A Methodology for SIP and SOAP Integration Using Application-Specific Protocol Conversion

GORAN DELAC, IVAN BUDISELIC, and IVAN ZUZAK, University of Zagreb
IVAN SKULIBER and TOMISLAV STEFANEC, Ericsson Nikola Tesla

15

In recent years, the ubiquitous demands for cross-protocol application access are driving the need for deeper integration between SIP and SOAP. In this article we present a novel methodology for integrating these two protocols. Through an analysis of properties of SIP and SOAP we show that integration between these protocols should be based on application-specific converters. We describe a generic SIP/SOAP gateway that implements message handling and network and storage management while relying on application-specific converters to define session management and message mapping for a specific set of SIP and SOAP communication nodes. In order to ease development of these converters, we introduce an XML-based domain-specific language for describing application-specific conversion processes. We show how conversion processes can be easily specified in the language using message sequence diagrams of the desired interaction. We evaluate the presented methodology through performance analysis of the developed prototype gateway and high-level comparison with other solutions.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.4 [**Distributed Systems**]: Distributed Applications; H.3.5 [**Online Information Services**]: Web-Based Services

General Terms: Design, Languages

Additional Key Words and Phrases: Protocol conversion, SIP, SOAP, design concepts, middleware, specialized application languages

ACM Reference Format:
Delac, G., Budiselic, I., Zuzak, I., Skuliber, I., and Stefanec, T. 2012. A methodology for SIP and SOAP integration using application-specific protocol conversion. ACM Trans. Web 6, 4, Article 15 (November 2012), 28 pages.
DOI = 10.1145/2382616.2382618 http://doi.acm.org/10.1145/2382616.2382618

1. INTRODUCTION

Rapid development of Internet infrastructure and businesses in the last decade has been accompanied by emergence of numerous diverse services. Despite the differences in technologies used for service development, growing user demands are pushing for service personalization and integration. Various Internet protocols for service consumption are converging as Web developers strive to integrate services across technology domains. A good example of such an effort is the integration of the Session Initiation Protocol (SIP) and H.323 Internet Telephony signaling protocols [Ho et al.

This work is supported by the Ministry of Science, Education, and Sports of the Republic of Croatia through the Computing Environments for Ubiquitous Distributed Systems (036-0362980-1921) research project.
Authors' addresses: G. Delac (corresponding author), I. Budiselic, and I. Zuzak, Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, 10000 Zagreb, Croatia; email: goran.delac@fer.hr; I. Skuliber and T. Stefanec, Ericsson Nikola Tesla, Krapinska 45, 10000 Zagreb, Croatia.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permission may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1559-1131/2012/11-ART15 $15.00
DOI 10.1145/2382616.2382618 http://doi.acm.org/10.1145/2382616.2382618

2001]. The proposed architecture introduces a converter which translates between SIP and H.323 operations enabling multiparty video conferencing. Another interesting service integration example is SIP-XMPP protocol conversion [Saint-Andre et al. 2007], which enables interworking between instant messaging systems that adhere to RFC 2779 [Day et al. 2000]. In both cases, the key to building an interoperable system across different protocol platforms is to properly define a mapping between semantically equivalent protocol elements. However, intuitive protocol conversion is possible only if both protocols share a certain amount of functionality. For example, SIP and H.323 are highly compatible for protocol conversion since they share the same purpose as signaling protocols for Internet Telephony services.

The aim of this article is to address protocol conversion issues for application-layer protocols. Application-layer protocols usually lack semantic equivalence due to significant differences in their purpose. Specifically, the presented research focuses on exploring integration possibilities of SIP and SOAP.

SIP [Rosenberg et al. 2002] is an application-layer textual protocol designed for establishing and managing sessions. The protocol is used in the 3GPP's IP Multimedia Subsystem (IMS), an architectural framework for delivering IP services in telecommunications networks. SIP itself does not define how IP services are created or how to perform semantically proper data exchange among them. In practice, it is most commonly used to control real-time multimedia sessions such as Internet Telephony calls, video conferences, and instant messaging. SIP provides a flexible platform for implementing various types of sessions by working in conjunction with other protocols. In other words, SIP offers primitives that can be used to construct various services accessible over the Internet. For example, SIP is used as a signaling protocol for VoIP [Rosenberg and Shockey 2000] and therefore serves as a backbone for the Internet Telephony service. Basic SIP primitives include management of user presence and location, and establishment and management of service sessions. Recent SIP integration efforts include an HTML5 client *simpl5* [Doubango 2012] which enables HTML5-compliant browsers to connect to SIP or IMS networks. Using this integration it is possible to make and receive audio/video calls or utilize instant messaging from the browser.

Widely accepted in enterprise business solutions, the Web Services (WS) [Curbera et al. 2002] protocol stack is emerging as a de facto standard used for exposing services. The communication with WS services is based on SOAP, a request-response RPC message exchange protocol [W3C 2007a]. SOAP messages are XML-based documents transferred using HTTP or SMTP protocols. WSDL documents describe the structure of SOAP messages for a specific service and therefore enable WS services to be loosely coupled with the client side [W3C 2007b]. However, the basic WS protocol stack does not address signaling issues, meaning there are no formal rules for creating and maintaining sessions and transactions. Proposals to add certain aspects of transactions and signaling into the WS standard exist. These include WS Composite Application Framework (WS-CAF) [OASIS 2011] and WS Choreography Description Language [Kavantzas et al. 2005]. However, these proposals only provide guidelines for implementing transaction and signaling management and are not widely accepted in the industry.

One of the key motives for the integration of SIP and SOAP can be found in existing infrastructure that employs these protocols. On one hand, the IMS architectural framework is one of the cornerstones of telecommunication networks. IMS offers standardized solutions for charging, billing, and presence—functionalities that are important for commercial services, especially mobile ones. However, IMS does not standardize means of exposing generic, Internet-based services. On the other hand, the WS protocol stack is designed specifically for exposing services on the

Internet, but lacks standardized and widely accepted means for managing sessions and transactions which are the basis for charging and billing. Therefore, it is clear that SIP and SOAP complement each other with their inherent functionalities. Furthermore, since both protocols are already widely accepted and used, SIP and SOAP should be integrated in a manner that requires little or no change to any existing communication nodes. For example, a SIP phone should be able to get data from an e-Health Web Service that exposes a patient's blood pressure to the doctor. Conversely, it should be possible to receive SOAP messages from a SIP-based event notification system. To meet this goal, in this article, we introduce a new intermediary node: a gateway that transparently translates messages between SIP and SOAP protocols.

The core component of this gateway is a SIP/SOAP protocol conversion process. In general, there are two basic steps in constructing a protocol conversion process. The first step is deriving a set of mapping rules that represent the conversion process, while the second step is implementing these mapping rules. However, as we show in this article, due to fundamental differences between SIP and SOAP, it is not possible to define a general set of mapping rules that could be used to integrate these protocols for a large number of applications. For example, the mapping rules used to integrate a SOAP-based chat bot service and a SIP instant messaging client differ from those used to integrate a SIP phone and a sensor network exposed through SOAP. Beyond the obvious syntactic differences, these mapping rules must properly maintain the semantics of messages, and those semantics are inherently application specific. To address this issue, the presented SIP/SOAP gateway is capable of executing various protocol conversion processes implemented as plugins. In the scope of this article, a *conversion plugin* is a computer process that defines message mapping rules for a specific set of SIP and SOAP communication nodes. The aggregate functionality of all these nodes and the conversion plugin governing their interaction is called a *conversion application* or just *application*. To ease development of conversion plugins, we introduce an XML-based language PCCL (Protocol Conversion and Coordination Language) [Budiselic et al. 2007]. Such an approach enables mapping rules, described by PCCL, to be automatically translated into an implementation of the conversion plugin. Writing a PCCL definition is the only required step for adding an application to the gateway. PCCL was designed to support rapid development based on message sequence diagrams of application use cases.

The contribution of this article is a methodology for integrating SIP and SOAP protocols using application-specific converters. The presented methodology is founded on three ideas. First, a considerable portion of SIP/SOAP integration, such as parsing and composing messages as well as network and storage management, can be reused for all applications. On the other hand, we argue that it is impossible to define general mapping rules for SIP and SOAP messages, so these mapping rules must be defined in an application-specific manner. We describe a generic SIP/SOAP protocol conversion gateway that implements all the reusable functionality and relies on application-specific conversion plugins to drive each individual application. Second, defining conversion plugins should be done using a Domain-Specific Language (DSL) because developing conversion plugins in general-purpose languages is tedious and requires a lot of boilerplate code distracting from the actual conversion specification. Towards this goal, we describe PCCL, an XML-based language for defining application-specific rules for SIP/SOAP integration. PCCL specifications are translated into application-specific conversion plugins that are used by the generic SIP/SOAP gateway to run the application. Third, the domain-specific language should closely model message sequence diagrams of an application since its primary concern is message mapping. We designed PCCL according to this idea and we present a method for writing PCCL definitions using message sequence diagrams for the application's use cases as design aids.

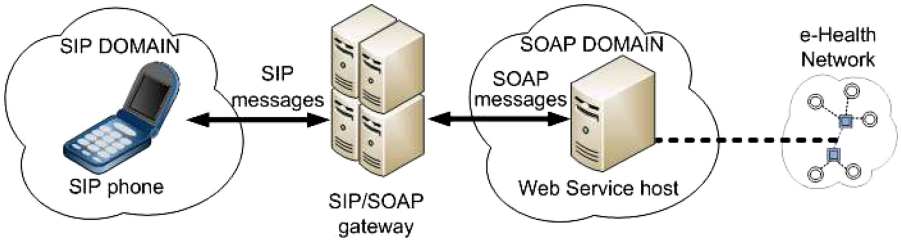


Fig. 1. E-Health SIP/SOAP integration use case.

The rest of the article is structured as follows. Section 2 presents a SIP/SOAP integration use case that is used in examples throughout the work. Section 3 is devoted to fundamental theoretical background behind protocol conversion. Protocol projection theory is used to demonstrate how protocol mismatches influence the potential for constructing a conversion process. In Section 4 we analyze protocol mismatches present in integration of SIP and SOAP and present a set of requirements for PCCL and the conversion gateway. The Protocol Conversion and Coordination Language is described in Section 5. Section 6 presents a description of the generic SIP/SOAP gateway architecture and explains how PCCL specifications get executed by the gateway. In Section 7 we present how conversion applications can be defined in PCCL using message sequence diagrams as design aids. In Section 8 we discuss the performance characteristics of the presented methodology based on the results of experiments we performed on the prototype implementation of the SIP/SOAP gateway. Section 9 surveys related work in SIP/SOAP integration. Finally, concluding remarks and directions for further research are given in Section 10.

2. A SIP/SOAP INTEGRATION USE CASE

In this section we briefly describe an e-Health [Cubic et al. 2010] use case for SIP/SOAP integration that we will reference in the remainder of the article. This example uses the SIP extension for event notification [Roach 2002] to allow doctors to use a SIP phone to access patients' blood pressure and pulse that are exposed through a Web service.

Figure 1 presents the high-level architecture of the example system. The e-Health network monitors pulse and blood pressure of several patients and exposes its data through a Web service. The premise of the use case is that the SIP phone does not have the Web services protocol stack implemented and therefore cannot access the Web service directly. Similarly, the Web service host does not have a SIP implementation and therefore cannot be communicated with directly by SIP clients. To enable the SIP phone to access sensor data through the Web service, the SIP/SOAP gateway is used as a protocol converter between the SIP and SOAP domain.

The Web service exposes two methods. Both methods have a *patientId* parameter used to identify a specific patient. The *getData* method is used for one-time data pull. The method has no additional parameters and returns the patient's pulse and blood pressure values in an XML structure in the body of the SOAP response message. The *setSubscriptionStatus* method has three additional parameters that specify the subscription duration, value thresholds, and a Web service callback method and returns a token identifying the subscription. If the pulse or blood pressure of the patient reaches the specified threshold within the subscription duration, the Web service will notify the subscribed client with the current data values and the subscription token via the Web service callback method specified as an argument to *setSubscriptionStatus*.

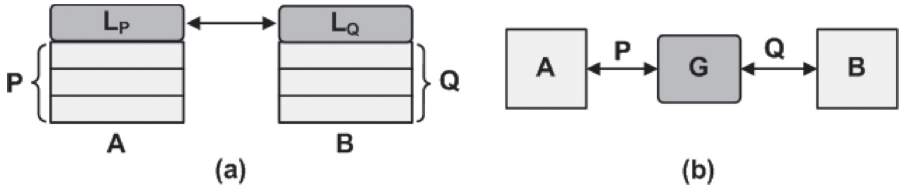


Fig. 2. Two ways of implementing a conversion mechanism for two processes using different protocols.

The SIP/SOAP gateway mirrors this SOAP interface to its SIP client. To request one-time data retrieval, that is, an invocation of *getData*, the SIP client sends a SIP SUBSCRIBE message with the *Expires* header set to the number one to the gateway. Any other value of the *Expires* header is used as the subscription duration in minutes for an invocation of *setSubscriptionStatus*. Further details about the use case are explained in the following sections when they are required to describe a feature of PCCL or the SIP/SOAP gateway.

3. PROCESS INTEROPERABILITY

Process interoperability theory defines two processes that communicate by exchanging messages as interoperable if they implement the same communication protocol [Lam 1988]. This implies that each process is capable of understanding syntax and semantics of messages it receives from the other process. To achieve interoperability between two communicating parties that do not implement the same message exchange protocol, an additional conversion mechanism needs to be established. The conversion mechanism can be implemented in two distinct ways as shown in Figure 2. The first approach is to construct an additional layer atop the existing protocol stacks which would facilitate the conversion, as shown in Figure 2(a). Processes A and B, that implement protocol stacks P and Q, respectively, communicate by implementing additional layers L_P and L_Q. This approach requires that both communicating processes be extended with additional functionality in order to become interoperable.

The alternative approach in building a converter is to implement the conversion mechanism as a separate, third process. The converter then operates as a gateway between processes by performing protocol conversion, as presented in Figure 2(b). Processes A and B, that implement protocols P and Q, respectively, communicate by exchanging messages through the converter process G. The benefit of this approach is that the existing systems do not need to be modified in order to become interoperable. Furthermore, since the conversion logic is centralized, modifications to the conversion process do not affect the communication endpoints. However, in contrast to the decentralized approach presented in Figure 2(a), building a separate converter can induce scalability issues, since it can become the system's bottleneck. Furthermore, message exchange with the converter process creates greater communication overhead than in the decentralized approach.

Regardless of the implementation, the core purpose of the conversion process is to translate between messages of various protocols. A conceptual overview of the conversion process, common for both implementation approaches, is shown in Figure 3. Each protocol is defined by its state space, message space and state transitions. The state space is the set of all states that a process implementing a certain protocol can transition to during its operation. The message set contains all the messages a process can receive or send, while the state transitions define how the process changes states depending on the received or sent messages. State transitions occur only when a message is received or sent out. For example, processes A and B, presented in Figure 3, implement message exchange protocols P and Q, respectively. Protocol P is defined by

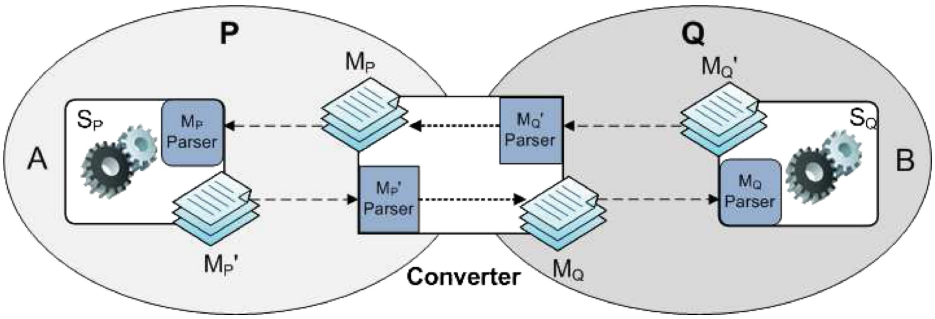


Fig. 3. Conceptual view of protocol conversion.

its state space S_P and message sets M_P and M_P'. M_P is the set of all possible incoming messages, while M_P' is the set of all outgoing messages. Similarly, protocol Q is defined by S_Q, M_Q and M_Q'. During its operation, process A receives messages from M_P, interprets them, changes states accordingly, and sends out response messages from M_P'. Similar actions during communication are performed by process B. To achieve interoperability of processes A and B, which do not implement the same protocol, a protocol conversion process is used. The conversion process receives messages from sets M_P' and M_Q', interprets and translates them to M_Q and M_P, respectively. Apart from simply performing syntax changes, the converter needs to map messages from one set to semantically equivalent messages in the other set. Mapping messages to their semantic equivalents is the key issue of protocol conversion as many protocols greatly vary in their purpose which dictates message semantics. For example, application-layer protocols tend to have lower semantic equivalence since they are designed for diverse applications like Internet Telephony, instant messaging, or text transfer. On the other hand, data-layer protocols have a higher level of message equivalence, since their common purpose is to transfer data regardless of data semantics.

Therefore, the key research goal in protocol conversion is to formally define whether a useful protocol converter can be constructed for the given protocol pair. Although this field has been a subject of intensive research in the last couple of decades, no general solution methodology for protocol conversion is known. In Lam [1988], Calvert and Lam [1989], Okumura [1986], and Tao et al. [1995], the authors propose formal methods for protocol converter generation. These methods are used to generate converters by combining protocol state machines, that is, by searching for state machine intersections. The resulting conversion process is also described by a state machine that contains an intersection of the protocols' functionalities. However, successful generation of a converter depends on the aforementioned level of message equivalence, that is, existence of protocol mismatches. In Calvert and Lam [1990], the authors define two types of protocol mismatches: hard and soft mismatches. For two protocols that have hard mismatches, a general converter that retains any useful functionality of the two protocols cannot be constructed. This means that processes that communicate using these two protocols cannot exchange messages in a meaningful way since one protocol lacks a crucial functionality of the other protocol. On the other hand, soft mismatches allow the construction of a general protocol converter. However, such a converter has only a subset of the functionality of both protocols. Despite this, for some protocol conversion cases, such reduced functionality proves to be an adequate solution.

In order to further describe how protocol mismatches affect protocol conversion, we will refer to the theory of protocol projection [Lam 1988], usually used in protocol

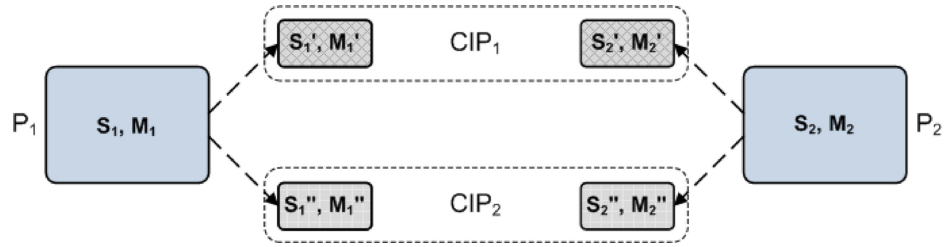


Fig. 4. Construction of two common image protocols for P_1 and P_2.

verification. Protocol projection is based on aggregating the functionality of a protocol in order to disregard behavior unimportant for proving a certain assertion. This is done by partitioning the state space of a protocol. Each state partition is aggregated into a single state in the projection. Similarly, the message set is partitioned into message sets that cause state transitions in the projection. This process is referred to as protocol aggregation. The newly created protocol with fewer states and messages is called an image protocol. The image protocol represents behavior of the original protocol on a higher level of abstraction, that is, some functionality is disregarded. Furthermore, a key property of this approach is that if an assertion holds for the image protocol, it also holds for the original protocol. Therefore, image protocols are more suitable for verifying certain functionalities of a protocol.

The concept of an image protocol is applicable to the problem of protocol conversion, where the goal is to find an image protocol common to both original protocols. Such an image protocol would contain a subset of the original protocols' functionalities. It is possible for two protocols to have more than a single common image protocol, as shown in Figure 4. State sets S_1' and S_1" are constructed by aggregating states from S_1 in two different ways. Similarly, states from S_2 of protocol P_2 are aggregated to form sets S_2' and S_2". The same aggregation rules hold for message sets M_1 and M_2. If state space S_1' is equal to S_2' then a common image protocol CIP_1 exists between P_1 and P_2. The same can be said for the common image protocol CIP_2. When integrating two protocols the goal is to find a common image protocol that retains as much of the original functionality as possible.

To measure the difference between an image protocol and its original, a simple concept of protocol resolution was introduced. The resolution is given by the size of message and state sets of an image protocol. The grater the number of states and messages defined by the protocol, the higher is its resolution. By definition, all image protocols have a lower resolution than that of their originals. Thus, the goal of protocol conversion is to find a common image protocol with the highest resolution. The maximum resolution of a common image protocol varies for diverse protocol pairs as they differ in the level of their semantic equivalence. The minimum resolution is constrained by the requirements of the desired protocol converter. If this resolution cannot be achieved, a hard mismatch exists between the two protocols for the given protocol integration case. On the other hand, if the desired protocol converter can be defined by disregarding functionalities that caused the mismatch, a soft mismatch between the two protocols exists.

When it is possible to find a common image protocol with a sufficiently high resolution and satisfactory functionality, that is, a soft mismatch exists, the solution to protocol conversion is to build a memoryless converter. By implementing the common image protocol, a memoryless conversion process translates messages of one protocol into semantically equivalent messages of the other protocol without performing

any additional functionality. An example of a memoryless converter is the SIP-XMPP gateway [Saint-Andre et al. 2007].

In case that protocol A does not have a desirable functional property of protocol B, the common image protocol does not have sufficient functionality to describe a conversion process between A and B, that is, a hard mismatch exists. In such cases, it still may be possible to construct a converter by extending the conversion logic with an additional state machine. However, such additional functionality cannot be derived by applying the aforementioned formal conversion methods. Protocol converters with extended functionality are referred to as finite-state converters. The built-in state machine implements functionality which protocol A lacks and therefore enables the conversion. For example, let the aforementioned protocol B require each message to be marked with a sequence number. If protocol A does not support messages marked with sequence numbers neither will any common image protocol of A and B. In that case, to achieve the conversion, additional functionality is required in order to append sequence numbers to messages of protocol A. The characteristic property of finite-state converters is that the set of messages sent by the converter has a higher resolution than the message set received by the converter.

4. INTEGRATION OF SIP AND SOAP PROTOCOLS

In this section, we discuss key challenges in integrating SIP and SOAP and show that a general protocol converter for all applications cannot be constructed because SIP and SOAP have a hard mismatch and there is no general mapping of semantically equivalent messages; instead, message mapping must be defined on a per-application basis. We present a set of requirements for SIP/SOAP integration based on a domain-specific language PCCL used to define application-specific message mapping and address the SIP/SOAP hard mismatch. Additionally, we present requirements for a gateway that performs protocol conversion between SIP and SOAP using plugins defined in PCCL.

4.1. Challenges in SIP/SOAP Integration

The integration of SIP and SOAP presents several challenges due to differences in the intended purpose of the two protocols. One of the core issues in SIP/SOAP conversion arises from the fact that SIP is a stateful protocol, while SOAP is stateless. Since SIP is designed for session management, SIP nodes are required to store session data, that is, state. The stored state usually includes authentication tokens and other session parameters, like QoS (Quality of Service) settings. Since session data allows SIP nodes to associate messages with corresponding sessions, it is essential for the nodes to maintain session state. The following example illustrates how differences in state management influence SIP/SOAP conversion.

In order to initiate a session, two SIP nodes must complete a three-way handshake by exchanging a session identifier. The session is initiated by sending an INVITE message (1) containing the session identifier, as shown in Figure 5(a). An OK message (2) is sent in reply to the INVITE message, signalizing that the node is ready to exchange data in the proposed session. Finally, the session is established by an ACK message (3). On the other hand, SOAP is a stateless protocol that conforms to SOA principles [Huhns and Singh 2005]. Therefore, data is exchanged between the communicating parties in a simple request-response message exchange pattern, as shown in Figure 5(b). This means that session initiation and termination messages cannot be mapped to SOAP messages; neither can the session identifier be exchanged prior to session establishment. Since a valid session identifier is a crucial parameter for data exchange in sessions, SOAP lacks a desired logical property of SIP, thus a hard mismatch exists for any given pair of SIP and SOAP nodes. In order to resolve this

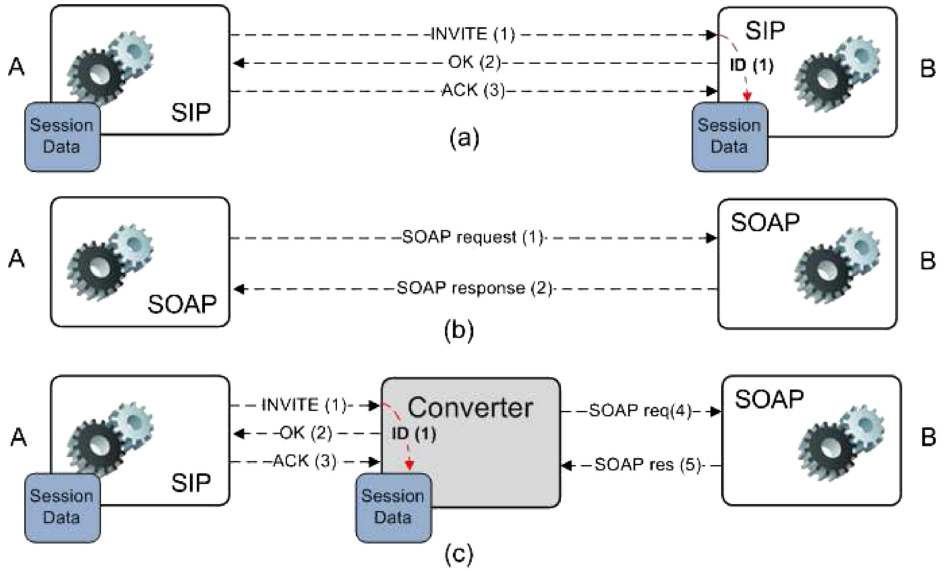


Fig. 5. Resolving the hard mismatch of SIP and SOAP.

mismatch, a SIP/SOAP converter needs to handle session management on behalf of the SOAP node; see Figure 5(c). This means that the converter is extended with an additional functionality, not common for both protocols, that is, a finite-state converter needs to be constructed.

Another conversion issue arises from the complexity of SIP and SOAP messages and determining a semantically equivalent message in one protocol given a message in the other. Since both SIP and SOAP are application-layer protocols, the syntax and semantics of the message content vary depending on the application functionality. In effect, the conversion middleware has to analyze the incoming message content and use it to construct valid responses. This is a much more complex conversion problem than for data-layer protocols since they only transfer data without analyzing it. This data is not part of the conversion process and is processed by higher-level protocols. SIP transfers plain unstructured text as payload, often embedded messages of other textual protocols such as HTML, XMPP, or even SOAP. In addition, SIP messages can be used to transfer various user-defined data structures stored as text. On the other hand, SOAP messages are XML-based structured documents. The syntax and semantics of SOAP messages depend on the functionality of the corresponding service.

Since the conversion process for SIP and SOAP must interpret the semantics of received messages, and these semantics are application specific, semantically equivalent SIP and SOAP messages cannot be defined for all applications and a general SIP/SOAP converter is not possible. Therefore, the set of message mapping rules must be defined specifically for any given SIP/SOAP conversion application.

4.2. Requirements for SIP/SOAP Integration

To address the challenges presented in the previous subsection, we designed PCCL. PCCL is a domain-specific language for specifying session management and message mapping rules for a SIP/SOAP application. The design of PCCL was guided by the following requirements. First, to support coordination of an arbitrary number of SIP and SOAP nodes, the DSL must allow the designer to specify URLs of all nodes

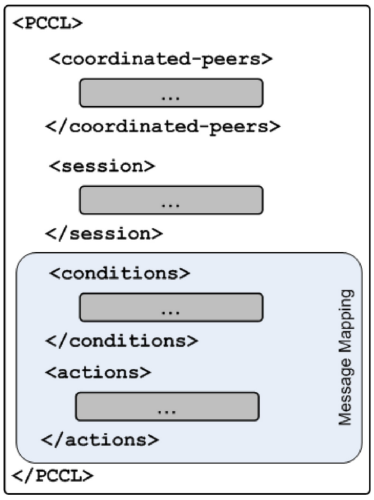


Fig. 6. PCCL specification structure.

and other properties required to communicate with each node. In PCCL terminology, SIP and SOAP communication nodes are called *peers*. Second, the DSL must provide constructs for managing sessions between the peers. Primarily, it must be possible to define session identifiers for both SIP and SOAP messages and store other session information required for a particular coordinated application. These session identifiers are used to associate incoming SIP and SOAP messages with a session and the corresponding session data stored in the gateway. Third, since semantically equivalent messages are determined by application specifics, the DSL must contain constructs for specifying message mapping between the protocols.

To support execution of conversion applications defined in PCCL, the design of the SIP/SOAP gateway was based on three key requirements. First, the gateway must be able to parse and compose SIP and SOAP messages and manage network resources for conversion applications. Specifically, the gateway must be able to receive and send SIP and SOAP messages. Second, to support session management defined for a conversion application in PCCL, the gateway must be able to store session data. Third, several conversion applications must be able to run simultaneously through the same gateway. Furthermore, it must be possible to add new PCCL definitions when a new conversion application is to be run, or remove them when they are no longer necessary without interrupting the system's operation.

5. SIP/SOAP CONVERSION LANGUAGE

As a consequence of SIP and SOAP protocol mismatches described in the previous section, and the fact that message mapping must be defined per application, conversion plugins require substantial implementation effort in general-purpose languages. To enable rapid development of conversion plugins we have designed an XML-based domain-specific language [Mernik et al. 2005] called PCCL for Protocol Conversion and Coordination Language. The conversion specification written in PCCL is automatically translated into a conversion plugin implementation.

The structure of a PCCL specification is modeled after the requirements presented in Section 4.2 and is shown in Figure 6. The *coordinated peers* element provides constructs for establishing logical identifiers for communication nodes that are integrated in the application and defining their parameters. The *session* element is used to handle

```
<coordinated-peers>
 <SIP name="phone">
  <address>phone.example.com</address>
  <port>5060</port>
 </SIP>
 <WS name="ws">
  <wsdl>
   http://ehealth.example.com/Service.asmx?WSDL
  </wsdl>
  <invokes methodName="getData"/>
  <invokes methodName="setSubscriptionStatus"/>
 </WS>
</coordinated-peers>
```

Fig. 7. The coordinated peers PCCL element for the e-Health use case.

the hard mismatch of SIP and SOAP in session management. This element specifies how session identifiers are created from message headers and content, and defines variables for storing state. The message mapping specification is separated into two sections: conditions and actions. The *conditions* element is used to control the flow of the application based on the headers and content of incoming messages. The *actions* element specifies output messages and state changes that will be carried out in response to the received message. The separation of conditions and actions simplifies conversion specification through reusability of defined rules. All the elements of PCCL are described in further detail in the following subsections. The semantics of PCCL are defined leveraging examples from the SIP/SOAP integration use case described in Section 2. A nearly complete listing of the PCCL specification of this use case is discussed in Section 7 and can be used as further reference for the PCCL language. Additional information on PCCL can be found in Budiselic et al. [2007].

5.1. Defining Coordinated Peers

PCCL provides constructs for integrating an arbitrary number of SIP and SOAP communication nodes. All the nodes in the application are defined in the coordinated peers section. Each node is assigned a logical identifier used to refer to it in the remainder of the specification. Additionally, all required information for communicating with the node is also listed in this section.

Every SIP node is defined by its FQDN or IP address and the port number. Since interaction with a Web service is described in its WSDL document, the URL of the Web service's WSDL is defined for every SOAP node in the application. The gateway retrieves the WSDL and uses it to create SOAP message templates. Additionally, the URL used to send messages to the Web service is extracted from the WSDL document. PCCL also provides means for restricting the interface of a Web service by specifying only some of the exposed method names.

The coordinated peers element for the use case from Section 2 is shown in Figure 7. In this use case there are only two communication nodes: a SIP phone and a Web service exposing patients' vital signs to doctors. The SIP node is described by a *SIP* PCCL element. In this example, the SIP phone is assigned the logical identifier *phone* through the *name* attribute of the *SIP* element. Furthermore, the *address* and *port* elements define the address and SIP listening port of the SIP node.

On the other hand, the Web service is defined by a *WS* element. The URL of the service's WSDL is defined in the *wsdl* element. The *invokes* elements are used to define the method names of the service that will be invoked by the gateway during the execution of the application. In this use case, both *getData* and *setSubscriptionStatus* will be used so they are each defined by an invokes element.

```
<session name="session">
 <id name="SipSessionId" peer="phone" location="SIP:header:call-id" type="string"/>
 <id name="SoapSessionId" peer="ws" location="SOAP:body:subscriptionToken"
  wsMethod="setSubscriptionStatus" type="string"/>
 <var name="patientId" type="string"/>
 <var name="sequenceSection" type="string"/>
 ...
</session>
```

Fig. 8. Session identifiers and variables for the e-Health use case.

5.2. Managing Session State

To resolve the hard mismatch between SIP and SOAP in regards to protocol-level session management, session identifiers for SIP and SOAP are defined in PCCL on a per-application basis. These session identifiers are used to correlate incoming messages to an existing session. In addition to session identifiers, the gateway can store arbitrary data that has to persist throughout the session in session variables. The PCCL *session* element is used to define both session identifiers and session variables. Session variables are referenced in other parts of the PCCL specification where they are set or read. The only difference between session identifiers and session variables is that session identifiers are updated automatically by the gateway as new messages are received, whereas session variables must be explicitly managed in other PCCL elements.

The *session* element contains an attribute *name* that is used to define a session namespace. Session identifiers are assigned to the namespace using the *id* PCCL element. The *name* attribute of the *id* element defines the name of the identifier in the namespace and the *peer* attribute refers to one of the logical identifiers defined in the coordinated peers section. Session identifier values are extracted from received messages. The *location* attribute is used to specify the location of the identifier value in the message. SOAP session identifiers also have a *wsMethod* attribute in the *id* element. The identifier value is extracted from the response message received by the gateway when it invokes the method specified by the *wsMethod* attribute. Session variables are defined by *var* elements with a *name* attribute specifying the variable's name in the namespace. Both *id* and *var* elements have a *type* attribute that defines the type of the identifier or variable and can be either *string*, *integer*, or *float*.

The SIP/SOAP integration use case described in Section 2 uses two session identifiers and several session variables, only two of which are shown in Figure 8 due to space constraints. This use case uses only one namespace called *session*. The SIP *call-id* header value is used as the SIP session identifier in the gateway. Since all SIP messages contain this header value and it is unique within a session, it is sufficient to correlate incoming SIP messages with the session.

However, as SOAP messages don't contain session identifiers, the value chosen for the SOAP session identifier depends on the specific application. In this SIP/SOAP integration use case, a session identifier for SOAP messages is only required for the callback from the Web service when a threshold is reached. All other SOAP messages received by the gateway are SOAP responses that are easily correlated with the SOAP request and therefore do not require a session identifier. Prior to receiving any such callback, the gateway will have had to invoke the *setSubscriptionStatus* method of the Web service. As described in Section 2, this method returns a subscription token, and this subscription token is then used in PCCL to define the SOAP session identifier.

The variable *patientId* is used to store the patient identifier from the received SIP request for the duration of the session. This stored value is then passed as an argument to the appropriate Web service method. The *sequenceSection* variable is used

```
<conditions>
 <protocol name="SIP">
  <message peer="phone" type="SUBSCRIBE">
   <if>
    <EQ left="header:Expires" right="1" type="int" />
    <exec>
     OneTimePull
    </exec>
   </if>
  </message>
 </protocol>
</conditions>
```

Fig. 9. The matching rule for the one-time data pull SIP SUBSCRIBE message in the e-Health use case.

to guide the conversion process as a state machine, as described in Section 7. The remaining session variables that are not shown here serve the same purpose as the *patientId* variable for other parameters of the request, such as threshold values and subscription duration.

5.3. Message Mapping

The message mapping section of a PCCL specification consists of a set of rules that define how output messages are generated depending on input messages. Mapping rules are divided into two sections: conditions and actions. Each message that satisfies a set of conditions is said to *match* the specification. The matching process depends on both the received message content and the current session state of the application running through the gateway. A matching message is assigned a set of actions which describe output messages that are to be sent out of the gateway and any session state changes. This concept is further explained in the following subsections.

5.3.1. Conditions. Message matching rules are defined inside the *conditions* element of the PCCL specification. The *conditions* element contains two *protocol* elements that define all the matching rules for messages of that protocol. The *protocol* element's *name* parameter indicates whether the protocol is SIP or SOAP. A parameter is used instead of element names to support extensibility of PCCL to other protocols. The SIP *protocol* element contains one *message* element for each SIP communication node and SIP method pair. Each *message* element has *peer* and *type* attributes that define the communication node and method, respectively. *Message* elements contain *if*, *elif*, and *else* elements used to control condition testing. These elements in turn contain various condition testing elements like *EQ* (equals), *GT* (greater than), and *LT* (less than). Condition testing elements have attributes *left* defining the left operand, of the comparison, *right* defining the right operand, and *type* defining the type of the operands. The *left* and *right* attributes can be data extracted from the message, session data, or a constant. If the condition is met, the list of actions specified in the following *exec* element will be carried out by the gateway.

A matching rule for the one-time data pull SIP SUBSCRIBE message is shown in Figure 9. The *EQ* condition testing element compares the *Expires* header from the SIP message with one. As explained in Section 2, an *Expires* header value of one indicates that the client is requesting a one-time data pull from the Web service. Therefore, the *OneTimePull* action will get executed. A more detailed listing of the same *protocol* element is shown in Section 7.

The SOAP *protocol* element contains a *message* element for each method name, communication node, and request/response triple. These parameters are specified through the *name*, *peer*, and *type* attributes. Condition matching for SOAP messages is the same as for SIP messages.

```
<get type="string" location="body">
  "?"$session.patientId${1}
</get>
```

Fig. 10. The *get* element for extracting the patient's identifier from a one-time data pull request.

```
<set name="session.sequenceSection" type="string"
     value="OTP-waiting-for-SOAP" />
```

Fig. 11. The *set* element used to update the *session.sequenceSection* variable.

5.3.2. Actions. PCCL actions are used to define output messages and session state changes in response to a received message. Each action presents a set of activates executed by the gateway after the message matching process is completed. Actions can perform three basic activities: extracting and storing session data from input messages, changing session state, and defining output messages.

Every PCCL action is defined inside an *action* element. The *action* element has *name* and *inType* attributes. The *name* attribute assigns a name to the action so that it can be referred to in *exec* elements of message matching rules. The *inType* attribute specifies the protocol of incoming messages that can cause this action to be executed and can be either SIP or SOAP.

Extraction and storing of session data from the input message is specified using *get* elements. Data is extracted from the location defined by the *location* attribute which can be either *header:header_element* or *body*. The extracted value can be further specified using a small custom language. The general form of *get* element contents is

"left_delim"{discard first n chars}$namespace.var${discard last n chars}"right_delim".

The first string that is exactly between the left and right delimiters at the specified location in the message is further processed by discarding characters from the front and the back and finally stored in the specified session variable. Any component except the variable name can be left out. Delimiters default to matching anything and no characters are discarded unless specified otherwise.

An example of a get element is shown in Figure 10. This *get* element extracts the patient's identifier from a one-time data pull request. The request parameters are stored in the first and only line of the body of the SIP message, and the patient's identifier is the last part of the line, preceded immediately by a question mark. The question mark is used as the left delimiter and the newline character is discarded using the language shown before. The extracted patient identifier is stored in the *session.patientId* variable.

The PCCL *set* element is used to set the value of a session variable to a constant. The *name* attribute defines the target variable, the *type* attribute defines the type of the variable, and the *value* attribute contains the value to which the variable will be set. This is primarily useful for controlling the value of a variable used to guide the conversion process as a state machine.

Figure 11 shows the *set* element used to update the *session.sequenceSection* variable after receiving the one-time data pull request. Since the gateway creates a SOAP request and sends it to the Web service to get the requested data, the next expected incoming message is the SOAP response from the Web service.

Finally, output messages are defined using *send* elements. Each action can create multiple output messages. The *protocol* attribute specifies weather the output message protocol is SIP or SOAP. SIP *send* elements have *peer* and *type* attributes that define the destination SIP node and SIP method. PCCL provides constructs to manipulate both header and body sections of the message. Header values are defined through *header* elements with *name* and *value* attributes. The body of the SIP output message

```
<send protocol="SOAP" peer="ws" name="getData" type="Request">
 <arg name="patientId" type="string">
  $session.patientId$
 </arg>
</send>
```

Fig. 12. SOAP *send* element for relaying the client's one-time pull data request to the Web service.

is defined by the *body* element. Both the body and header values can be constructed from session variable values and constants.

For SOAP output messages, the *send* element has *peer*, *name*, and *type* attributes that define the target Web service, invoked method name, and if the message is a request or a response. Arguments are inserted into the message using the *arg* element. Every *arg* element has *name* and *type* attributes. The argument value is specified inside the *arg* element as a constant or the value of a session variable. The gateway inserts the value into a message template generated from the service's WSDL.

The *send* element that generates the SOAP *getData* request is shown in Figure 12. Since the *getData* Web service method has only one parameter, only one *arg* element is used in PCCL. The parameter's name is *patientId* and the supplied value is the value stored in the session variable *session.patientId*. In effect, *session.patientId* is used as a temporary variable to copy the patient's identifier from the SIP request into the SOAP request message using a *get* and a *send* element.

Several complete *action* elements are shown in Figure 17 and are not repeated here to save space.

PCCL is constrained in terms of computation capability. Specifically, arithmetic on session variables isn't supported. In terms of language theory, PCCL defines a finite-state machine. Even though the message set size is unbounded and the session storage of the gateway can be in an unbounded number of states, every plugin can only specify a finite number of message matching rules. All internal states that don't match any specified condition are therefore equivalent to a dead state in the FSM. This design choice was made for two primary reasons. First, in our experience with integrating SIP and SOAP, arithmetic was never required; the conversion process typically extracts and copies message fragments, and PCCL can do that well. Second, this design conforms to the idea of a finite-state converter discussed in the Introduction, that is it is supported by theory. Furthermore, it is easy to generate fast code for such a constrained converter.

6. ARCHITECTURE OF THE SIP/WS GATEWAY

To support execution of PCCL specifications and following the requirements presented in Section 4.2, we designed a generic SIP/SOAP gateway architecture [Budiselic et al. 2010]. The generic gateway implements functionality common to all SIP/SOAP applications and uses conversion plugins for application-specific session management and message mapping. An overview of the architecture is shown in Figure 13. The gateway consists of five modules: the PCCL compiler, compiled plugin repository, active plugin repository, SIP/SOAP message handling module, and the arbiter.

The PCCL compiler translates PCCL application specifications into *compiled plugins*. Compiled plugins are executable versions of PCCL specifications. A PCCL specification is translated only once, when the application is first added to the gateway. The compiled plugins of all the applications supported by the gateway are stored in the compiled plugin repository. A compiled plugin is instantiated into an *active plugin* when the gateway receives the first message of an application. All active plugins are stored in the active plugin repository while the application is exchanging messages through the gateway. The relationship between compiled and active plugins is similar

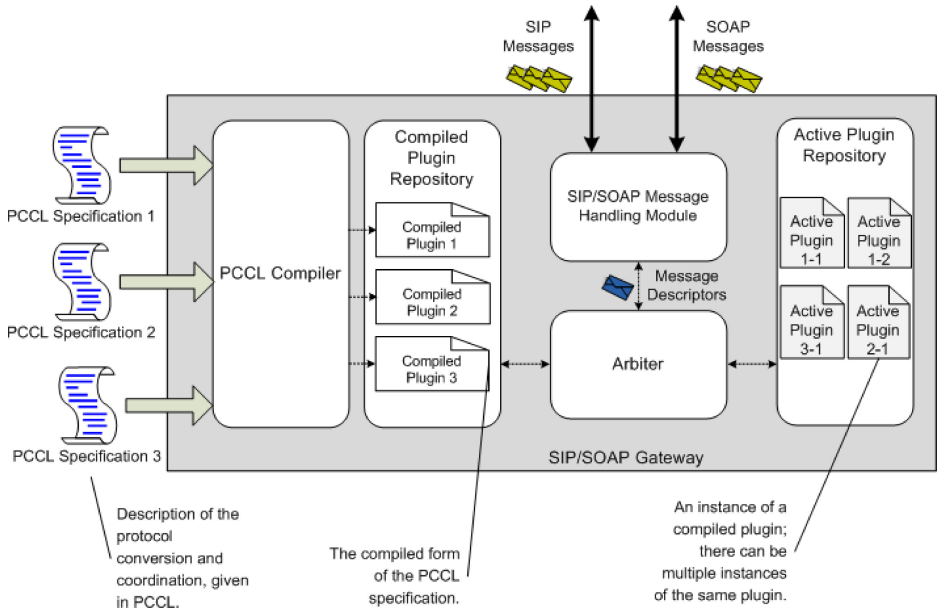


Fig. 13. Architecture of the SIP/SOAP gateway.

to the relationship between a class and an object of that class in object-oriented programming languages; the compiled plugin is a template for creating active plugins. Once created, the active plugin accumulates session data in an application-specific manner and determines the gateway's response to received messages, as defined by the original PCCL specification.

All message exchange with other communication nodes is done through the generic SIP/SOAP message handling module. This module parses incoming SIP and SOAP messages into internal message descriptors used by the gateway. Similarly, the message handling module composes SIP and SOAP messages from outgoing message descriptors generated in the gateway. Additionally, the module manages network connections and therefore serves as an abstraction layer for SIP and SOAP interaction with the outside world.

The arbiter is the control unit of the gateway. The main functions of the arbiter are interacting with the message handling module and active plugins to process incoming messages and management of active plugins as applications are started and finished. The interaction of the arbiter with the message handling module and an active plugin is shown in Figure 14. The arbiter exchanges message descriptors with the message handling module through queues. For every received message (1), the message handling module places one message descriptor into the queue. The arbiter removes the descriptor from the queue (2) and attempts to find an active plugin that can process the received message. For every active plugin from the active plugin repository, the arbiter first tries to correlate the message with the session that the plugin is handling (3). The session correlation process in the plugin is based on the *session* section of the PCCL specification. The arbiter passes the session identifier extracted from the message descriptor (4) along with the descriptor for condition matching in the plugin (5). The first step of condition matching is to compare the session identifier with session identifiers stored in the plugin if it is already defined. If these values do not match, the message is not part of the session executing through that active plugin. Otherwise, the

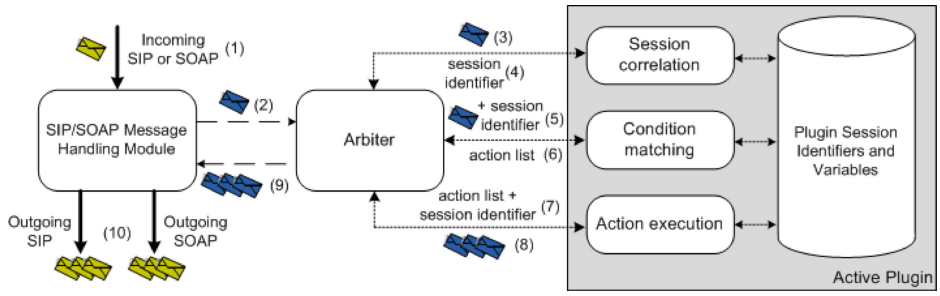


Fig. 14. Interaction between the arbiter, the SIP/SOAP message handling module, and an active plugin.

full matching process described by the PCCL *conditions* element is performed on the descriptor. Matching is successful if the session identifier matched the stored session identifier and the message satisfies at least one condition. If matching was successful, the list of actions generated by *exec* elements of the PCCL specification is returned to the arbiter (6). Otherwise, the arbiter repeats steps (3) through (6) for the next active plugin, until matching succeeds for one of them. If no active plugin matches the received message, the arbiter searches for a compiled plugin that could process the message as the first message of a new application. Compiled plugins are instantiated one by one until a matching plugin is found. If no matching compiled plugin is found, the message is discarded without effect.

For a successful match, the arbiter passes the action list and the session identifier to the plugin (7). The plugin then executes the actions generating outgoing message descriptors (8) and possibly changing its internal session identifiers and variables as specified in the *action* elements of the PCCL specification. If the received message ends the session and the application should be terminated, the plugin will include a special *DELETE* action into the action list. If the *DELETE* action is present in the action list, the arbiter removes the plugin from the active plugins repository after all the other actions are executed by the plugin. Finally, the arbiter pushes the outgoing messages to the outgoing message queue (9). Based on those descriptors, the message handling module composes and sends out outgoing SIP and SOAP messages (10).

7. CREATING A PCCL SPECIFICATION FOR A SIP/SOAP APPLICATION

Defining a complex interaction between SIP and SOAP communication nodes can be challenging even when using a DSL such as PCCL. A more methodical approach can help in overcoming these challenges. The PCCL language was defined in a way to support writing SIP/SOAP application specifications based on message sequence diagrams. In this section, we describe how to use message sequence diagrams to write a PCCL specification for the SIP/SOAP integration example from Section 2.

As a starting point in writing a PCCL specification, every use case of the application should be described by a message sequence diagram. The message sequence diagram should specify the order of messages exchanged between the communication nodes and the SIP/SOAP gateway and key parameters of all messages for that use case. The key parameters specific for a message are header values that define a certain use case or the expected contents of the message body. Additionally, for SIP, the key parameters include the SIP request method (e.g., NOTIFY) or response code (e.g., 200 OK). For SOAP messages, the SOAP method name, message type (request or response), and argument names and values should all be indicated. Since message sequence diagrams are just design aids in this approach, certain key parameters can be excluded if they can be easily deduced from the context.

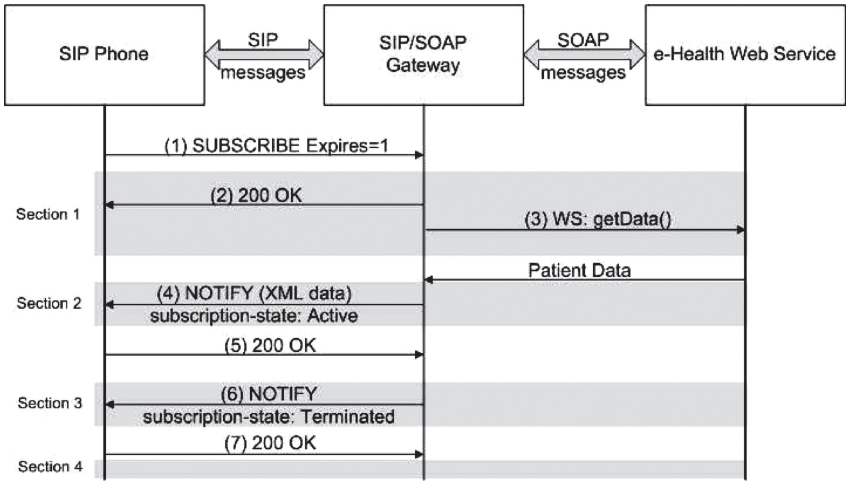


Fig. 15. One-time data pull message sequence diagram.

The application described in Section 2 has two use cases determined by the functionality of the Web service exposing the patient's vitals to doctors. The one-time data pull use case is shown in Figure 15. A SIP SUBSCRIBE message (1) starts the session. The *Expires* header set to one indicates to the gateway that the client is requesting a one-time data pull. The gateway should respond with a SIP OK message (2) and invoke the e-Health Web service's *getData* method (3). The Web service should respond with the requested data and the gateway should send this data to the SIP client in the body of a SIP NOTIFY message (4). The SIP client should indicate that the data was received via a SIP OK message (5) after which the gateway should initiate session termination with a SIP NOTIFY message (6). Finally, the client should reply with a SIP OK message (7) to terminate the session on the gateway.

The event notification use case is modeled with the message sequence diagram shown in Figure 16. The SIP phone specifies the subscription duration in the *Expires* header of the initial SUBSCRIBE message (1). The SUBSCRIBE message also contains desired threshold values for the patient's vitals. The gateway then acknowledges the request (2) and registers the subscription with the Web service (3). When invoking the *setSubscriptionStatus* method, the gateway specifies to the Web service that the callback method for event notification is called *sendData*. If subscription setup is successful, the gateway notifies the SIP client (4) and expects a SIP OK message in response (5). If any threshold value is reached for the duration of the subscription, the Web service will invoke the *sendData* callback method (6) returning the current vitals values in XML. The gateway then sends this XML description to the SIP client (7) after which the SIP session is terminated (8–10).

Based on message sequence diagrams for all the use cases of an application, it is straightforward to write *conditions* and *actions* elements of the PCCL specification for the application. Since the SIP/SOAP gateway is driven by incoming messages, that is, it performs actions exclusively as a response to a received message, the individual message sequence diagrams should be partitioned into sections between these incoming messages. All messages in each section will be outgoing messages from the gateway to one of the SIP or SOAP nodes. These outgoing messages should be constructed and sent out by the gateway in response to the received message that starts a section. There is a one-to-one correspondence between each section in a message sequence diagram and an *action* element in the application's PCCL specification. For each received

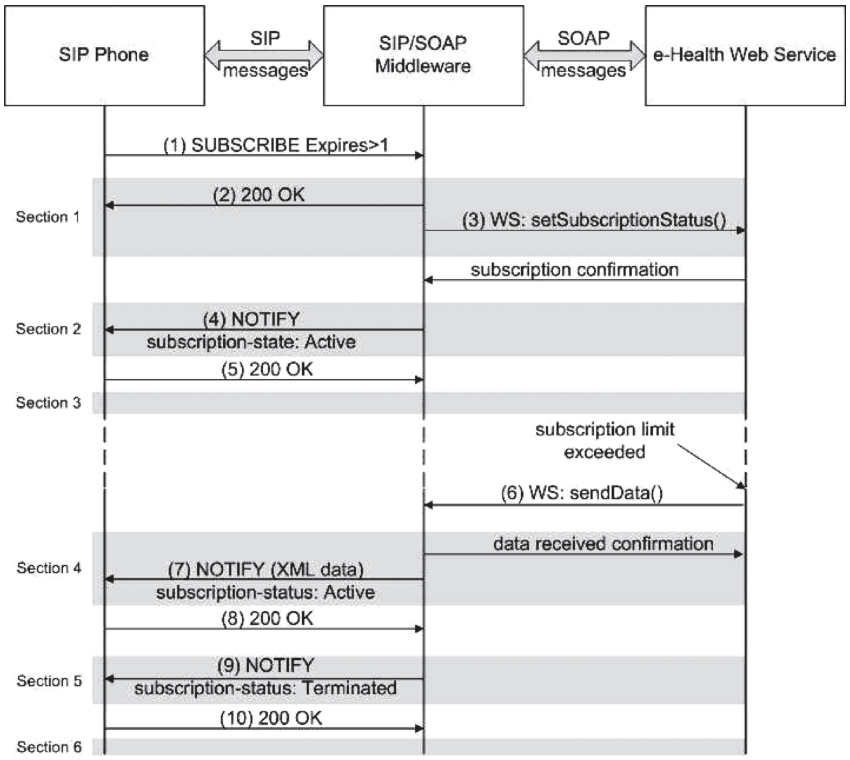


Fig. 16. Event notification message sequence diagram.

message, the middleware will determine which use case is currently active and which action should be activated. Actions should update the session state of the active plugin so that the next received message can be correctly processed.

Partitioning the message sequence diagram in Figure 15 yields four sections indicated with a gray background. Section 1 contains messages (2) and (3), Section 2 contains message (4), Section 3 contains message (6), and Section 4 is empty and starts after message (7). These four sections are described in PCCL with the four *action* elements shown in Figure 17.

Every action sends the messages in the corresponding sections and sets up the *session.sequenceSection* variable for the next section. This variable is used in *condition* elements to choose the appropriate action to activate as shown in Figure 18.

Every *condition* element tests if the application is in the state that expects this message to advance its state. If the state matches the expected state for the received message, the message is processed with the corresponding action element from Figure 17.

Segmenting the message sequence for the event notification use case yields six sections shown in grey in Figure 16. Figure 19 shows the entire *message* PCCL element for SIP SUBSCRIBE messages in the example application. The first *if* element handles the start of the one-time data pull use case as described earlier. The second *if* element handles the event notification use case. Any SUBSCRIBE message that doesn't satisfy either of these conditions will be reported as an error and ignored by the system. The remainder of the PCCL specification for the example application is omitted due to space constraints.

```
<action name="OneTimePullSection1" inType="SIP">
 <get type="string" location="body"> "?"$session.patientId${1} </get>
 <send protocol="SIP" peer="phone" type="OK" />
 <send protocol="SOAP" peer="ws" name="getData" type="Request">
  <arg name="patientId" type="string"> $session.patientId$ </arg>
 </send>
 <set name="session.sequenceSection" type="string" value="OneTimePullSection2" />
</action>

<action name="OneTimePullSection2" inType="SOAP">
 <get type="string" location="body:getDataResult"> $session.notify_body$ </get>
 <send protocol="SIP" peer="phone" type="NOTIFY">
  <header name="subscription-state" value="Active" />
  <header name="Content-Type" value="text/plain" />
  <body> $session.notify_body$ </body>
 </send>
 <set name="session.sequenceSection" type="string" value="OneTimePullSection3" />
</action>

<action name="OneTimePullSection3" inType="SIP">
 <send protocol="SIP" peer="phone" type="NOTIFY">
  <header name="subscription-state" value="Terminated" />
 </send>
 <set name="session.sequenceSection" type="string" value="OneTimePullSection4" />
</action>

<action name="OneTimePullSection4" inType="SIP">
 <set name="session.sequenceSection" type="string" value="OneTimePullTerminated" />
</action>
```

Fig. 17. *Action* PCCL elements for the four message sequence sections in Figure 15.

8. PERFORMANCE EVALUATION

The goal of this section is to asses the impact of the protocol conversion approach proposed in this article on the performance of the conversion gateway. In order to focus on the conversion process itself, we analyzed the performance of the *arbiter* module defined in Section 6. The other components of the gateway, like SIP and SOAP parsers, are inherent components to any system that converts between SIP and SOAP and were therefore not analyzed.

In our experimental setup we deployed a prototype implementation of the SIP/SOAP gateway on a dedicated workstation to minimize external effects on the measurements. Apart from the gateway, the experimental setup consisted of a locally deployed SIP client and a Web service deployed using the Apache CXF Framework [Apache 2012]. In all the experiments, we used the SIP client to send a SIP INVITE message to the SIP/SOAP gateway. To isolate the arbiter from the rest of the system, we made multiple copies of the message descriptor generated by the SIP parser and the arbiter processed them as if they were generated from separate messages.

For the test application, we used PCCL to define a simple plugin that extracts the *From* header value from the SIP INVITE message, changes the *state* session variable, and constructs a SOAP request. The plugin action definition also contained the DELETE action so that the arbiter removes the plugin from the active plugins repository after a message is processed. Therefore, the active plugin repository was empty before every message descriptor got processed and the state of the gateway was the same for all messages. In every experiment, we measured the time interval from the moment the arbiter takes the message descriptor from the incoming SIP queue to the moment it puts the SOAP message descriptor in the outgoing SOAP queue.

As explained in Section 6, to process a received message, the arbiter first searches for a plugin that can handle that message, that is, a matching plugin. In the first experiment we explored the cost of this search process. To that end, we defined an additional plugin that doesn't match INVITE messages and used a varying number

```
<conditions>
 <protocol name="SIP">
  <message peer="phone" type="SUBSCRIBE">
   <condition>
    <if>
     <EQ left="header:Expires" right="1" type="int" />
     <exec>OneTimePullSection1</exec>
    </if>
   </condition>
  </message>
  <message peer="phone" type="OK">
   <condition>
    <if>
     <EQ left="session.sequenceSection" right="OneTimePullSection3" type="string"/>
     <exec>OneTimePullSection3</exec>
    </if>
    <elif>
     <EQ left="session.sequenceSection" right="OneTimePullSection4" type="string"/>
     <exec>OneTimePullSection4</exec>
    </elif>
  </message>
 </protocol>
 <protocol name="SOAP">
  <message name="getData" peer="ws" type="Response">
   <condition>
    <if>
     <EQ left="session.sequenceSection"
       right="OneTimePullSection2" type="string" />
     <exec>OneTimePullSection2</exec>
    </if>
   </condition>
  </message>
 </protocol>
</conditions>
```

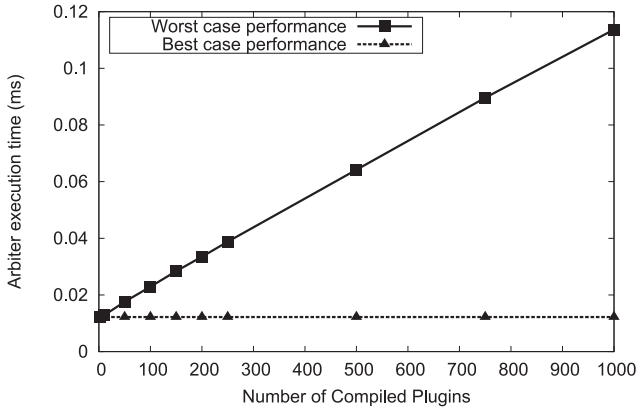
Fig. 18. The *conditions* PCCL element describing action activation logic for each message sequence section in Figure 15.

```
<message peer="phone" type="SUBSCRIBE">
 <condition>
  <if>
   <EQ left="header:Expires" right="1" type="int" />
   <exec>OneTimePullSection1</exec>
  </if>
  <elif>
   <GT left=:header:Expires" right="1" type="int" />
   <exec>EventNotificationSection1</exec>
  </elif>
 </condition>
</message>
```

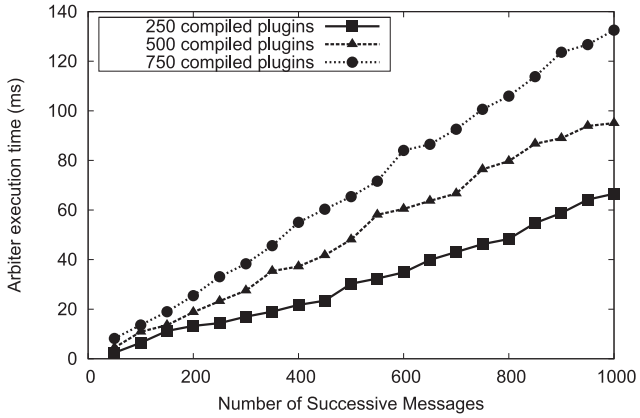
Fig. 19. The complete SIP SUBSCRIBE *message* PCCL element for the e-Health application.

of copies of that plugin alongside a single matching plugin. Since the setup of the experiment ensures that there are no active plugins when a message is received, the arbiter must sequentially search the compiled plugins repository to find the matching plugin. On one extreme, the matching plugin might be the first one instantiated. In that case, the message can be processed immediately and the search stops. This situation corresponds to the best-case search performance. On the other extreme, the matching plugin could be the last one instantiated which leads to worst-case search performance.

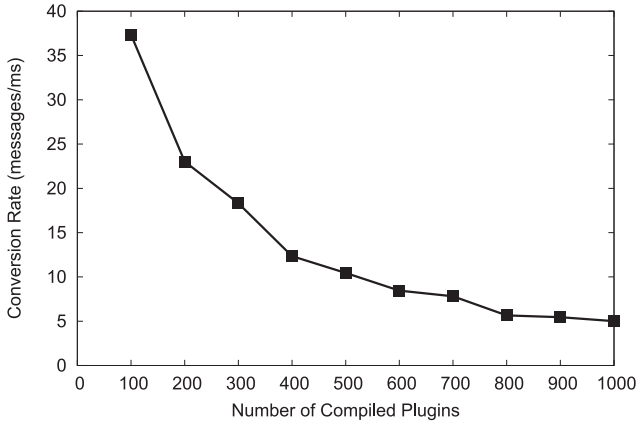
The experimental results in Figure 20(a) show execution times for best- and worst-case conversions performed by the arbiter for a single incoming message. The numbers on the graph are averages over 1000 messages. We achieved the best-case performance by placing the matching plugin for the incoming message at the front of the compiled plugin list. Similarly, the worst-case performance was achieved by placing the matching plugin at the end of the list. As could be expected, the results show a linear dependency of the worst-case execution time and the number of compiled plugins. The



(a) single message, varying active plugin count



(b) multiple messages



(c) message conversion rate

Fig. 20. Conversion process evaluation.

difference between the worst- and best-case performance is a result of the overhead inherent to the conversion methodology which is based on polling all available plugins until a matching plugin is found.

These results show that the order in which the plugins are instantiated and tested for matching is significant because the plugin search process introduces a potentially large delay in message processing. There are several approaches that could be applied to make sure that the average performance is closer to the best case. If the probability distribution for incoming messages is known ahead of time, the plugins could be statically arranged to achieve optimal average performance by placing plugins that process more frequent messages near the front. Alternatively, the plugins could be rearranged dynamically by the gateway so that most recently or most frequently used plugins get placed at the front. A similar issue arises in organizing the active plugin repository. Ultimately, the best strategy will depend on the characteristics of the use case.

In the second experiment, we measured how much time the arbiter takes to process a batch of messages arriving in close succession. The results in Figure 20(b) show a linear increase of total processing time with the increase of the number of messages. The three lines show the performance given 250, 500, and 750 compiled plugins in the gateway, and all measurements were done for the worst-case plugin search time, that is, when the single matching plugin was at the end of the plugin list. The results indicate that the processing time per message remains stable as the number of messages increases. The slope of the graph increases with the number of plugins due to plugin search overhead.

In the third experiment, we measured the message conversion rate of the arbiter in relation to the number of compiled plugins. The experiment was carried out using 1000 identical message descriptors for each plugin count and measuring the time required for their conversion. As in the previous experiment, measurements were performed for the worst-case plugin search performance. The results presented in Figure 20(c) show that the message conversion rate is inversely proportional to the number of compiled plugins.

9. RELATED WORK

Complementing features of SIP and SOAP are recognized by academia and the industry, resulting in many research efforts focused on integrating these protocols. The common goal for most research in this area is a superset of both protocols, a solution or a new standard that uses Web services for creating platform-independent services and semantically proper data exchange among them, and uses SIP for establishing and maintaining sessions among such services.

Most existing solutions propose modifications to communication nodes' protocol stacks by adding a SIP or SOAP protocol layer. In Liu et al. [2004] the authors consider the problem of session state management in integrating SIP and SOAP. They propose to insert session data into SOAP message envelopes. However, the proposed intervention requires modification of existing Web services and thus differs from our approach.

Research presented in Dong and Newmarch [2010] also relies on using both a SIP stack and a Web services stack in all communication nodes. The presented approach is to first use SIP for establishing a session, and afterwards utilize the obtained session identifier in SOAP messages. Such an approach can provide guidance for future services that utilize both SIP and SOAP protocol stacks in all communication nodes. However, it does not propose a solution for general integration with existing SOAP-based services and requires changes to nodes' protocol stacks.

A similar approach is used in the Akogrimo project [Jähnert et al. 2007, 2010] where SIP and the existing IMS architecture of telecommunication networks is used for establishing and maintaining sessions among Web services that communicate with SOAP messages. Additionally, authors describe a prototype that utilizes Web service to mimic formalisms of IMS nodes in order to interact with modern telecommunication networks. While this work clearly shows a possible evolution of both Web services and the IMS architecture, it doesn't explain how to enable direct communication and interaction of SOAP and SIP nodes in general.

Another similar approach is described in Lakas et al. [2007]. The goal of this research is bridging the gap between the Web services on the Internet and the Public Switched Telephone Network (PSTN) for typical telecommunications networks' services like voice, voicemail, and location services. SOAP is used to carry SIP messages in its headers and a specially designed SIP/PSTN gateway is used for transmitting SIP messages to SIP User Agents that don't support Web services in their protocol stacks. Like the previously mentioned solutions, this approach also doesn't propose means for interaction among SIP and SOAP nodes in general.

A somewhat different approach is used in the IMS Enterprise Suite SOAP Gateway [IBM 2010; Wicks et al. 2009] that allows reuse of IMS applications as Web services. The gateway translates incoming SOAP messages and relays them to IMS SIP applications. The mapping between clients external to the IMS network, that communicate with the IMS network through Web services, and servers, which are IMS applications, is done through automatically generated mapping files. Mapping files are generated for each Web service interface that exposes some IMS application functionality to external networks. Compared to the solution presented in this article, this gateway doesn't provide bidirectional conversion functionality, as it is not possible to define mapping between SIP clients and Web-service-based servers. Additionally, mapping files contain only simple, stateless session information. Thus, more complex interaction between SOAP clients and SIP servers isn't feasible.

Similarly, Avaya products [Avaya 2010] (Avaya SOA, Avaya SIP Application Server, Avaya Communication Process Manager) provide means to access telecommunication infrastructure through Web services. The proposed solution implements a set of services that can access SIP-based nodes. The Avaya service set is also exposed as Web services, therefore effectively allowing SOAP messages to reach a SIP network. Although allowing the communication nodes to remain unchanged, this approach does not provide a general solution for SIP/SOAP conversion due to predefined and limited conversion functionality.

Another similar commercial solution is presented in Maes [2009]. The solution is based on a gateway comprised of several Oracle products that provide seamless instant messaging between various kinds of end-user clients, including SOAP-based and SIP-based clients. However, it must be noted that instant messaging itself has simple, unidirectional transactions. Thus, the presented solution doesn't provide means for complex bidirectional transactions that are commonly required. Also, since instant messaging is not stateful, the solution doesn't address the issue of WS statelessness.

The problem of stateful, complex bidirectional transactions among SIP-based and SOAP-based nodes is identified in Levenshteyn and Fikouras [2006]. The authors clearly identify the issues of WS statelessness and utilize additionally proposed Web services standards, like the already mentioned WS-CAF [OASIS 2011], for adding state to Web services. Additionally, the authors acknowledge the issue of session mapping between SIP and SOAP. However, no concrete solution is presented for this issue, only a guideline for utilizing the Session Description Protocol (SDP) as a carrier of Web services session information towards SIP nodes, and WS-CAF as a carrier of SIP session information towards Web services.

Another approach to solving the problem of stateful bidirectional transactions among SIP-based and SOAP-based nodes uses Business Process Execution Language (BPEL) [OASIS 2007] or Enterprise Service Bus (ESB) products (e.g., OpenESB [Xerox 2009], Oracle Service Bus [Oracle 2008], MicrosoftBizTalk [Microsoft 2010]). Both BPEL and ESB are examples of the general-purpose programming-in-the-large concept [DeRemer and Kron 1976] for conducting Enterprise Application Integration (EAI). By using the mediation EAI pattern, brokering between different heterogeneous applications and protocol scan can be established. For example, mediation between SIP-based and SOAP-based nodes can be achieved by creating BPEL or ESB workflows that utilize advanced language mechanisms (like BPEL correlation sets) or inline code snippets (e.g., Java for OpenESB, Oracle Service Bus, and Oracle Mediator; C# for Microsoft BizTalk) for managing state between SIP and SOAP. However, such an approach is heavyweight in comparison to the one presented in this article. First, it requires design and implementation of protocol-level statefulness that isn't inherently supported by BPEL or ESB. For instance, an example of such statefulness issue is the necessity to handle SIP session ID. On the other hand, PCCL solves this issue automatically, with minimal setup. Second, business- or application-level statefulness must be implemented in BPEL correlation sets or ESB inline code snippets. For general message coordination scenarios, both of these require general-purpose programming in languages like Java or C#. PCCL, on the other hand, has explicit language constructs for coordinating and converting SIP and SOAP semantics. Thus, it can be concluded that BPEL and ESB approaches require more knowledge on the behalf of implementers. While basic knowledge of SIP and SOAP is certainly required to create a coordinated application with SIP and SOAP nodes, PCCL eliminates much of the complexity inherent to this process. On the other hand, BPEL and ESB approaches are designed for the broader EAI scope and therefore don't focus on the specifics of SIP/SOAP integration. Using PCCL, implementers can focus on application-specific semantics of SIP and SOAP messages that will be exchanged through the gateway and express the conversion process in PCCL itself, without relying on general-purpose programming.

In Kongdenfha et al. [2009] the authors present a generic framework based on protocol mismatch patterns that can be used to construct an adapter (converter) for heterogeneous service-oriented architectures. The authors present a taxonomy of common protocol mismatch patterns. The patterns need to be instantiated by the converter developers in order to resolve mismatches for a given pair of services. More specifically, a combination of BPEL [OASIS 2007] and a domain-specific query language is used to define the conversion rules. Although the presented approach has similarities with the conversion method proposed in our article, some significant differences exist. In our article we clearly state and address the mismatches specific to SIP/SOAP conversion. We acknowledge that the presented requirements could be used to define mismatch patterns as suggested by the authors. However, the authors limit their discussion on service integration to the business process level, that is, they observe a protocol as a sequence of message exchanges between clients and services. Thus, proposed mismatch patterns are complemented with business logic to facilitate a specific conversion scenario. We believe that protocol mismatches should be analysed for a variety of application-layer protocols that are not necessarily compliant with service composition engines. Following this approach, more comprehensive mismatch patterns could be developed. Furthermore, the authors propose an implementation based on extending a BPEL execution engine. Thus, the conversion rules can be linked to specific events in the BPEL workflow which define service interactions. Although this approach is valid, we suggest that protocol conversion should be more clearly separated from a specific workflow execution engine. Thus, the converter events should be limited to message

reception/sending. In such a way it is possible to construct a conversion gateway that can be utilized by various business process execution engines.

10. CONCLUSION AND FUTURE WORK

In this article we explore the possibilities of designing a general-purpose converter to integrate SIP and SOAP services. As most other application-layer protocols, SIP and SOAP have pronounced semantic differences that lead to hard protocol mismatches. For example, SIP is oriented around messages exchanged in the context of a session while SOAP is a request-response protocol with no support for sessions. Additionally, it is, in general, not possible to define semantically equivalent SIP and SOAP messages since message semantics on the application layer depend on the application. For example, the same responses from a weather forecast Web service might need to be mapped to completely different SIP messages for SIP nodes expecting the data to be formatted in a specific way.

Through an analysis of SIP and SOAP and predefined business goals, we defined a methodology for integrating SIP and SOAP protocols. The methodology is based on three key ideas. First, SIP and SOAP integration should be done through a generic SIP/SOAP gateway. This gateway can implement SIP and SOAP message handling and facilities for storing session data, while using application-specific conversion plugins for message mapping and session management rules. We describe the architecture of a generic SIP/SOAP gateway that has been implemented and used in practice.

Second, message mapping and session management rules should be specified in a domain-specific language since defining complex interactions between SIP and SOAP nodes in a general-purpose language quickly becomes impractical. We describe the Protocol Conversion and Coordination Language that is an XML-based language that supports bidirectional conversion of SIP and SOAP messages and coordination of an arbitrary number of SIP and SOAP communication nodes in a SIP/SOAP application. PCCL isolates users of the SIP/SOAP gateway from the many intricacies of SIP and SOAP protocols and allows them to focus on the interaction logic required for their application. Conversely, PCCL is also isolated from the architecture of the gateway. This fact makes it possible to change the architecture of the gateway, for example, from a centralized to a distributed system, without changing PCCL specifications of previously defined applications.

Third, the domain-specific language should provide constructs for modeling message sequence diagrams of an application because its central purpose is to define message mapping which is exactly specified in message sequence diagrams. We show how to specify an e-Health SIP/SOAP application in PCCL based on message sequence diagrams and provide a nearly complete PCCL specification.

The presented research offers several paths for further research. As noted in Section 3, a dedicated intermediary node such as the SIP/SOAP gateway can become a bottleneck and a single point of failure in a system. The presented gateway was implemented as a centralized system running on a single-server machine. However, the presented architecture does not limit the implementation to such a configuration. Since the modules in the gateway are loosely coupled and communicate through well-defined interfaces, it should be possible to distribute the gateway to several machines with small modifications like replacing in-memory message queues with mailboxes that can be accessed through the network.

Through performance analysis of the arbiter module, we've shown the significance of the order of conversion plugins in the plugin repository. Some approaches for addressing this issue are suggested in Section 8, and could be explored further.

Using a more formal model of message sequence diagrams, it should be possible to create an even simpler domain-specific language for defining message mapping

between SIP and SOAP nodes. This new language could be translated to PCCL or even directly to executable conversion plugins.

Finally, the methodology presented in this article for integrating SIP and SOAP can be applied to other protocols that have hard mismatches and do not present a meaningful general mapping between semantically equivalent messages.

ACKNOWLEDGMENTS

The authors thank Ivan Benc from Ericsson Nikola Tesla d.d., Sinisa Srbljic, Dejan Skvorc, Miroslav Popovic, Klemo Vladimir, Marin Silic and Zvonimir Pavlic from the University of Zagreb, Faculty of Electrical Engineering and Computing, Jakov Krolo from Corvus Info d.o.o., Croatia and Daniel Skrobo from Asseco SEE, Croatia.

REFERENCES

APACHE. 2012. Apache cxf: An open-source services framework. http://cxf.apache.org/.

AVAYA. 2010. Avaya products. http://www.avaya.com.

BUDISELIC, I., DELAC, G., SEGO, D., AND STEFANEC, T. 2007. SIP/WS interworking triggering gateway. In *Proceedings of the Ericsson Nikola Tesla Summer Camp: New Generation Network Applications and Protocols*. 270–311.

BUDISELIC, I., ZUZAK, I., AND BENC, I. 2010. Application middleware for convergence of ip multimedia system and web services. In *Proceedings of the 33rd International MIPRO Convention*. 507–512.

CALVERT, K. AND LAM, S. 1990. Formal methods for protocol conversion. *IEEE J. Select. Areas Comm. 8*, 1, 127–142.

CALVERT, K. L. AND LAM, S. S. 1989. Deriving a protocol converter: A top-down method. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM'89)*. ACM Press, New York, 247–258.

CUBIC, I., MARKOTA, I., AND BENC, I. 2010. Application of session initiation protocol in mobile health systems. In *Proceedings of the 33rd International MIPRO Convention*. 367–371.

CURBERA, F., DUFTLER, M., KHALAF, R., NAGY, W., MUKHI, N., AND WEERAWARANA, S. 2002. Unraveling the web services web: An introduction to soap, wsdl, and uddi. *IEEE Internet Comput. 6*, 86–93.

DAY, M., AGGARWAL, S., MOHR, G., AND VINCENT, J. 2000. Instant messaging/presence protocol requirements. RFC 2779, Network Working Group, Internet Engineering Task Force.

DEREMER, F. AND KRON, H. 1976. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Softw. Engin. SE-2*, 2, 80–86.

DONG, W. AND NEWMARCH, J. 2010. *Adding Session and Transaction Management to Web Services by Using SIP*. Lap Lambert Academic Publishing.

DOUBANGO. 2012. Simpl5: The world's first html5 sip client. http://code.google.com/p/sipml5/.

IBM. 2010. IBM ims soap gateway. http://www-01.ibm.com/software/data/ims/soa-enterprise-suite/soap/.

HO, J.-M., HU, J.-C., AND STEENKISTE, P. 2001. A conference gateway supporting interoperability between sip and h.323. In *Proceedings of the 9th ACM International Conference on Multimedia (Multimedia'01)*. ACM Press, New York, 421–430.

HUHNS, M. N. AND SINGH, M. P. 2005. Service-Oriented computing: Key concepts and principles. *IEEE Internet Comput. 9*, 75–81.

JÄHNERT, J., CUEVAS, A., MORENO, J. I., VILLAGRA, V. A., WESNER, S., OLMEDO, V., AND EINSIEDLER, H. 2007. The "akogrimo" way towards an extended ims architecture. In *Proceedings of the 11th International Conference on Intelligence in Networks (ICIN'07)*.

JÄHNERT, J., MANDIC, P., CUEVAS, A., WESNER, S., MORENO, J. I., VILLAGRA, V., OLMEDO, V., AND STILLER, B. 2010. A prototype and demonstrator of Akogrimo's architecture: An approach of merging grids, soa, and the mobile Internet. *Comput. Comm. 33*, 1304–1317.

KAVANTZAS, N., BURDETT, D., RITZINGER, G., FLETCHER, T., LAFON, Y., AND BARRETO, C. 2005. Web services choreography description language, version 1.0. W3C Candidate Recommendation. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109.

KONGDENFHA, W., MOTAHARI-NEZHAD, H. R., BENATALLAH, B., CASATI, F., AND SAINT-PAUL, R. 2009. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Trans. Services Comput. 2*, 94–107.

LAKAS, A., SERHANI, M., BOULMALF, M., AND BADIDI, E. 2007. A framework for integrating sip-based communication services and web services. In *Proceedings of the IADIS International Conference on Telecommunications, Networks and Systems*.

LAM, S. S. 1988. Protocol conversion. *IEEE Trans. Softw. Engin. 14*, 353–362.

LEVENSHTEYN, R. AND FIKOURAS, I. 2006. Mobile services interworking for ims and xml web services. *IEEE Comm. Mag. 44*, 9, 80–87.

LIU, F., CHOU, W., LI, L., AND LI, J. 2004. WSIP: Web service sip endpoint for converged multimedia/multimodal communication over ip. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*. IEEE Computer Society, 690–697.

MAES, S. 2009. Intelligent message processing: Patent. U.S. Patent Application Publication, pub no. US2009/0125595 A1 (pub date 5/14/09).

MERNIK, M., HEERING, J., AND SLOANE, A. M. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv. 37*, 316–344.

MICROSOFT. 2010. Microsoft biztalk server. `http://www.microsoft.com/biztalk/`.

OASIS. 2007. Web services business process execution language (wsbpel). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

OASIS. 2011. Web services composite application framework (ws-caf). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf`.

OKUMURA, K. 1986. A formal protocol conversion method. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures and Protocols (SIGCOMM'86)*. ACM Press, New York, 30–37.

ORACLE. 2008. Oracle service bus. `http://www.oracle.com/technetwork/middleware/service-bus/overview/index.html/`.

ROACH, A. 2002. SIP-Specific event notification. RFC 3265, Network Working Group, Internet Engineering Task Force.

ROSENBERG, J., SCHULZRINNE, H., ET AL. 2002. SIP: Session initiation protocol. RFC 3261, Network Working Group, Internet Engineering Task Force.

ROSENBERG, J. D. AND SHOCKEY, R. 2000. SIP: A key component for internet telephony. *Comput. Teleph. 8*, 124–139.

SAINT-ANDRE, P., HOURI, A., AND HILDEBRAND, J. 2007. Interoperability between the extensible messaging and presence protocol (xmpp) and sip for instant messaging and presence leveraging extensions (simple). Internet-Draft draft-ietf-autoconf-manetarch-07, Internet Engineering Task Force.

TAO, Z. P., BOCHMANN, G., AND DSSOULI, R. 1995. An efficient method for protocol conversion. In *Proceedings of the 4th International Conference on Computer Communications and Networks (ICCCN'95)*. IEEE Computer Society, 40–47.

W3C. 2007a. SOAP version 1.2 part 0: Primer (second edition). `http://www.w3.org/TR/soap12-part0/`.

W3C. 2007b. Web services description language (wsdl) version 2.0 part 1: Core language. W3C Recommendation. `http://www.w3.org/TR/2007/REC-wsdl20-20070626`.

WICKS, G., AERSCHOT, E. V., BADREDDIN, O., ET AL. 2009. *Powering SOA Solutions With IMS*. Vervante.

XEROX. 2009. The open enterprise service bus. `http://openesb-dev.org/`.

Received August 2011; revised June 2012; accepted August 2012