `http://doi.acm.org/10.1145/2095536.2095555`.

OMA. OMA Web Services Enabler OWSER Core Specification. Approved Version 1.1, Open Mobile Alliance, Mar 2006.

OMA. OMA service environment (OSE). Approved Version 1.0.5, Open Mobile Alliance, October 2009.

OMA. Enabler Release Definition for RESTful bindings for Parlay X Web Services. Technical Report V2, Open Mobile Alliance, July 2012. URL http://technical.openmobilealliance.org/Technical/release_program/docs/CopyrightClick.aspx?pck=ParlayREST&file=V2_0-20120724-A/OMA-ERELD-ParlayREST-V2_0-20120724-A.pdf.

Savas Parastatidis, Jim Webber, Guilherme Silveira, and Ian S. Robinson. The role of hypermedia in distributed system development. In *Proceedings of the First International Workshop on RESTful Design*, WS-REST '10, pages 16–22, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-959-6. doi: 10.1145/1798354.1798379. URL http://doi.acm.org/10.1145/1798354.1798379.

V. Pimentel and B.G. Nickerson. Communicating and Displaying Real-Time Data with WebSocket. *Internet Computing, IEEE*, 16(4):45–53, 2012. ISSN 1089-7801. doi: 10.1109/MIC.2012.64.

Marek Potociar. Jsr 311: Jax-rs: the java api for restful web services. *Technical report*, 2009.

L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly & Associates, May 2007.

J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocol. RFC 5245, April 2010. URL http://www.ietf.org/rfc/rfc5245.txt.

J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. URL http://www.ietf.org/rfc/rfc3264.txt.

J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, June 2002a. URL http://www.ietf.org/rfc/rfc3261.txt.

J. Rosenberg, H. Schulzrinne, B. Campbell, C. Huitema, and D. Gurle. Session Initiation Protocol (SIP) Extension for Instant Messaging. RFC 3428, December 2002b. URL http://www.ietf.org/rfc/rfc3428.txt.

Frederick Schulzrinne, Casner and Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003. URL http://www.ietf.org/rfc/rfc3550.txt.

Java Servlet Specification. Version 3.0, oracle america. *Inc., Maintenance Release*, 2011.

SunMicrosystems, Inc. The SIP Servlet Tutorial, 2008. URL http://docs.oracle.com/cd/E19355-01/820-3007/.

Maurice H. ter Beek, Franco Mazzanti, and Stefania Gnesi. CMC-UMC: A Framework for the Verification of Abstract Service-oriented Properties. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 2111–2117, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. doi: 10.1145/1529282.1529751. URL `http://doi.acm.org/10.1145/1529282.1529751`.

Thijssen. Asynchronous operations in REST, 2011. URL `http://www.adayinthelifeof.nl/2011/06/02/asynchronous-operations-in-rest/`.

Dragos Vingarzan et al. IMS/NGN Performance Benchmark Part 2: Subsystem Configurations and Benchmarks, 2007. URL `http://webapp.etsi.org/workprogram/Report_WorkItem.asp?WKI_ID=25501`. ETSI/TISPAN 6 Workitem 06024-2.

Ivan Zuzak, Ivan Budiselic, and Goran Delac. A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems. *J. Web Eng.*, 10(4):353–390, 2011.