Formalizing REST APIs for web-based communication and SIP interworking

Federica Paganelli · Terence Ambra · Alessandro Fantechi · Dino Giuli

Received: date / Accepted: date

Abstract Significant research efforts for the convergence of Web and Telecommunication services have been recently spent by research and industry stakeholders. The IETF and W3C are cooperating in specifying how web browsers should evolve to natively support communication services. In this perspective, devising novel mechanisms for signaling message exchange and possible interworking between Web- and SIP-based systems is a hot topic of research. Indeed, discussions are 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 paper we propose a rigorous approach for design and implementation of REST communication services (e.g., a call service) which leverages formal verification techniques, while allowing to meet a specific performance requirement (i.e., maximum call setup delay). First, we formalize the call resource behavior through a Finite State Machine representation by modeling and simulating service expected behavior and its interworking with SIP User Agents through a tool for the analysis of communicating state machines. Then, we use the model-checking capabilities offered by the tool for the verification of formal properties. Finally, we implement a prototype that, thanks to the previous formalization step, is shown to be functionally correct, while yielding acceptable performance.

Keywords web services · internet of services · REST · final state models · call control · SIP

Federica Paganelli
CNIT Research Unit at the University of Florence, via S. Marta 3 50139 Florence, Italy
E-mail: federica.paganelli@unifi.it

Terence Ambra
Department of Information Engineering at the University of Florence, via S. Marta 3 50139 Florence, Italy
E-mail: terence.ambra@unifi.it

Alessandro Fantechi
Department of Information Engineering at the University of Florence, via S. Marta 3 50139 Florence, Italy
E-mail: alessandro.fantechi@unifi.it

Dino Giuli
Department of Information Engineering at the University of Florence, via S. Marta 3 50139 Florence, Italy
E-mail: dino.giuli@unifi.it

1 Introduction

The Web is evolving from a document-centric paradigm to an increasingly interactive and collaborative form of information sharing and real-time communication. Indeed, some standardization efforts by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) have recently started for defining the Web Real-Time Communication (WebRTC) interface and protocol specifications [13] to allow the native support of voice and video communications by web browsers. These standardization activities focus on the transfer of media streams between browsers, but do not deal with signalling, which is left to service developers, nor with interoperability with legacy systems, such as systems based on the Session Initiation Protocol (SIP) [58].

In the telecommunication domain, several web-based APIs have been defined by the research and industry communities for the signaling and control of peer-to-peer or multiparty communication services provided by legacy telecommunications platforms.

The REpresentational State Transfer (REST) architectural style, which was introduced by Fielding [23] as a resource-oriented architecture style for networked

systems, is considered a best practice for the design of web service interfaces. REST principles have thus been applied to the design of signaling Application Programming Interfaces (APIs) in several standardization efforts, such as in OMA RESTful bindings for Parlay X Web Services [48] and RESTful Network API for WebRTC Signaling [49], as well as in several research works, as discussed by Belqasmi et al [12].

SIP-based architectures definitely play a major role among telecommunications legacy frameworks, as argued by Amirante et al [6]. Interworking between the emerging browser-enabled systems and SIP-based ones is thus a hot issue [6, 60]. Actually, convergence of HTTP and SIP domains is not straightforward since these protocols rely on different principles. Indeed, typical usage of SIP is stateful and peer-oriented, while HTTP is stateless and based on the client-server model [14, 35]. Several authors worked on this topic and discussed resource-oriented design of REST-based APIs for the convergence of Web- and Telecom-centric services. To the best of our knowledge, most works in this research area focus on the design of the signaling APIs [39, 19, 28]. Only some of them also discuss implementation details [19], but none of them adopts a rigorous approach for modeling the required resources and their behavior.

Indeed, lack of rigorous modeling is a limitation of many works claiming to be RESTful. As argued by Zuzak et al [65] this fact is causing *"negative effects, such as confusion in understanding REST concepts, misuse of terminology and ignorance of benefits of the REST style"*. Thus, REST principles are often misunderstood and misapplied. As discussed by Roy Fielding in his blog [24], several systems that claim to be RESTful are simply HTTP-based Remote Procedure Call (RPC) implementations and violate REST principles.

Widespread misunderstanding of REST concepts results in a difficulty in fully taking advantage of REST benefits (that is scalability, interoperability and simplicity). Indeed, REST is an abstract model for the Web architecture [25], hence it describes how a Web application should behave to maximize desirable properties, such as simplicity and performance [65]. As the Web gets more complex and grows in functionality, (i.e., real-time communication, Web of Things [50], Linked Data [15]), preserving its desirable properties becomes a major concern.

In the application domain of this work (real-time communication on the web), the lacking adoption of formal modeling also makes it difficult to design and verify the correct interworking with external systems and protocols (i.e., SIP-based systems), as needed. For instance, there is no guarantee that error conditions, such as deadlocks, will not occur. On the other hand, formal development techniques have been criticized for encouraging the production of inefficient, even if correct, software [9].

The objective of this work is pursuing a more rigorous approach for the design of REST signalling APIs for real-time communication on the web, leveraging formal verification techniques, while guaranteeing the satisfaction of a specific performance requirement (i.e., maximum call setup delay).

More specifically we present a new approach and related results for the design and implementation of a communication service interworking with SIP-based systems and exposing a RESTful API (i.e., a service interface compliant with REST principles). We will use a call service as reference example throughout this paper. By leveraging a resource-oriented service design methodology, our original contribution is twofold. First, 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. Analogously, the state machine formalism is used to model the components of a SIP-REST gateway. As a result, the RESTful call service and the components for its interworking with SIP User Agents (UAs) are represented through a set of communicating state machines, while their behavior can be simulated through a tool for the analysis of communicating state machines. Then, we show the use of the model-checking capabilities offered by the tool for the verification of formal properties (e.g., deadlock absence). Second, we present a web application prototype that implements the specifications originating from the FSM formalization. The prototype offers a RESTful real-time communication service accessible to web browsers and supports the interworking with SIP UAs. We show how, thanks to the adopted tool, the behavior of the prototype can be easily compared with the expected behavior exhibited by the simulation of the model. Moreover we also present preliminary performance analysis results.

Thus, the proposed approach improves current research by enhancing the design of a REST API for real-time communication with formal modeling. This allows to simulate and validate the call service model against functional requirements, including the correct interworking with SIP UAs. We also report on the implementation of a prototype to check: i) its coherence with respect to the validated model and ii) whether it meets quantitative performance requirements.

The remainder of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we describe our approach for the design of a RESTful real-time communication service interworking with SIP sys-

tems, which relies on the adoption of a state machine formalism. We also discuss the used tool for simulating the service expected behavior and verifying some formal properties. Section 4 describes the implementation of the prototype and its performance analysis. Finally, Section 5 concludes the paper by discussing obtained results as well as providing some insight into future work.

2 Related work

Recent technological evolution in the telecommunication domain has been driven by the increasing need for making capabilities of an operator's network accessible and invokable by external consumer applications. To this purpose, service-oriented principles [21] have inspired the specifications of Service Delivery Platforms that expose telecom capabilities via open APIs to enable enhanced and flexible service provision and composition.

Menkens and Wuertinger [43] highlight major obstacles towards the development of web-telecom converged applications: i) 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 supporting developers in composing telecommunication services with Web services; ii) telecommunications specifications, such as IMS and SIP [58], are not supported by default by widely adopted platforms for mobile devices; and iii) application developers typically adopt Internet and Web protocols and data formats. More specifically, convergence of Web and SIP-based services is considered difficult to achieve, since HTTP and SIP are based on different principles [14, 35]: i) typical usage of SIP is stateful, while HTTP is stateless; ii) SIP is peer-oriented while HTTP is based on the client-server paradigm.

2.1 Real-time Communication on the Web

The need of migrating real time communication from dedicated VoIP networks to the Web has been deeply analyzed in [61], where a comparison of the complexity of SIP-based systems is made with that of Web architectures. This need has become more critical with the upcoming of standards for real-time communications on the web that have been promoted by the IETF and the W3C (i.e., RTCWeb protocol [5] and WebRTC APIs [13]) for the native support of direct and interactive communication between browsers without the need of third-party browser plugins, such as Adobe Flash Player.

Indeed, WebRTC specifies a set of ECMAScript APIs that support the exchange of media streams between two browsers. These APIs allow web application developers to develop novel JavaScript applications that exploit such browser capabilities. As mentioned above, the signaling mechanism is not specified in the standard and left to application developers. A major requirement in the choice of the protocol supporting signaling message exchange is to enable bidirectional communication between browser and server. This could be realized through different mechanisms, such as long-held HTTP requests (e.g., HTTP long polling) and the WebSocket protocol, which establishes a bidirectional channel for message exchange over TCP [22], as discussed in [52, 2, 34].

Sege et al [59] provide a brief literature survey on WebRTC and SIP integration. Li and Zhang [40] discuss the need for integrating WebRTC with IMS and provide a preliminary description of an integration solution, while Amirante et al [6] discuss the main technical issues entailed by the integration of SIP-based solutions with WebRTC applications and propose a working solution for a conferencing system. Finally, a general-purpose open WebRTC gateway is described in [7].

2.2 Web APIs for Telecom services

In order to effectively support third party application developers, some standard specifications for the web-based exposure of telecom services have been defined [45, 12]. Web-based interfaces may be distinguished into those that comply with Web Service (WS) specifications and those complying with REST guidelines.

The Open Mobile Alliance (OMA) has defined a web service framework called OMA Web Services Enabler [47]. The Parlay group, which is a standardization body working in collaboration with OMA, the Third Generation Partnership Program (3GPP) and the European Telecommunications Standards Institute (ETSI), has defined the Parlay X specifications [1]. Parlay X is a set of Web Service APIs for accessing a wide range of telecom network capabilities, such as third party call control, call notification, short messaging, and payment. OMA has released the specifications for the RESTful bindings for Parlay X Web Services in 2012 [48]. The currently available version (version 2.0) includes simple non-session services such as short and multimedia messaging, payment and location services, and accessory features for call services. OMA has also recently released a set of REST APIs for WebRTC signalling over HTTP [49]. The upcoming of this standard strengthens the suitability of REST for communication signaling on

the web, but, to the best of our knowledge, no implementation is available yet.

Several research works have investigated the adoption of Web services for exposing telecommunication capabilities [17, 27]. Recently, researchers are increasingly focusing their efforts on RESTful services, rather than on WS ones, since RESTful services are deemed more lightweight and close to web application programming models [3, 16].

Fu et al [26] present an early feasibility prototype for a REST service architecture for bridging presence service across heterogeneous domains. Moriya and Akahani [44] conduct an experiment with human participants to evaluate their productivity in developing web-telecom applications with Parlay X. They detect two major problems: i) programmers may not be aware of the call session state since the SOAP/HTTP-based interface makes the interaction look stateless and synchronous; ii) analogously, programmers may apply a procedural style, while disregarding the event-driven (i.e., asynchronous) nature of telecommunication services.

Handling of session-based capabilities (e.g., a call between two end users) is discussed in several works. Lozano et al [41] propose a set of REST APIs for the exposure of session-based IMS capabilities. Asynchronous notification is handled through HTTP polling. Davids et al [19] discuss different options for allowing voice and video communications on the Web. They propose a RESTful web communication API over HTTP, where asynchronous notification is realized through long-lived HTTP. Nicolas et al [46] propose an approach for telecom and web service convergence that exploits the WebSocket protocol. However, the design of REST APIs is not discussed in detail and the message flow is described only for presence and location update services. Griffin and Flanagan [28] apply a resource-oriented design methodology to define a call control interface that can be exploited by browser-based applications. They take a simple call model as reference, by adapting the Computer Supported Telecommunications (CSTA) industry standard. Also Li and Chou [39] analyse CSTA services and propose some REST design patterns to properly model communication services within REST.

2.3 Formal models for communication services and protocols

Use of formal models is not a widespread practice in commercial development of networking protocols and applications [54], despite their known benefits in terms of property verification and unambiguous specification. In fact, this lack of rigor has lead to the widely accepted adoption of simulation-based testing as the current practice for validating networking protocols, software and hardware.

Nonetheless, several works have applied formal methods to the verification of networking protocols and systems. For instance, Zave [64] presents a Promela model of invite dialogs in SIP. Other authors [20, 8] propose a Coloured Petri Net model of the SIP INVITE transaction to verify some functional properties of SIP.

As regards REST systems, Zuzak et al [65] explore the use of a FSM-based approach for modeling RESTful systems with the goal of contributing to the understanding of the REST style. Porres and Rauf [53] present an approach to model the structural and behavioral interface of a RESTful web service. They use UML class and protocol state machine diagrams to model the conceptual and behavioral aspects of the web service and generate a contract in the form of preconditions and postconditions for methods of an interface.

To the best of our knowledge, the use of formalization in the design of a REST real-time communication service interworking with SIP-based systems has not been investigated yet.

2.4 Motivation of our work

Our work basically accounts for the results achieved by related work in the design of RESTful communication services, especially results achieved by Li and Chou [39], Davids et al [19] and Griffin and Flanagan [28], mentioned above. However, none of them delve into the modeling of either the call flow or interoperability with SIP legacy systems. Griffin and Flanagan [28] analyze the normal and error/exception flows for a call in a narrative way, i..e. without adopting a formal model.

Instead, formal modeling may ease the design and verification of a system and its correct interworking with other systems. Therefore, the main contribution of this work consists in enhancing the design of REST API for real-time communication services with formal modeling to simulate and validate a call service model against functional requirements. We adopt Finite State Machines to model main components of a RESTful communication service, including those handling the interworking with SIP UAs, and verify some properties of the system.

3 FSM-based design of REST communication services

In this section we show a design approach for REST communication services which enhances a well-known REST-oriented design methodology [55] through the adoption of a state machine formalism and a tool for the analysis of communicating state machines which allows to verify the dynamic behavior of the system. We use a call service as a reference example of communication services throughout the rest of this paper.

We first introduce the adopted formalization approach and main REST principles. Then, we describe the design of a REST communication service interworking with SIP and we also discuss how we handled Web and Telecommunications convergence issues. Finally we verify some desired properties of the modeled system.

3.1 Formalization

In order to provide a formal model of the call resource behavior, we chose to adopt the UML State Diagram notation since it is widely adopted and intuitive to understand. In particular, a UML State Diagram is adopted for defining the behavior of each distinguished partner in the protocol. Indeed, since we do not use the advanced features offered by UML State Diagrams (such as hierarchical states, histories, concurrent regions), each diagram is actually a Finite State Machine (FSM). The full model is anyway made up of a collection of such machines, according to a UML Component Diagram serving as a reference.

We have adopted the UML on the fly Model Checker (UMC) tool [11] to support the design, simulation and verification of the developed model. UMC 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 UML state diagrams. UMC accepts a system specification given in UML-like style as a collection of active objects, modelled by state-machines.

Adopting UMC obliges us to use semantically well-founded state machine descriptions, since the tool enforces the standard formal semantics given to UML State Diagrams.

We could have adopted another model checker (such as SPIN [31], which is popular as a formal protocol validation tool) to formally prove desired properties of the produced model. This would have required however a translation of the UML State Diagrams into the specific model checker input language (Promela for SPIN), with the possibility of introducing semantic misinterpretations.

We have used the two main functionalities offered by the tool for the analysis of the behavior of a system:

- the ability to simulate the model computations;
- the model checking ability.

The former functionality has been used for verifying the model itself, but especially to define the correct computations to which the implemented prototype should adhere. Such computations have been expressed for a better visualization as Message Sequence Charts (or UML Sequence Diagrams), as shown in the following sections. The expression of the computations in this form is however not supported by the tool.

The latter functionality has been used to verify some desired properties for the modeled system, in order to validate the model. Indeed, UMC allows to verify properties specified in a variant of the CTL temporal logic [18], called UCTL [10]. UCTL combines both branching-time and linear-time operators. In a linear temporal logic, operators are provided for describing events along a single computation path. In a branching-time logic the temporal operators quantify over the paths that are possible from a given state. In CTL a path quantifier can prefix an assertion composed of arbitrary combinations of linear-time operators, according to the (simplified) syntax:

1. Path quantifiers are:
   - A - “for every path”
   - E - “there exists a path”
2. Linear-time operators are:
   - Xp - p holds true next time.
   - Fp - p holds true sometime in the future
   - Gp - p holds true globally in the future
   - pUq - p holds true until q holds true

In section 3.5 we show the use of UMC for verifying some behavioral properties, defined by means of a subset of the operators above.

In synthesis, the adoption of UML State Diagrams and UMC has allowed us to design and validate a formal model of the developed protocol.

3.2 REST principles

The REST architectural style was proposed by Fielding [23] in his doctoral dissertation as an architectural style for building large-scale distributed hypermedia systems. On the REST vision, data sets and objects handled by client-server application logic are modeled as resources. The key principles of REST are fivefold [23]:

1. *URIs as resource identifiers.* Resources are exposed by servers through Uniform Resource Identifiers (URIs).

Since URIs belong to a global addressing space, resources identified with URIs have a global scope;
2. *Uniform interface*. The interaction with the resource is fully expressed with four primitives, i.e., create, read, update and delete;
3. *Self-descriptive messages*. Each message contains the information required for its management (metadata are used for content negotiation and errors notification);
4. *Stateless interactions*. Each request from client to server must contain the information required to fully understand the request, independently of any request that may have preceded it;
5. *Hypermedia As The Engine Of Application State (HATEOAS)*. A hypermedia system is characterized by participants transferring resource representations that contain links; the client can progress to the next step in the interaction by choosing one of these links [51, 55].

We adopted the methodology for resource-oriented design proposed by Richardson and Ruby [55]. According to this methodology, designers have to first figure out the dataset on which the service will operate, and split it into resources. Then, 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.

3.3 RESTful design of web-based communication services

Hereafter we describe how we designed REST signalling APIs for real-time communication on the web.

We took into account a simple case study for exemplifying the proposed approach, namely a call setup between i) two REST clients (e.g., web browsers), and ii) a REST client and a SIP UA. In both cases an intermediary component is needed for the signalling message exchange required for the call setup. The media path does not necessarily require an intermediary component, unless additional proccesing is required (e.g., transcoding).

3.3.1 Defining the service domain and resources

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 defined the following main resources: *calls* and *call*.

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. It also offers a factory method to instantiate new calls and retrieve existing calls, as explained in subsection 3.3.3.

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.

3.3.2 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). [1] 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}`.

3.3.3 Uniform interface

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.

Table 1 shows the operations which can be invoked on the *calls* and *call* resources and represent the exposed REST API. GET and POST methods can be invoked on the *calls* resource, while the *call* resource exposes the PUT, GET and DELETE methods. For instance, a POST request on the `/calls` URI is for 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}`).

3.3.4 Resource representation

According to Fielding [23] "*REST components perform actions on a resource by using a representation to cap-*

[1] The use of verbs in URI fragments is a common practice in web development outside of REST.

Table 1 REST API for call resource management

Resource URIs	HTTP Method	Description
`/calls`	GET	Retrieve a list of calls
`/calls`	POST	Create a new call resource
`/calls/{call_id}`	PUT	Update the call resource
`/calls/{call_id}`	DELETE	Delete the call resource
`/calls/{call_id}`	GET	Retrieve the call resource

ture the current or intended state of that resource and transferring that representation between components". At each interaction step, a representation may indeed 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 *call* resource representation contains the following data fields:

- *to*, indicates the callee, identified by a URI;
- *from*, indicates the caller, identified by a URI;
- *state*, indicates the call state;
- *offer*, contains the session description that the caller sends to the callee to request the establishment of a call [57]. This information is represented according to the Session Description Protocol (SDP) [30], which is a standard format for describing streaming media initialization parameters.
- *answer*, contains the session description that the callee sends to the caller in response to an offer to negotiate the media session establishment.

3.3.5 Modeling the Service Behavior as FSM

We modeled the behavior of the *call* resource through a FSM representation, as shown in Fig. 1.

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 through INVITE client and server transactions [58].

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 carried out to inform the other peer that the resource reached a new state and new transitions are permitted, in compliance with the REST HATEOAS constraint, as discussed in section 3.3.6.

In detail, the FSM model shown in Fig. 1 represents the state evolution of the *call* resource for the call session setup between two REST Clients (caller and 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;
2. *Calling*, indicates an initiated call;
3. *Timeout*, *Cancel*, *Busy* and *Error*, indicate that the call failed due to out of time, caller-side call cancellation, callee-side call cancellation (e.g., the callee is busy) and request errors events, respectively;
4. *Proceeding*, indicates a call in progress;
5. *Answered*, indicates that the callee accepted the call;
6. *Acked*, indicates that the caller confirmed the call;
7. *Closed*, indicates a terminated call.

The call resource behavior is briefly described hereafter. For the sake of conciseness, we limit the description of the *call* resource behavior to the case of a successful call and we only analyze some possible error conditions.

Starting from the initial pseudo-state the caller performs a POST request on the `/calls` URI to trigger the creation of a *call* resource (NEW state). The newly created *call_id* is returned back to the client. The client then invokes a PUT operation on the `/calls/{call_id}` URI and passes the identifier of the callee and the offer description encoded in the standard Service Description Protocol (SDP) format in the message body. The call state is updated to CALLING. Next intermediate transitions are all triggered by a PUT request on the `/calls/{call_id}URI`, which can be sent by the caller, the callee and/or the server for continuing the session setup process by properly 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. Then, 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 a 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 kept them separate for clarity.

3.3.6 Web and Telecommunications convergence

The FSM-based representation of the call resource helps understanding how we modeled the SIP peer-to-peer

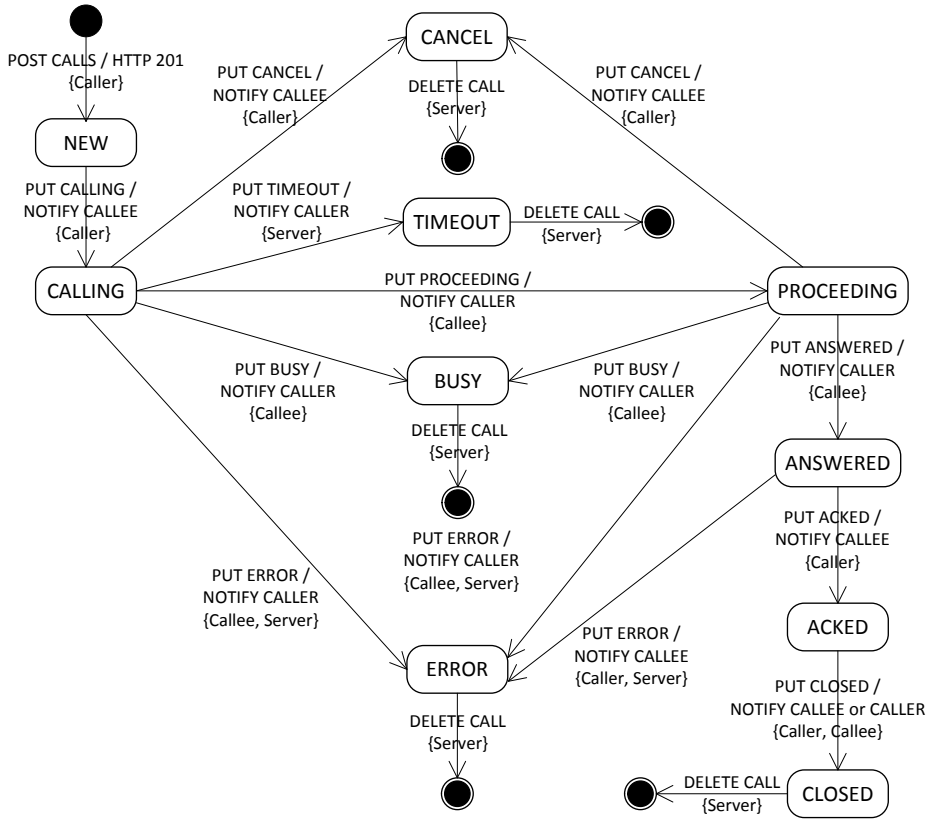

Fig. 1 Finite-state machine of the call resource with two REST clients.

paradigm onto the Web client-server one and how we handled the compliance with the HATEOAS constraint (introduced in Section 3.2).

SIP UAs are peers that can act as clients or servers, i.e., initiate or respond to SIP transactions. In our reference scenario, two REST clients are involved in the call establishment. With respect to the call, they should act as peers (i.e., a REST client should be able to initiate or receive a call). However, this behavior should be mapped onto the REST client-server interaction model. This means that the state of the call resource evolves according to the interactions of both clients with the server. In order to manage this aspect, we introduced a notification mechanism. Thus, when the call state has changed, due to an action performed by one REST client or the occurrence of other events (e.g., a timeout), the server sends a notification message to the second client to signal that the call state has changed.

Moreover, the notification message is used to convey the reference to the next permitted transitions (i.e., the actions that can be performed by the notified client). This is the key mechanism that guarantees compliance with the HATEOAS constraint. Indeed, a REST service is a web of interconnected resources with an underlying hypermedia model that determines not only the relationships among resources but also the possible net of resource state transitions [4]. REST clients discover and decide which links/control to follow/execute at runtime. According to this constraint, an application evolves through subsequent transitions of resources from one state to another. The system can thus model and advertise permitted transitions by means of resource representations delivered to clients [51]. Then, client applications decide which possible forward steps can be run based on their specific application goals and/or through end users' actions. Thus, the client-server interaction is stateless since the request sent by the client contains all the information needed by the server to process it (e.g., the Call resource URI and the action to be performed). On the contrary, SIP UAs are stateful, i.e., they maintain the dialog or transaction state. Thus, signaling messages for a call setup contain information strictly related to the call session evolution and next permitted transitions are encoded in the SIP UA implementation that maintains the client and server transactions state machines.

3.4 Interworking with SIP

The FSM representation shown in Fig. 1 can also model a call between a REST client and a SIP UA. 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, which implements the notification action according to the SIP specifications.
2. SIPMessageReceiver, which 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. 1), the SIPMessageReceiver, the SIPMessageSender, the REST client and the SIP UA state machines (described hereafter and shown in Figs. 2, 3, 4 and 5, respectively).

Fig. 2 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. Then, the action related to this transition is performed and 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. 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 UA (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.

Fig. 3 models the behavior of the SIPMessageReceiver component. This state machine is similar to the previous one, since it includes only the Standby and Executed states. The transition to the Executed state is activated by a SIP message (SIP_MESSAGE_IN). 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

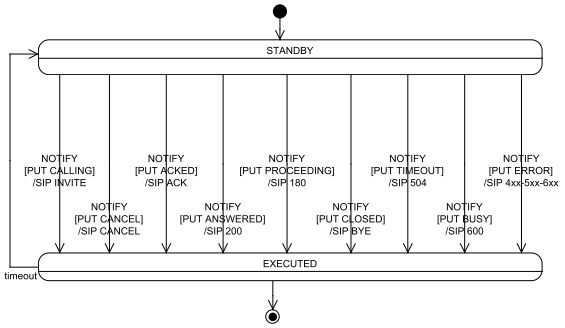

Fig. 2 Finite-state machine of the SIPMessageSender component.

SIPMessageReceiver requests a call resource creation (POST request) triggered by the reception of a SIP INVITE message from the SIP UA. Then, upon the reception of a HTTP 201 message sent by the call resource component, a PUT Calling request is invoked (this explains why Fig. 3 shows a transitions triggered by an HTTP_MESSAGE_IN event).

The REST Client and SIP UA state machines have been defined for modeling user agents acting as caller and callee, but are not part of the contribution of this work. Therefore we provide them for the sake of completeness, but we limit the description of their behavior to a scenario of a successful call. Fig. 4a and Fig. 4b show the REST Client FSM acting as the caller and the callee, respectively. Fig. 5a and Fig. 5b show the SIP UA Client and Server FSMs, which are based on the SIP INVITE client and server transaction, respectively [58].

The model of the overall system is shown in Fig. 6 by using a UML Component Diagram representation, which describes how a system is split 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 is an object instance, which is exposed as a state machine. Fig. 6 shows the event-based operations exposed by each FSM class interface and the dependency of other classes on these interfaces.

3.5 Formal verification with UMC

We used the UMC tool available on the web (version 4.2) [42] to verify some behavioural properties expressed in UCTL.

First we verified deadlock absence, since it is a well-known property which is desirable in most systems.

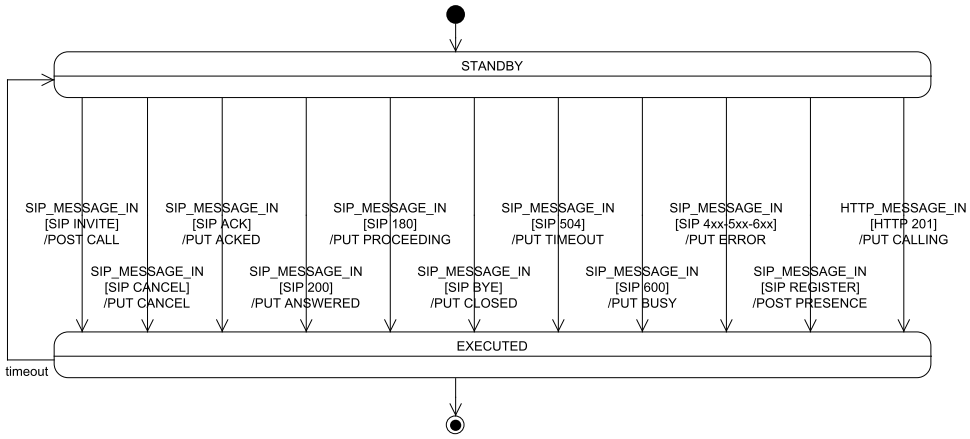

Fig. 3 Finite-state machine of the SIPMessageReceiver component.

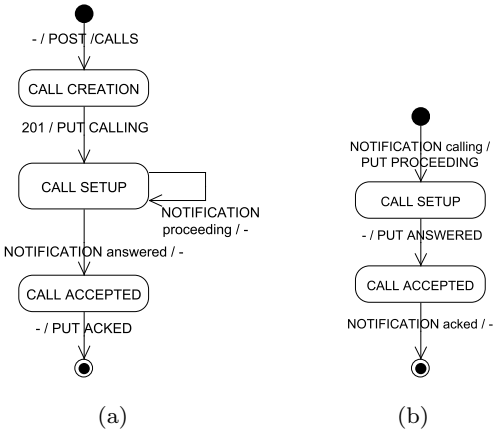

Fig. 4 Finite-state machines of the REST Client component for a scenario of successful call: (a) REST Client acting as the caller, and (b) REST Client acting as the callee.

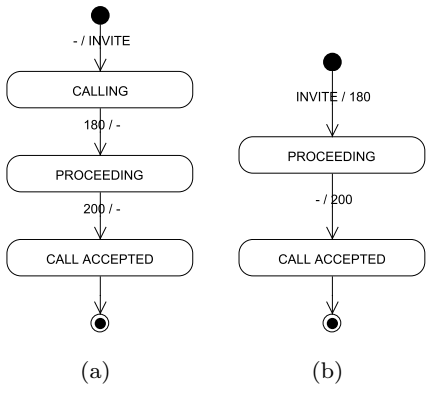

Fig. 5 Finite-state machines of the SIP UA Client and Server component for a scenario of successful call: a) SIP UA Client acting as the caller, and b) SIP UA Server acting as the callee.

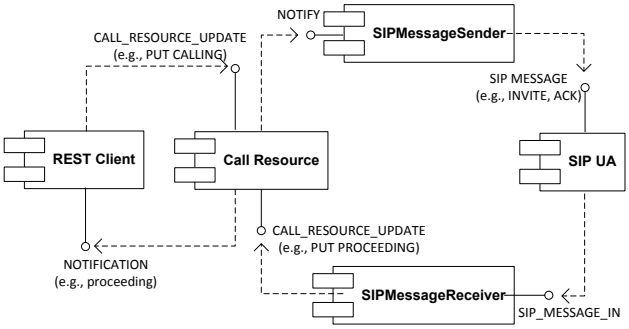

Fig. 6 Component diagram for the communicating state machines model.

Then, we verified a service-oriented property (service responsiveness) and a property characterizing RESTful services (resource connectedness).

3.5.1 Deadlock absence

The absence of deadlocks is expressed by the UCTL formula F1 in Table 2, which means "for every path the CLOSED state is eventually reached", namely the UMC model permits to always terminate a successful call verifying the deadlock absence condition.

Fig. 7 shows an excerpt of the model-checking capabilities offered by the UMC tool representing the detailed explanation of the result of the formal verification of this deadlock absence property.

3.5.2 Service responsiveness

The objective is to check if a service is responsive, i.e., if it guarantees a response to each received request [10]. An example is expressed by the UCTL formula F2 in

Table 2 Formulae for verification of properties with the UMC tool

Verified formulae	
Deadlock Absence	F1 = $AF\ CLOSED$
Service Responsiveness	F2 = $AG\ [\ PUT\_CALLING\ ]$ $A\ [\ true\ {}_{true}U\ NOTIFY\_PUT\_PROCEEDING \vee PUT\_CANCEL \vee NOTIFY\_PUT\_TIMEOUT \vee NOTIFY\_PUT\_BUSY \vee NOTIFY\_PUTERROR\ true]$
Resource Connectedness	F3 = $\neg E\ [\ true\ {}_{\neg\ HTTP\_201\_MESSAGE}\ U\ PUT\_CALLING\ true]$

Fig. 7 Formal verification of deadlock absence by the UMC tool.

Table 2, which states that each time a request for a call setup is triggered (through a PUT_CALLING request), in all computations at a certain time the caller is notified of either a successful or aborted call setup progress. More precisely, the caller is notified that the state of the call resource has been updated to a Proceeding, Timeout or Busy state through the corresponding PUT request, unless the caller itself has terminated the call (PUT_CANCEL request).

3.5.3 Resource Connectedness

Resource connectedness refers to the property wherein every resource in the web service is reachable from the base resource by successive requests. Resource connectedness is an important property of RESTful web services which is implied by the HATEOAS constraint, as discussed in Section 3.3.6. An example is provided by the formula F3 in Table 2, which states that it may never happen that a PUT_CALLING request is triggered if a HTTP 201 Message has not been sent before. This formula expresses a necessary condition under the HATEOAS constraint for a PUT_calling request to be invoked on a `/calls/{call_id}` URI. An HTTP 201 Message is sent in response to a POST request on the `/calls/` URI and its body should contain the `/calls/{call_id}` URI. Therefore, the formula means that a PUT request for changing the state of a call resource to CALLING cannot been issued if a POST request on the `/calls/` URI has not been triggered before. It is to note that the UMC model does not include the specifications of the message content and references to URIs, but expresses only the communication skeleton.

We model checked the above mentioned properties and results are shown in Table 3.

Table 3 Validation results

Verified formulae	**F1**	**F2**	**F3**
Validation result	true	true	true
Evaluation time (ms)	12.03	13.08	0.66

4 Prototype Implementation

This section introduces the prototype that we implemented by taking into account the REST API and the state machine specifications described in Section 3. This prototype offers a real-time communication service (i.e., voice and video call) between web browsers as well as web browsers and SIP UAs. The prototype also offers additional services, in particular a presence management service, briefly described in Section 4.2, which is needed in order to track users availability status and contact details.

First, we describe the prototype architecture and related implementation details. Then, we show how we evaluated the prototype to check: i) its coherence with respect to the validated model (subsection 4.6) and ii) whether it meets quantitative performance requirements (subsection 4.7).

4.1 Prototype Architecture

The prototype is a web application, which is made of the following main modules, as shown in Fig. 8:

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

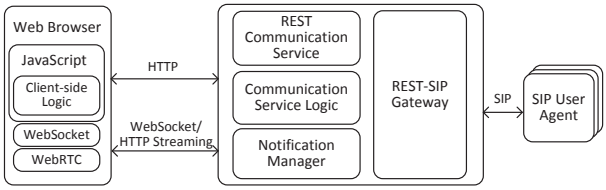

Fig. 8 Functional architecture of the prototype.

4.2 REST Communications Service implementation

The Communication Service has been implemented as a Java-based web application deployed on an Apache Tomcat 7.0 servlet container. The implementation of the REST Communication Service is based on Jersey, a Java-based framework for developing RESTful Web Services.

The Communication Service Logic includes the classes implementing the behaviour of the exposed resources (i.e., *Calls* and *Call* classes), as well as the classes handling the connection with the database for data persistence.

The REST-based exposure of these resources is handled by appropriate classes built using the Jersey library (*CallsResource* and *CallResource* classes). Through this interface, messages are exchanged that contain a representation of resources. These messages can be serialized according to different formats. For instance, both JSON and XML are easily supported by the majority of available REST libraries. We chose JSON since it is less verbose and requires less time for serialization/deserialization than XML [29].

For the sake of completeness, we briefly introduce a set of additional resources (i.e., *presences* and *presence*) that have been implemented to handle user registration and presence update in the system, which is a prerequisite for a call setup.

The *presence* resource represents a user's availability status and contact information. This resource is exposed to REST clients and SIP UAs in a way similar as for the *call* resource. A REST client directly sends a POST request on the `/presences` URI to trigger the creation of the *presence* resource for that user. Then, it subscribes to the events of interest and creates a notification channel, as explained in section 4.3. A SIP UA sends a SIP REGISTER message to the REST-SIP Gateway to provide the server with 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. Event notifications towards the REST Client and SIP UA are handled by the Notification Manager and REST-SIP Gateway, respectively.

4.3 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 two alternative solutions for handling asynchronous notifications: one solution

uses the WebSocket Protocol [22]; the second solution is based on the HTTP Streaming mechanism and the asynchronous processing of HTTP requests provided by application containers implementing the Servlet 3.0 specifications [37]. Implementation details for these mechanisms are reported hereafter, while their comparative evaluation is discussed in Section 4.7.

4.3.1 Notification over WebSocket

WebSocket allows web browsers and servers to exchange messages through a bidirectional channel over TCP. In our prototype, the server uses a WebSocket channel that has been previously setup upon a client's request to deliver notification messages. According to the WebSocket specifications [22], a WebSocket connection is established through an initial handshake between client and server. First, the browser sends an HTTP request asking for switching to the WebSocket protocol (i.e., the request message must contain an Upgrade header). If the server accepts the protocol switching, it sends a response with the 101 Switching Protocols code in the Status Line. The HTTP connection is replaced by the WebSocket connection, while an Observer object instance is created to handle the delivery of resource updates to this client (see Fig. 9). The class that specializes the Observer by implementing the notification over the WebSocket protocol is called CallWSWriter.

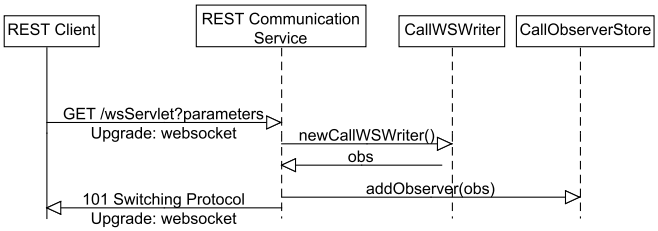

Fig. 9 Subscription for notifications over a WebSocket channel.

4.3.2 Notification through HTTP streaming and Asynchronous servlet processing

This solution implements the widely adopted HTTP Streaming server push mechanism [32]. In the HTTP Streaming approach, the browser initiates a connection to the server through a request and the server keeps the connection open instead of returning back a complete response and immediately signaling the connection closure (chunked transfer encoding). The server can send response updates (e.g., event notifications) through this long lasting connection. We implemented this server push mechanism by exploiting the asynchronous processing of requests, as allowed by Servlet 3.0 implementations.

Asynchronous processing means that in case of a request, a thread is in charge of accepting the request and putting it into a processing queue. Such thread does not get blocked and can be used for other tasks. Consequently, especially in case of asynchronous HTTP requests, this mechanism allows a more efficient usage of resources.

A client makes a subscription to the notification services through a POST request on the `/notifications/{userId}` URI. The server handles this request by instantiating an Observer instance (see Fig. 10). The class that specializes the Observer by implementing the HTTP streaming notification mechanism is called CallACWriter.

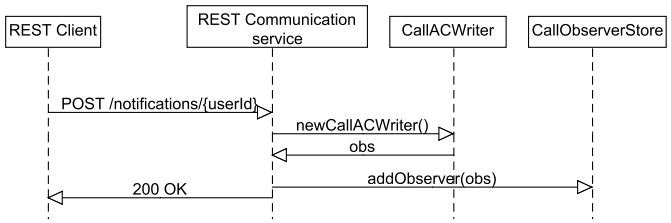

Fig. 10 Subscription for notifications through HTTP Streaming and asynchronous servlet processing.

4.4 REST-SIP Gateway

The REST-SIP Gateway is made of two main components, called SIPMessageSender and SIPMessageReceiver, which implement the specifications described in section 3.4. The SIPMessageSender handles the delivery of notification messages directed to SIP UAs. The SIPMessageReceiver handles the communication in the opposite direction. It receives messages originating from SIP UAs 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 [36].

4.5 Client-side logic

This component is made of JavaScript files that are processed by web clients to handle the signalling mes-

sages exchange 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 [13].

These scripts handle the interaction with the user, the invocation of REST methods and the handling of notifications sent by the server. The media channel establishment relies on the WebRTC API, namely the *getUserMedia* function, which allows a web browser to access 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 for negotiating 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 contain 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 that can be invoked by the callee, i.e., 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. 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.

4.6 Coherence with design specifications

In this subsection we analyze some reference scenarios to show how we verified that the implemented prototype behavior is coherent with the specifications formalized through the communicating state machines.

Fig. 11 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 updates the call status to CALLING. This change will be notified to the callee by the proper Observer instance. The notification message contains also the offer session description, i.e., the set of media streams and codecs the caller wishes to use, as well as the IP addresses and ports the caller would like to use to receive the media [57]. For the sake of conciseness, we don't show the use of the ICE protocol [56] for NAT traversal, which is recommended in the WebRTC specifications [13]. 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: i) it parses the session offer and generates the answer; and ii) 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.

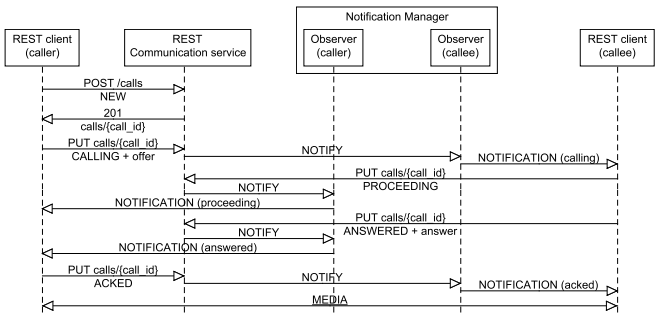

Fig. 11 Call setup between two web browsers.

Fig. 12 and Fig. 13 show an analogous message flow for a successful call setup between a web browser acting as the caller and a SIP UA 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 UA is handled by the REST-SIP Gateway (i.e., the SIPMessageSender and SIPMessageReceiver components) in compliance with the design specifications described in Section 3.

The coherence between the UML model and the prototype implementation has been verified through *cosimulation*, that is, running both on the model and on the implementation the test scenarios we have defined for

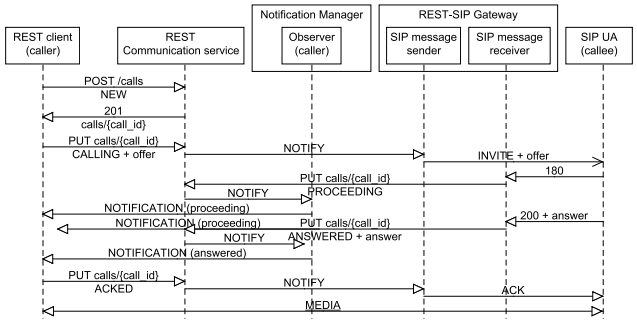

Fig. 12 Call setup between a web browser (caller) and a SIP UA (callee).

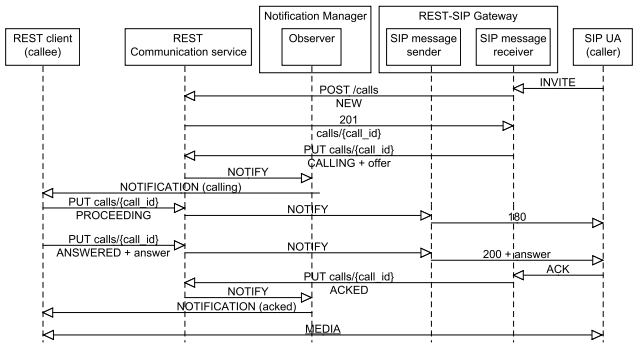

Fig. 13 Call setup between a SIP UA (caller) and a web browser (callee).

this purpose, such as those shown in Figs. 11, 12 and 13.

We checked that the interactions shown in the previous sequence diagrams were coherent with the evolution charts of the communicating state machines generated by the UMC tool for a successful call setup. Fig. 14 shows an excerpt of the chart representing the evolution of the state machine from the first POST request to a PUT PROCEEDING invocation for the first reference scenario. We found no substantial difference, therefore we can infer that the simulation as well as the implementation show the same computations, and hence that the latter satisfies the properties proved for the former.

4.7 Preliminary performance analysis

We performed a set of test iterations to evaluate the performance of our prototype. The objective of this preliminary analysis is to assess that the prototype implementation, which is consistent with the formalization, satisfies acceptable performance requirements in a simple scenario. We took as reference the maximum value for the call setup delay in Telecom/Web converged services environments, as defined by the TS 186 008-2 standard [63] (i.e., 8 seconds).

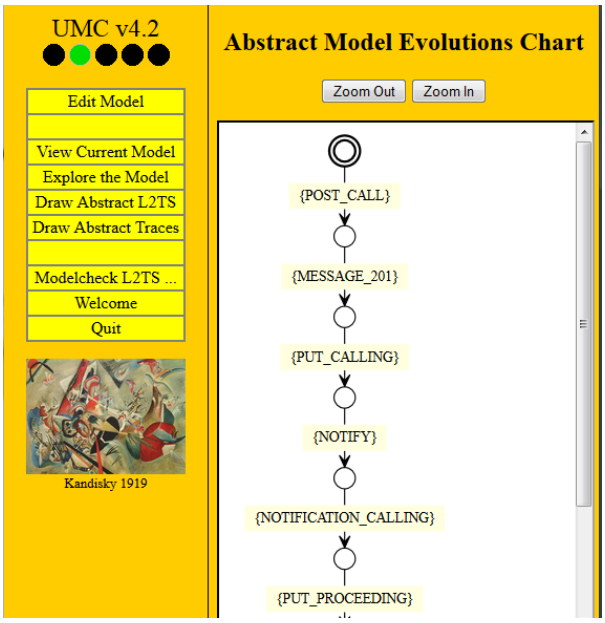

Fig. 14 Excerpt of the FSM evolution chart generated by the UMC tool for a call setup between two web browsers.

This experiment was aimed at evaluating the performance in terms of time delay in REST-to-REST and REST-to-SIP call scenarios. We used the following metrics: i) the *call setup delay*, defined as the time elapsing between the call setup request (POST HTTP message) and the reception of the final response (call ANSWERED notification); ii) the *subscription delay*, defined as the time between the delivery of the subscription request and the establishment of the notification channel; and iii) the *notification delay*, defined as the time between the occurrence of an event (i.e., the reception of a PUT request for changing a resource state) and the reception of the corresponding notification by the subscribed client.

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 the SIPp testing tool [62]. We chose to perform this experiment on a single machine in a laboratory environment to gather results on the delays due to processing tasks, while minimizing network delays. In order to simulate a configurable number of REST clients requesting a call setup, we developed a web application that allows to configure the number of calls to be initiated and the time delay between two consecutive calls. In this experiment, we configured the web application to simulate the initiation of 100 consecutive calls with 5 seconds of delay. The obtained results are presented hereafter.

The call setup between two web browsers required approximately 50 ms, where approximately 30 ms were due to the initial phase (PUT and POST invocation). A call setup between a web browser and a SIP UA required 110 ms, where 60 ms were required on average 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 cases is essentially due to time needed for processing the SIP messages and performing the corresponding REST invocation.

These results show how the average call setup delay in our system is comparable with analogous measures for call setup delay in SIP environments. For instance, the study by [38] reports an average call setup delay of 40 ms between two SIP UAs. The maximum call setup delay measured in our system is well below the acceptable limit of 8 seconds defined by the TS 186 008-2 standard [63].

Table 4 compares subscription and notification delays obtained by adopting the WebSocket and HTTP Streaming notification approaches. The subscription delay in the two cases has been measured in the following way: in the WebSocket case, the delay is the time measured at the client side between the delivery of the request to activate the WebSocket channel and the reception of the HTTP 101 response message; in the HTTP Streaming case, we measured the time interval between delivery of the subscription request and reception of the first HTTP response chunk. As shown in Table 4 the subscription delay with WebSocket is 7 ms on average, while the delay with HTTP Streaming is 13 ms. The notification delay is around 11 ms for both approaches.

Table 4 Subscription and Notification delays

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

A second experiment was aimed at measuring resources consumption in terms of CPU usage. In order to obtain more reliable measurements, the browser Google Chrome and the SIPp testing tool were located on a second machine. The two machines were connected via a 100 Mbps Ethernet/LAN. In the machine hosting the Tomcat Application Container and the web application we used JProfiler, which is a JVM profiler offering 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 consecutive calls. First, all users send a presence registration request to the server, and then they initiate a call to a SIP UA, one after the other with a time interval of 5 seconds between two consecutive calls.

Figures 15 and 16 show the CPU load for the WebSocket and HTTP Streaming approaches, respectively.

In both cases, the CPU load has some peaks in the first time instants. This is due to the creation of presence resources occurring in the initial phase of the experiment. Then, the subsequent peaks are due to the processing of POST and PUT requests for the call setup, recurring at intervals of approximately 5 seconds. In the case of HTTP Streaming (Fig. 16), these peaks increase up to a CPU usage of 20% and then decrease to a level close to zero. With WebSocket, the application shows a CPU pattern usage with peaks up to 10% CPU usage, but with a minimum CPU usage that never drops below 5%. At the time of writing, available studies on WebSocket have only focused on network latency and throughput in reference scenarios characterized by continuously streamed data, as in the study by Pimentel and Nickerson [52]. Thus, it is not straightforward to compare our results with these studies.

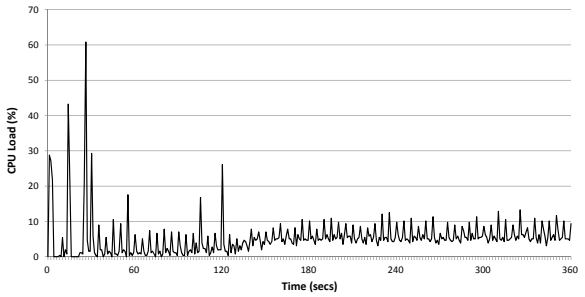

Fig. 15 CPU usage with the WebSocket notification approach.

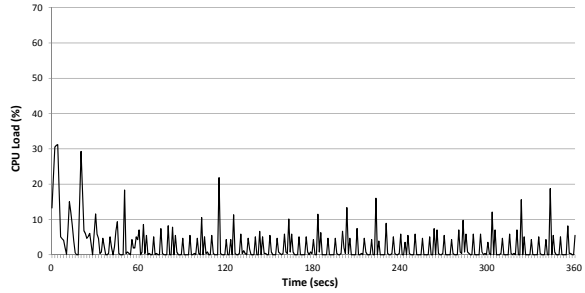

Fig. 16 CPU usage with the HTTP Streaming notification approach.

In a more accurate performance analysis it would be worthwhile to evaluate delays and resource consumption in more complex and realistic workload scenarios, which we plan to make in future work.

5 Conclusions

In this work, we have proposed an approach for the design and implementation of a REST API for real-time communication services based on formal modeling.

We have shown how, while extending a resource-oriented design methodology, we modeled call resource behavior through a Finite State Machine (FSM) representation. The adoption of a FSM representation (as UML State Diagrams) allowed us to use the analysis, exploration and model-checking capabilities offered by the UMC tool in order to: i) model the service as a set of communicating state machines; ii) simulate their behavior and interworking with client components (i.e., a REST client and a SIP UA); iii) verify some formal properties (deadlock absence, service responsiveness and resource connectedness).

Leveraging these specifications, we implemented a REST call service that can be invoked by any recent browser supporting WebSocket and WebRTC standards without requiring any additional code download. We evaluated the functional correctness as well as the performance of this prototype implementation.

According to the adopted REST principles, the exposed API supports stateless interaction, since every request from the client to the server contains all the information required to serve the request. The set of resources exposed through a uniform HTTP-based interface can be extended to offer other similar services, such as instant messaging and video conference.

We also discussed compliance with the HATEOAS constraint. This means that at each interaction step the client is provided with the options that are permitted at that point. In particular, at each transition the client receives the instructions on the possible next steps from the server. Therefore, the dynamic behavior of the caller and callee agents is embedded in the client-side application logic, but the possible next transitions are sent by the server. This promotes loose coupling between client and server-side implementations, while guaranteeing that the client behaves coherently with the application resource's state machine. This advantage is more evident if we compare this approach with legacy SIP UAs, which implement the client and server state machines defined in the SIP specifications.

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

Thanks to the results of a preliminary performance evaluation, we showed that, although a rigorous modelling approach has been adopted, the implemented prototype satisfies performance requirements. In the near future we will perform a more accurate performance analysis in realistic scenarios. Moreover, it would be interesting to extend the proposed approach and related implementation to expose more complex services, such as a videoconference service.

Although our work focused on real-time communication services, the proposed approach could be applied in other application domains to the generalized problem of the design of REST services based on the HTTP protocol and their interworking with external systems through other application protocols. The extensibility of the approach to other domains is straightforward if protocol interworking can be hindered by a subset of the mistmaches covered in this work (i.e., stateless versus stateful interaction, client-server versus peer-oriented paradigm).

The adoption of the state machine formalism allowed us to exploit the analysis, exploration and model-checking capabilities of the UMCtool to evaluate the compliance of the implemented prototype behavior with the communicating state machine model. In the near future, we are planning to extend this study towards formal specification of RESTful services. More specifically, we will extend the set of formal properties that express desirable attributes of our model, and 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 automatic code generation from the model and automatic test generation, again from the model. These two functionalities can be found, for what concerns UML tools, in the commercial IBM Rhapsody tool (that on the other hand does not provide model checking capabilities) [33]. We did not resort to automatic code generation for the prototype implementation, but rather we have directly implemented the solution, using the extracted computations as a guidance and reference to verify the correct working of the prototype.

Automatic code generation from the model would have allowed to formally relate the model and implementation, ensuring that properties proved on the model hold for the implementation as well. In this paper we have limited ourselves to check conformance of the man-

ually written implementation w.r.t. the model by means of computation traces inclusion. Indeed, the "Draw Abstract Traces" feature of UMC provides an automaton that generates all the computation traces of the model. Navigating this automaton we can produce test scenarios that can be replayed on the implementation: if the behaviour is the same, it means that the implementation's computations include the examined computation traces of the model. We have shown in Fig. 14 an example of conformance between model traces and test scenarios.

The combined use of UMC and Rhapsody could be used for further exploring the possibilities offered by the integration of their features towards the rigorous design of web-based communication signaling and interworking.

Acknowledgements The authors acknowledge the technical support by Luca Capannesi and Lorenzo Mazzi, from the University of Firenze.

References

1. 3GPP (2009) Open Service Access (OSA); Parlay X web services; Part 1: Common. 3GPP TS 29.199-01
2. Agarwal S (2012) Real-time web application roadblock: Performance penalty of html sockets. In: 2012 IEEE International Conference on Communications (ICC), pp 1225–1229, DOI 10.1109/ICC.2012.6364271
3. Aijaz F, Ali S, Chaudhary M, Walke B (2009) Enabling high performance mobile web services provisioning. In: Vehicular Technology Conference Fall (VTC 2009-Fall), 2009 IEEE 70th, pp 1–6
4. Alarcon R, Wilde E, Bellido J (2011) Hypermedia-driven restful service composition. In: Service-Oriented Computing, Springer, pp 111–120
5. Alvestrand H (2013) Real Time Protocols for Browser-based Applications. Internet-Draft, IETF, URL http://tools.ietf.org/html/draft-ietf-rtcweb-overview-08
6. Amirante A, Castaldi T, Miniero L, Romano SP (2013) On the seamless interaction between webRTC browsers and SIP-based conferencing systems. Communications Magazine, IEEE 51(4):42–47
7. Amirante A, Castaldi T, Miniero L, Romano SP (2014) Janus: A general purpose webrtc gateway. In: Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications, ACM, New York, NY, USA, IPTComm '14, pp 7:1–7:8, DOI 10.1145/2670386.2670389, URL http://doi.acm.org/10.1145/2670386.2670389
8. Bai Y, Ye X, Ma Y (2011) Formal modeling and analysis of sip using colored petri nets. In: Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on, pp 1–5, DOI 10.1109/wicom.2011.6040445
9. Becucci M, Fantechi A, Giromini M, Spinicci E (2005) A comparison between handwritten and automatic generation of c code from sdl using static analysis. Software: Practice and Experience 35(14):1317–1347, DOI 10.1002/spe.673, URL http://dx.doi.org/10.1002/spe.673
10. ter Beek M, Fantechi A, Gnesi S, Mazzanti F (2008) An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In: Leue S, Merino P (eds) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol 4916, Springer Berlin Heidelberg, pp 133–148, DOI 10.1007/978-3-540-79707-4_11, URL http://dx.doi.org/10.1007/978-3-540-79707-4_11
11. ter Beek MH, Mazzanti F, Gnesi S (2009) CMC-UMC: A Framework for the Verification of Abstract Service-oriented Properties. In: Proceedings of the 2009 ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '09, pp 2111–2117, DOI 10.1145/1529282.1529751, URL http://doi.acm.org/10.1145/1529282.1529751
12. Belqasmi F, Glitho R, Fu C (2011) Restful web services for service provisioning in next-generation networks: a survey. Communications Magazine, IEEE 49(12):66–73
13. Bergkvist A, D C Burnett CJ, Narayanan A (2012) WebRTC 1.0: Real-time Communication Between Browsers. W3C Working Draft, W3C, URL http://www.w3.org/TR/webrtc/
14. Bond G, Cheung E, Fikouras I, Levenshteyn R (2009) 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, ACM, New York, NY, USA, IPTComm '09, pp 13:1–13:12, DOI 10.1145/1595637.1595654, URL http://doi.acm.org/10.1145/1595637.1595654
15. C Bizer TBL T Heath (2009) Linked data - the story so far. Journal on Semantic Web and Inf Systems 5(3):1–22
16. Chen N, Chen Z, Zheng X, Chen G (2013) Mobile cloud based system architecture for remote-resident multimedia discovery and access. In: Web Information System and Application Conference (WISA), 2013 10th, pp 361–364

17. Chou W, Li L, Liu F (2008) Web services for communication over IP. Communications Magazine, IEEE 46(3):136–143, DOI 10.1109/MCOM.2008.4463784
18. Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2):244–263
19. Davids C, Johnston A, Singh K, Sinnreich H, Wimmreuter W (2011) 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, ACM, New York, NY, USA, IPTcomm '11, pp 2:1–2:7, DOI 10.1145/2124436.2124439, URL http://doi.acm.org/10.1145/2124436.2124439
20. Ding LG, Liu L (2008) Applications and Theory of Petri Nets: 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, chap Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets, pp 132–151. DOI 10.1007/978-3-540-68746-7_12, URL http://dx.doi.org/10.1007/978-3-540-68746-7_12
21. Erl T (2007) SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Prentice Hall PTR, Upper Saddle River, NJ, USA
22. Fette I, Melnikov A (2011) The WebSocket Protocol. RFC 6455, URL http://tools.ietf.org/rfc/rfc6455.txt
23. Fielding R (2000) Architectural Styles and the Design of Network-Based Software Architectures. PhD thesis, Architectural Styles and the Design of Network-Based Software Architecture
24. Fielding RT (2008) REST API must be hypertext driven. URL http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
25. Fielding RT, Taylor RN (2002) Principled design of the modern web architecture. ACM Transactions on Internet Technology (TOIT) 2(2):115–150
26. Fu C, Belqasmi F, Glitho R (2010) RESTful web services for bridging presence service across technologies and domains: an early feasibility prototype. Communications Magazine, IEEE 48(12):92–100, DOI 10.1109/MCOM.2010.5673078
27. Griffin D, Pesch D (2007) A Survey on Web Services in Telecommunications. Communications Magazine, IEEE 45(7):28–35, DOI 10.1109/MCOM.2007.382657
28. Griffin K, Flanagan C (2011) Defining a Call Control Interface for Browser-based Integrations Using Representational State Transfer. Comput Commun 34(2):140–149, DOI 10.1016/j.comcom.2010.03.029, URL http://dx.doi.org/10.1016/j.comcom.2010.03.029
29. Hameseder K, Fowler S, Peterson A (2011) Performance analysis of ubiquitous web systems for smartphones. In: Performance Evaluation of Computer Telecommunication Systems (SPECTS), 2011 International Symposium on, pp 84–89
30. Handley M, Jacobson V (1998) SDP: Session Description Protocol. RFC 2327, URL http://www.ietf.org/rfc/rfc2327.txt
31. Holzmann G (2003) Spin Model Checker, the: Primer and Reference Manual, 1st edn. Addison-Wesley Professional
32. Huang M, Zhu L (2012) Research for Network Fault Real-time Alarm System Based on Pushlet. In: Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on, pp 212–215, DOI 10.1109/ICICEE.2012.63
33. IBM Rhapsody (2013) URL http://www-03.ibm.com/software/products/en/ratirhapfami
34. Imre G, Mezei G (2016) Introduction to a websocket benchmarking infrastructure. In: 2016 Zooming Innovation in Consumer Electronics International Conference (ZINC), pp 84–87, DOI 10.1109/ZINC.2016.7513661
35. Islam S, Gregoire J (2013) Converged access of IMS and web services: A virtual client model. Network, IEEE 27(1):37–44
36. Ivanov I (2008) Mobicents Communication Platform. URL http://www.mobicents.org/index.html
37. Juneau J (2013) New Servlet Features. In: Introducing Java EE 7, Apress, pp 1–14, DOI 10.1007/978-1-4302-5849-0_1, URL http://dx.doi.org/10.1007/978-1-4302-5849-0_1
38. Kellokoski J, Tukia E, Wallenius E, Hamalainen T, Naarmala J (2010) Call and messaging performance comparison between IMS and SIP networks. In: Internet Multimedia Services Architecture and Application(IMSAA), 2010 IEEE 4th International Conference on, pp 1–5, DOI 10.1109/IMSAA.2010.5729396
39. Li L, Chou W (2010) Design Patterns for RESTful Communication Web Services. In: Web Services (ICWS), 2010 IEEE International Conference on, pp 512–519
40. Li L, Zhang X (2012) Research on the integration of RTCWeb technology with IP multimedia subsystem. In: Image and Signal Processing (CISP), 2012

5th International Congress on, IEEE, pp 1158–1161
41. Lozano D, Galindo LA, García L (2008) 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, IEEE Computer Society, Washington, DC, USA, NGMAST '08, pp 18–24, DOI 10.1109/NGMAST.2008.97, URL http://dx.doi.org/10.1109/NGMAST.2008.97
42. Mazzanti F (2015) UMC model checker. URL http://fmt.isti.cnr.it/umc/V4.11/umc.html
43. Menkens C, Wuertinger M (2011) From service delivery to integrated SOA based application delivery in the telecommunication industry. J Internet Services and Applications 2(2):95–111
44. Moriya T, Akahani J (2010) Application programming gap between telecommunication and internet. Comm Mag 48(8):96–102
45. Mulligan C (2009) Open API standardization for the NGN platform. Communications Magazine, IEEE 47(5):108–113, DOI 10.1109/MCOM.2009.4939285
46. Nicolas G, Sbata K, Najm E (2011) 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, ACM, New York, NY, USA, iiWAS '11, pp 98–105, DOI 10.1145/2095536.2095555, URL http://doi.acm.org/10.1145/2095536.2095555
47. OMA (2006) OMA Web Services Enabler OWSER Core Specification. Approved Version 1.1, Open Mobile Alliance
48. OMA (2012) Enabler Release Definition for RESTful bindings for Parlay X Web Services. Tech. Rep. V2, Open Mobile Alliance, 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
49. OMA (2015) RESTful Network API for WebRTC Signaling. Tech. Rep. V1.0, Open Mobile Alliance
50. Paganelli F, Turchi S, Giuli D (2014) A web of things framework for RESTful applications and its experimentation in a smart city. Systems Journal, IEEE PP(99):1–12, DOI 10.1109/JSYST.2014.2354835
51. Parastatidis S, Webber J, Silveira G, Robinson IS (2010) The role of hypermedia in distributed system development. In: Proceedings of the First International Workshop on RESTful Design, ACM, New York, NY, USA, WS-REST '10, pp 16–22, DOI 10.1145/1798354.1798379, URL http://doi.acm.org/10.1145/1798354.1798379
52. Pimentel V, Nickerson B (2012) Communicating and Displaying Real-Time Data with WebSocket. Internet Computing, IEEE 16(4):45–53, DOI 10.1109/MIC.2012.64
53. Porres I, Rauf I (2011) Modeling behavioral restful web service interfaces in uml. In: Proceedings of the 2011 ACM Symposium on Applied Computing, ACM, pp 1598–1605
54. Qadir J, Hasan O (2014) Applying formal methods to networking: Theory, techniques and applications. Communications Surveys Tutorials, IEEE PP(99):1–1, DOI 10.1109/COMST.2014.2345792
55. Richardson L, Ruby S (2007) RESTful Web Services. OReilly & Associates
56. Rosenberg J (2010) Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocol. RFC 5245, URL http://www.ietf.org/rfc/rfc5245.txt
57. Rosenberg J, Schulzrinne H (2002) An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), URL http://www.ietf.org/rfc/rfc3264.txt
58. Rosenberg J, Schulzrinne H, Camarillo G, Johnston A, Peterson J, Sparks R, Handley M, Schooler E (2002) SIP: session initiation protocol. RFC 3261, URL http://www.ietf.org/rfc/rfc3261.txt
59. Sege P, Palch P, Papn J, Kubina M (2014) The integration of webrtc and sip: Way of enhancing real-time, interactive multimedia communication. In: Emerging eLearning Technologies and Applications (ICETA), 2014 IEEE 12th International Conference on, pp 437–442, DOI 10.1109/ICETA.2014.7107624
60. Singh K, Krishnaswamy V (2013) A case for sip in javascript. IEEE Communications Magazine 51(4):28–33, DOI 10.1109/MCOM.2013.6495757
61. Sinnreich H, Wimmreuter W (2010) Communications on the web. e & i Elektrotechnik und Informationstechnik 127(6):187–194, DOI 10.1007/s00502-010-0742-l, URL http://dx.doi.org/10.1007/s00502-010-0742-l
62. SIPp (2012) SIPp: Open Source test tool/traffic generator for the SIP protocol. URL http://sipp.sourceforge.net/
63. Vingarzan D, et al (2007) IMS/NGN Performance Benchmark Part 2: Subsystem Configurations and Benchmarks. URL http://webapp.etsi.org/workprogram/Report_

WorkItem.asp?WKI_ID=25501, eTSI/TISPAN 6 Workitem 06024-2

64. Zave P (2008) Understanding sip through model-checking. In: Schulzrinne H, State R, Niccolini S (eds) Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks, Lecture Notes in Computer Science, vol 5310, Springer Berlin Heidelberg, pp 256–279, DOI 10.1007/978-3-540-89054-6_13, URL http://dx.doi.org/10.1007/978-3-540-89054-6_13
65. Zuzak I, Budiselic I, Delac G (2011) A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems. J Web Eng 10(4):353–390