Design and development of Middleware Gateway IP Multimedia System and Web Services

P. SASHI REKHA, C. SREEDHAR
G.PULLA REDDY ENGINEERING COLLEGE, KURNOOL, A.P.
ASSOCIATE PROFESSOR, CSE DEPT., GPREC

Abstract— Rapid growth in Internet (Web) applications and usage of mobile in telecommunication is present trends in communication. In Web 2.0 web services architecture is become versatile tool for enterprise solutions in distributed environment systems. To provide an enterprise solution it is not possible to get professional solution at one place. Hence we need to integrate different web services to provide an enterprise solution to meet the requirements. To integrate different web services Service Oriented Architecture is developed which uses SOAP Protocol. Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. In mobile multimedia applications are used to transfer audio and video information using internet. To access internet on mobile signaling protocols are developed. Among those protocol stack session initiation protocol (SIP) is commonly used.

To combine modern web features (Web Services) on to mobile phone interoperability problem is occurred, because mobile uses SIP messages and web services uses SOAP (Simple Object Access Protocol) messages to perform communication. It means a web service does not understand SIP messages and vice-versa. In this regard a necessity is identified to develop application level gateway to exchange the SIP/SOAP message interface.

I. INTRODUCTION

Today’s telecom users are increasingly demanding. They are more individualistic, independent, informed and involved than ever before, and they welcome services that appeal to their emotions as well as their practical needs. New, exciting services and enhancements to existing services have a key role to play in making the communications experience much more like interacting face-to-face. New advanced terminals and communication mechanisms adapted for user needs will enable this and hide technical complexity. Users are now used to accessing information, entertainment and other content-rich services through a variety of channels. Telecom operators have a great opportunity to integrate and extend the multimedia experience through new highly personalized person-to-person, person-to-content and group services. In recent years, mobile devices have gained Internet access, bringing new challenges in protocol interoperability. While implementing non-native protocol stacks overcomes protocol boundaries for new services, integrating numerous existing services requires other approaches.

This paper presents an approach for integrating Session Initiation Protocol (SIP) services and Web Services that use the SOAP messaging protocol. In the context of this paper, a SIP service is a computer process that uses the SIP protocol to expose some functionality. SIP is a text-based application layer protocol used mainly in VOIP and video session management. Extensions of SIP are used in other areas, such as instant messaging and event notification. SIP is also used as the main signaling protocol in the Internet Multimedia Subsystem (IMS). IMS is a functional architecture designed for creation and deployment of multimedia telecommunication services over IP. The Web Services standards family defines a set of protocols for implementing systems based on the principles of Service Oriented Architecture (SOA). The basis of the Web Services stack are SOAP, WSDL and UDDI, a set of XML based protocols and standards for service invocation, description, and discovery. SOAP uses HTTP or SMTP as transport protocols, which coupled with being XML based, makes Web Services platform independent. The Web Services standards family is widely used in enterprise environments, primarily for intra organization service interoperability.

II. IP MULTIMEDIA SUBSYSTEM VS WEB SERVICES

The IP Multimedia Subsystem (IMS) is an architectural framework for delivering Internet Protocol (IP) multimedia services. It was originally designed by the wireless standards body 3rd Generation Partnership Project (3GPP), as a part of the vision for evolving mobile networks beyond GSM. Its original formulation (3GPP R5) represented an approach to delivering “Internet services” over GPRS. To ease the integration with the Internet, IMS uses IETF protocols wherever possible, e.g. Session Initiation Protocol (SIP). According to the 3GPP, IMS is not intended to standardize applications but rather to aid the access of multimedia and voice applications from wireless and wire line terminals, i.e. create a form of fixed mobile-convergence (FMC). This is done by having a horizontal control layer that isolates the access network from the
service layer. From a logical architecture perspective, services need not have their own control functions, as the control layer is a common horizontal layer. However in implementation this does not necessarily map into greater reduced cost and complexity.

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine – processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services provide a standard means of inter-operating between different software applications, running on a variety of platforms and/or frameworks. This document (WSA) is intended to provide a common definition of a web service, and define its place within a larger Web services framework to guide the community.

III. INTEGRATION OF SIP AND SOAP PROTOCOLS

Integrating SIP and SOAP presents several challenges arising from the differences in the intended use of the protocols. Since SIP is a stateful and SOAP primarily a stateless protocol, the primary challenge lies in the differences in state management. SIP is designed to be used mainly for VoIP signaling and thus SIP nodes locally store session state. This state is associated with what is called a transaction and usually includes authentication tokens and Quality of Services (QoS) settings.

Different to the statefulness of SIP, one of the main principles in SOA is keeping services stateless. Since storing state in local buffers couples the service to the user it reduces the number of requests that can be served at the same time. With this in mind, the SOAP protocol was designed to be a stateless request-response protocol.

A possible way for a stateful service to communicate over a stateless protocol is by sending application specific state information in message headers, which is also the approach used in our project. Furthermore, in order to support multiple applications running concurrently through the same application middleware, a common SIP/SOAP session must be maintained so that incoming messages can be associated with the appropriate application.

In recent paper, the same group of authors describes a new protocol called Web Services Initiation Protocol (WSIP) that is also a dual-stack solution. Support sessions, WSIP relies on WS session, a standardized extension to the basis WS stack which specifies how a Web Service based session establishment will result in a unique session ID for the client, which is subsequently included in a SOAP header of all the messages in that session. To support full duplex communication, every node hosts two services with separate WSDL descriptions, one for the client and one for the server role.

In summary, most research is focused on upgrading devices in the SIP domain with the Web Services protocol stack, and vice versa. While this can be a good solution when creating new services with interoperability in mind, it does not ease integration of existing services. We take a different approach in which devices need not be upgraded to dual stacks. Rather, the proposed application middleware acts as a gateway between systems and makes all the message exchange conversions and coordination required. This allows the usage of the middleware in situations where there is an unchangeable separation between protocol domains, e.g. SIP nodes cannot communicate using SOAP. Additionally, existing services that use either SIP or SOAP can communicate with each other through the middleware without change.

IV. USE CASE OF SIP AND SOAP INTEGRATION

We illustrate the use of the proposed middleware system with a use case of SIP-SOAP integration. An application level gateway was developed for this use case. Through this gateway, we were able to formulate a set of requirements for the proposed general application middleware architecture described in this paper. These requirements are discussed at the end of this section.

![Fig. 1. SIP/WS integration use-case](image)

Fig.1 presents the integration use case.

Information gathered from a sensor network is exposed through a Web Service. The sensor network monitors temperature in several rooms in a building. A mobile user with a SIP phone wants to access the current sensor information, but is unable to do so directly because the SIP phone doesn’t implement the Web Services protocol stack, and therefore cannot compose or parse SOAP messages. Similarly, the Web Services provide does not implement the SIP protocol, and thus cannot deal with SIP messages. As a solution, we introduced an application level SIP/WS gateway that presents a SIP interface to the SIP client for accessing the sensor network. The SIP client initiates communication by sending a SIP message to the gateway. Since the Web Service supports both synchronous and asynchronous data retrieval, the gateway mirrors this functionality in its SIP interface. To provide synchronous data retrieval, the gateway sends a SOAP message to the Web Service and extracts the data from the SOAP response. For asynchronous retrieval, the gateway exposes...
a single-method callback Web Service to the sensor network service.

The message sequence for synchronous communication with the sensor network service is depicted in Fig. 2. The SIP phone sends a SIP SUBSCRIBE message with the Expires header set to 1 indicating a one-time pull operation. The gateway parses the received SIP message, sends a SIP OK response message back to the SIP phone (2), constructs a SOAP message according to predefined application specific rules and sends it to the Web Service in the SOAP domain (3). To allow multiple users to access the sensor network Web Service at the same time, the gateway internally stores a link between the SIP transaction identifier (Call-ID header of the SIP SUBSCRIBE message) and the TCP connection it opens to the Web Service. After the Web Services responds with the requested sensor data in a SOAP response message, the gateway extracts the required result, looks up the related SIP transaction identifier and replies to the SIP client with a SIP NOTIFY message (4) containing the requested data. Finally, the session is terminated with an OK-NOTIFY-OK sequence (5-7).

Fig. 2 synchronous Data Retrieval

Fig. 2 synchronous Data Retrieval

Fig. 3 presents the asynchronous mode of operation in which the SIP user sets limits for the sensor readings and gets notified when a limit is exceeded. First, the SIP client specifies the subscription duration in seconds in the Expires header of the SUBSCRIBE message (1) additionally, the client specifies which sensor parameters should be monitored and what their limit values are. The gateway then registers the subscription (3) and notifies success to the client (4). To allow asynchronous notification by the sensor Web Service, the ad-hoc gateway exposes a callback SOAP Web Service interface. If a sensor value reaches the set subscription limit, the sensor network Web Service invokes the gateway service’s send Sensor Data method with the event XML message (6). Finally, the gateway relays the event description to the client (7) and the session is terminated (8-9).

By creating the described application level gateway, we were able to allow SIP clients to use the sensor network Web Service by using SIP exclusively, unaware of Web Service SOAP interactions. From analyzing the requirements for the application level gateway, some requirements for the general SIP/WS application middleware become apparent. First, the middleware must maintain a unified session between its SIP and SOAP clients. In the presented use case, all the messages shown in Fig. 2 and 3 are part of the same unified session. The purpose of the unified session is to associate related messages to each other, providing them with the necessary from previous messages. Second, the middleware must store this context so it can persist through the duration of a session. For example, in the sensor network use case, the gateway had to store the value of the Call-ID SIP header to define the unified session, as well as the value of the Expires header in the asynchronous mode of operation. Additionally, some applications require the ability to persist data across sessions. For example, in order to reduce message size, the SIP client application in the use case only sends request parameters on the first request and in case the parameters change. Since every request is processed in a separate session, the request parameters need to be stored across multiple sessions.

Third, to allow asynchronous communication with SOAP clients and applications initiated by a SOAP message, the middleware must be able to create and expose Web Services interfaces to external systems. A further set of requirements originates from the goal of general interoperability of SIP and SOAP. To make the application middleware architecture capable of handling diverse SIP – SOAP interactions, the design must enable user defined plug-ins that define the particulars of an application. Specifically, users must be able to define specific message sequences for their use case. To simplify this process, users should be provided with a simple language for defining these message sequences. Additionally, the architecture should support multiple SIP and SOAP clients communicating in the same application.
V. SIP/WS APPLICATION MIDDLEWARE

Fig. 4 shows the architecture of the developed SIP/SOAP message handling module contains the logic for receiving, parsing, composing and sending SIP and SOAP messages. The data storage module contains data structures for maintaining unified SIP-SOAP sessions and application specific data. Application specific data can persist either for the duration of a session or through several sessions in a single application. Application specific logic that defines an application’s message flow and data persistence is abstracted away behind a trigger interface. All available triggers are stored in the trigger repository module.

Finally, the arbiter module acts as the control unit of the system. The arbiter controls the message flow through the system, manages the loading of triggers into memory and updates the data storage.

There, the middleware is used in two ways- application developers create and store trigger repository, while clients use the middleware for consuming cross-protocol services modeled by triggers. To initiate communication, a SIP or SOAP client sends a request message to the application middleware. The request is processed by the message handling module and the arbiter is notified that a new request was received. The arbiter then searches the trigger repository for a trigger that defines the application specific logic for the particular message exchange and loads that trigger into memory. The arbiter passes the received message to the loaded trigger which replies with a list of messages that need to be sent out and updates the data storage. Finally, the arbiter passes the outing messages to the message handling module.

A. Generic SIP/SOAP Message Handling Module

The message handling module communicates with the arbiter through four message queues. These queues relay data structures representing incoming or outgoing SIP and SOAP messages. The internal structure of the module is presented in Fig.5.

The module consists of a SIP receiver (1) which listens for incoming SIP messages on the network interface and inserts them into the receive SIP queue (2). The SIP parser (3) picks up incoming messages from the queue, parses them into memory data structures, and inserts the structures into the in SIP queue (4) from where they are eventually picked up by the arbiter. The SIP parser used in the SIP/WS application middleware was generated by an ABNF parser generator [7, 9].

Similarly, when the system needs to sends a SIP message, the arbiter constructs a data structure describing the outgoing SIP queue (5). The SIP generator (6) takes the data structure from the out SIP queue, constructs the corresponding SIP message and pushes it into the send SIP queue (7). The SIP sender (8) module then takes the message from the queue, and sends it to the destination on the network. The module structure for SOAP message handling is analogous to the structure for SIP message handling (9-16).

B. Data storage module

The data storage consists of two data structures: the session info and the application info. The purpose of the session info data structure is to provide context to the application trigger when determining if a received message belongs to the active unified session of the application, or it is the first message of a new session. Application specific data required to persist only for the duration of a single session can also be stored in the session info data structure. The session data depends on the particular services being interconnected and is defined in the application specific triggers. For example, for the use case described in section 3, the SIP participant is identified using the Call-ID header value and From and To header tags which are known after the initial SUBSCRIBE-OK message sequence. On the SOAP side, the Call-ID is used as a session identifier and is
inserted into all SOAP messages. The value of the Expires header which indicates a one-time data pull operation or the subscription duration is also stored in the session info data structure.

The application info data structure is used to store information that persists across multiple sessions. To use this functionality, the application trigger defines a global application identifier that must be present in messages in all related sessions. For example, in the example discussed in section 3, every one-time pull operation is done in a separate session. However, the SIP client application only sends the request body on the first request or when the request changes. Subsequent request only contain headers in order to reduce message size. To support this kind of operation, the request body can be stored in the application info data structure.

C. Triggers and the Trigger repository

In order to create an application by connecting services through the SIP/WS application middleware, an application specific plug-in called a trigger must be created. The middleware uses the trigger to determine the appropriate reaction to received messages. For each received message, the trigger defines how the session and application info data structures need to be updated, and which outgoing messages need to be sent out. SIP/WS middleware triggers are stored in the trigger repository and fetched during processing of incoming messages. While an application is active, an instance of the trigger governing that application is called an active trigger.

To achieve an open pluggable architecture, the SIP/WS application middleware defines a uniform interface of three methods that each trigger must implement: getApplicationId, match and activate. All methods operate on the received message and the match and active methods are given access to session and application data.

The getApplicationId method must return the identifier of the application that a given SIP or SOAP message belongs to. The semantics of the application identifier are application specific and are defined in the getApplicationId method of every trigger using data unique to that application. For example, in the sensor network use case, the application identifier can be created based on the location of the sensor network Web Service and the SIP client URI when the initial SIP message is received. The match method defines the sequence of steps that the application middleware will execute in response to a received message. This sequence is returned as a list of action names that is then passed to the activate method to execute. Purge and delete actions are predefined in the middleware while other actions must be defined within the trigger. Purge resets the application by clearing all session and application data associated with the received message, while delete terminates the application by clearing all session and application data and uploading the trigger from the middleware. The result of each action is a set of outgoing messages and changes in the data storage. Fig. 6 illustrates how triggers are used when a message arrives to the middleware. Upon receiving a message, the arbiter must first find the trigger which can handle the received message. For each candidate trigger, the arbiter passes the message descriptor to the getApplicationId method (1). The arbiter then passes the message descriptor and read-only access to the data associated with the application identifier to the match method of the trigger (2). When the arbiter finds a trigger that returns a nonempty action list from the match method, it passes the returned action list along with the message descriptor to the trigger’s activate method (3). The activate method is given write access to the data associated with the application identifier. The activate method modifies the data store (4) and returns descriptors.

Fig. 5. Internal structure of the message handling module

Fig. 6. Role of triggers in the SIP/WS middleware
of outgoing messages to the arbiter (5). The arbiter then sends the messages using the message handling module.

VI. APPLICATIONS OF THE SIP/WS MIDDLEWARE

In this section we present a short overview of Protocol Conversion and Coordination Language (PCCL) [6, 7], an XML based language designed specifically for defining SIP/WS application middleware triggers. Based on a PCCL definition, a PCCL compiler generates an executable trigger which can be loaded into the middleware trigger repository. Due to the constraint of limited space, we demonstrate the usage of the language only with several parts of the trigger definition that drives the sensor network example described in Section 3. A PCCL definition consists of three parts: definitions of SOAP services (WS element), definitions of matching conditions (protocol element), and definitions of actions (action element). The WS element defines which SOAP services may be invoked by the trigger, while the action element defines the extract sequence of actions which will be executed if a message satisfies the condition defined in the protocol element. Since information about a SIP endpoint must be inserted into headers of outgoing SIP messages, there is no SIP element. Instead, the source of the required header values is determined inside action elements. Fig. 7 presents the WS element of the PCCL definition for the sensor network Web Service. The element defines ws1 as the logical identifier of the service while the WSDL and location elements define the address of the service WSDL document and endpoint. The ID elements declare Web Service methods that can be invoked by the application middleware. The WS element is also used to define Web Services exposed by the middleware, in this use case the sendSensorData method.

```
<WS name="ws1">
  <wSDL>http://192.168.02/WS/sensor.asmx?WSDL</wSDL>
  <loc>http://192.168.02/WS/sensor.asmx</loc>
  <host>192.168.0.2</host>
  <ID soapAction="setSubscriptionStatus" />
  <ID soapAction="getSensorData" />  
</WS>
```

The main logic of the trigger is defined in the protocol and action elements. The protocol elements define which actions will be executed depending on the content and type of the received message. Fig. 8 presents the protocol element defining the one-time data pull operation.

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

The name attribute of the protocol element and type attribute of the message element define the protocol and message type for which an action is being defined. In this ex, an action is defined for incoming SIP SUBSCRIBE messages. For each message type, a sequence of condition elements is used to direct the middleware’s actions based on the message contents and the application and session state. For the one-time data pull operation, the Expires header must be equal to one, which is specified by the EQ element. The exec element lists the actions that must be executed if all the condition are met. In this case, all the logic is stored in a single action called OneTimePull, outlined in Fig.9.

```
<action name="OneTimePull" inType="SIP">
  <get type="string" location="header:method">"?room=$mySession.roomIDS"</get>
  <set state="OneTime Pull" />
  <send protocol="SIP" type="OK" />
  <send protocol="SOAP" name="getSensorData" 
        type="Request" service="ws1">" 
    <arg name="roomID" type="string">$mySession.roomIDS</arg>
  </send>
</action>
```

```
Fig 9. Action element for the one-time data pull operation
```

Each action element contains a sequence of get, set and send elements used to extract information from the received message, set session state and define outgoing messages. In the example, regular expressions are used in the get element for extracting the room identifier and storing it into session data, while the set element sets the session state to “Onetime Pull”. The first send element defines a SIP OK message, while the second send element references the sensor network Web Service through the service attribute, and uses the arg sub element to specify that the method argument value is the room identifier extracted from the received SIP request.
VII. CONCLUSION

The possibility to create applications that integrate IMS exposed services and Web services becomes increasingly important. Up to date, however, the principal way to implement such applications was to enable applications to communicate with both SIP and SOAP protocols used for accessing services in these two domains. The middleware present in this paper enables creating applications crossing boundaries of IP Multimedia and Web services systems. The middleware provides a modular message handling infrastructure, application state management and network resource management. The advantage of the developed middleware over existing solutions is the plug-in based system for creating applications. Application specific plug-ins defines the middleware behavior in response to received messages by changing its internal state and sending outing messages.

Furthermore, to ease the development of applications, we introduced an XML based language for defining plug-ins.

VIII. ACKNOWLEDGMENTS

This work was done during the course of M.TECH (CSE). G. Pullareddy Engineering College, Kurnool, A.p.

IX. REFERENCES