Application Middleware for convergence of IP Multimedia system and Web Services

Ivan Budiselic, Ivan Zuzak
School of Electrical Engineering and Computing
University of Zagreb
Zagreb, Croatia
E-mail: ivan.budiselic@fer.hr, ivan.zuzak@fer.hr

Ivan Benc
Ericsson Nikola Tesla d.d.
Zagreb, Croatia
E-mail: ivan.benc@ericsson.com

Abstract - Current network applications are typically created for one of two worlds. Communication applications targeting mobile devices usually communicate using the SIP protocol and are integrated into IP Multimedia systems of mobile network operators. On the other hand, applications targeting the enterprise market typically adhere to the SOAP protocol and integrate with Web Services exposed on the Internet. However, existing and future applications would benefit from access to services exposed by both of these protocols in both the mobile network and the Internet. In this paper we present the architecture of an application middleware that acts as a bidirectional gateway among IP Multimedia and Web Services systems. The middleware provides infrastructure for SIP and SOAP message handling, and session and network resource management. The middleware exposes interfaces for defining application specific rules for communication between protocol domains. Lastly, we outline a domain specific language that simplifies definition of such rules.

I. INTRODUCTION

In the last decade, the Internet has evolved from a network for serving documents and exchanging e-mail to a network of services. Driven by divergent user demands and expectations, services are built on top of different platforms and communicate using different protocols. 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 primarily in VoIP and video session management [1]. 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) [2]. IMS is a functional architecture designed for creation and deployment of multimedia telecommunication services over IP.

The Web Services standards family [3] defines a set of protocols for implementing systems based on the principles of Service Oriented Architecture (SOA) [4]. The basis of the Web Services stack [5] 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.

In this paper we propose an application middleware architecture for user defined and application specific interoperation of SIP and SOAP services [6,7]. Throughout this paper the term *application* refers to a system that provides its functionality through composition of one or more SIP services and Web Services. The main design goal for the architecture was to develop a general application middleware that enables definition of application specific bidirectional exchange of SIP and SOAP messages. We achieve this by separating the core logic for message parsing and generation from the application specific logic which controls inter-protocol message conversion and message flow between middleware users. Application specific logic is specified through user-defined plug-ins called *triggers* which must conform to a simple yet general interface. Message conversion triggers are loaded dynamically in order to enable a single middleware instance to serve different and multiple applications simultaneously. Furthermore, we designed an XML based domain specific language for specifying the message conversions and message flow which can be used for automated trigger generation.

The rest of the paper is structured as follows: Section 2 gives an overview of problems specific to integrating SIP and SOAP protocols and gives a survey of related approaches taken by other research projects. Section 3 describes an example use case for SIP and SOAP interoperation while Section 4 presents the architecture of the proposed application middleware. Section 5 summarizes the results of applying the general purpose application middleware to the use case presented in Section 3. Also, we give an overview of the benefits of the proposed architecture in integrating SIP and SOAP protocols. Finally, Section 6 concludes the paper.

II. 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 fundamental 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 Service (QoS) settings.

Contrary to the statefulness of SIP, one of the main principles in SOA is keeping services stateless [5]. 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 [8], 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. The details of state management in the proposed middleware architecture are described in Section 4.

Due to the popularity of both SIP and Web Services, a lot of research aims to integrate these protocols. In [10], the possibility of integrating SIP with multimedia Web Services is explored in a way that allows service initiation not only from SIP or SOAP nodes, but also through dynamic service binding,. However, the solution requires that all nodes implement both the SIP and Web Services protocol stacks which is a limiting factor in real-world applications.

A similar approach was taken in the *Web Service SIP (WSIP) concept* [11] where each node implements both the SIP and Web Services stack. In a particular session, the client maintains separate SIP and SOAP processes that communicate within protocol bounds with the server's SIP and SOAP processes, respectively. The authors consider an alternative approach in which SOAP messages could be embedded into the Session Description Protocol (SDP) part of SIP INVITE messages. However, they dismiss the option since there is no standard in this area which decreases the interoperability of the services. They also acknowledge the problem of storing session state and propose that state be inserted into SOAP message envelopes.

In a more recent paper [12], the same group of authors describes a new protocol called Web Services Initiation Protocol (WSIP) that is also a dual-stack solution. To support sessions, WSIP relies on WS-Session [13], a standardized extension to the basic WS stack which specifies how a Web Service based session establishment will result in a unique *sessionID* 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.

A different approach to accessing mobile Web Services trough SIP by relying on the IP Multimedia Subsystem (IMS) Service Architecture Specification is described in [14]. The IMS is used to provide mobility and session management, as well as message routing, security and billing. The authors introduce *Mobile Web Services* as Web Services that are accessed by a mobile device. These mobile devices are assumed to have a full Web Services protocol stack installed, so no special provisions are required for consuming a Web Service.

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.

III. 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 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 Service provider 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 (1) with the *Expires* header set to 1 indicating a one-time pull operation. The gateway parses the received SIP message,

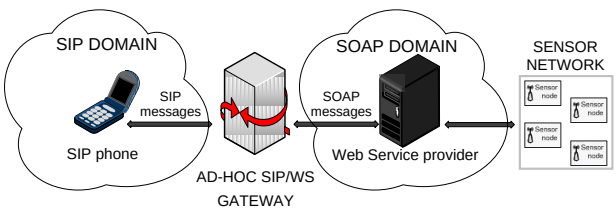

Fig. 1. SIP/WS integration use-case

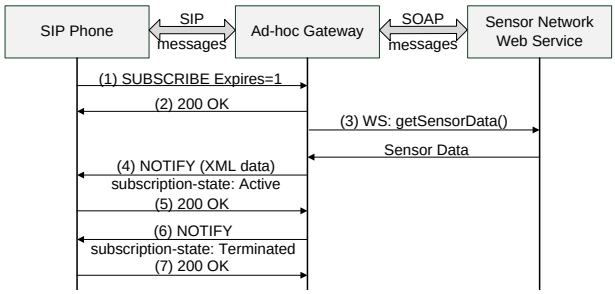

Fig. 2. Synchronous data retrieval

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 (the *Call-ID* header of the SIP SUBSCRIBE message) and the TCP connection it opens to the Web Service. After the Web Service 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. 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 *sendSensorData* method with the event XML message (6). Finally, the gateway relays the event description to the client (7) and the session is terminated (8-10).

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 context 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.

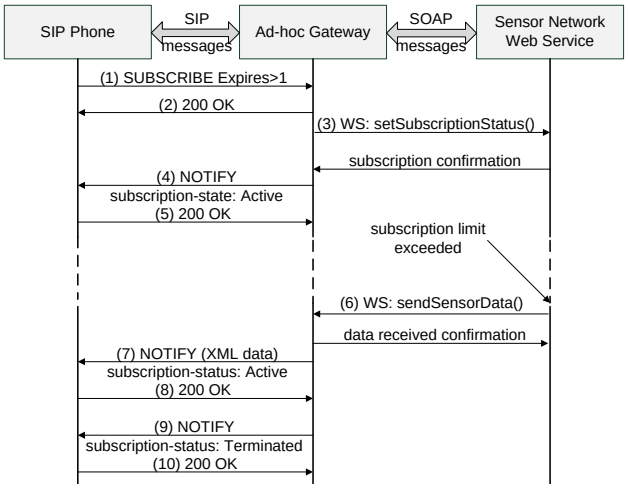

Fig. 3. Asnychronous event notification

IV. SIP/WS APPLICATION MIDDLEWARE

Fig. 4 shows the architecture of the developed SIP/WS application middleware. The *generic 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.

Therefore, the middleware is used in two ways – application developers create and store triggers into the trigger repository, while clients use the middleware for

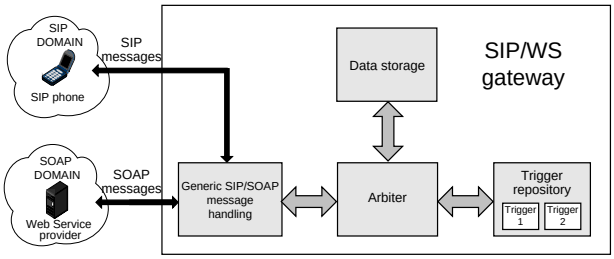

Fig. 4. SIP/WS application middleware modular structure

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 outgoing 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 *recieveSIP* 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 *inSIP* 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 send a SIP message, the arbiter constructs a data structure describing the outgoing SIP messages and pushes it into the *outSIP* queue (5). The *SIP generator* (6) takes the data structure from the *outSIP* queue, constructs the corresponding SIP message and pushes it into the *sendSIP* 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 requests 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.

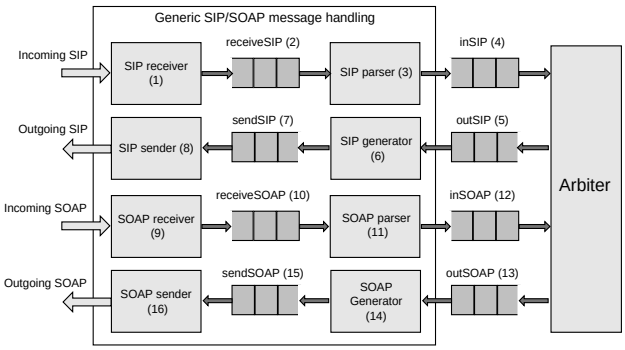

Fig. 5. Internal structure of the message handling module

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 activate 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

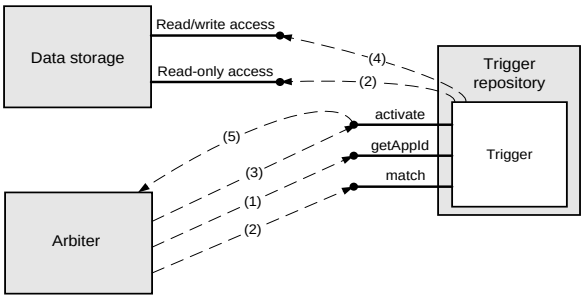

Fig. 6. Role of triggers in the SIP/WS middleware

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 unloading 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 of outgoing messages to the arbiter (5). The arbiter then sends the messages using the message handling module.

V. 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 exact 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>
```

Fig. 7. Web Service definition

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

Fig. 8. Protocol element for the one-time data pull operation

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 example, 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 conditions 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="$mySession.roomID$""
  </get>
  <set state="One Time Pull" />
  <send protocol="SIP" type="OK" />
  <send protocol="SOAP" name="getSensorData"
        type="Request" service="ws1">
    <arg name="roomID" type="string">
      $mySession.roomID$
    </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 *"One Time 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.

VI. 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 presented 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 define the middleware behavior in response to received messages by changing its internal state and sending outgoing messages. Furthermore, to ease the development of applications, we introduce an XML-based language for defining plug-ins.

VII. ACKNOWLEDGMENTS

This work was done during the course of Summer Camp 2007 jointly organized by the Ericsson Nikola Tesla d.d. and Faculty of Electrical Engineering and Computing, University of Zagreb.

REFERENCES

[1] J. Rosenberg, R. Shockey, “The Session Initiation Protocol (SIP): A key component for Internet telephony”, Computer Telephony, vol. 8, issue 6, Jun. 2000.

[2] G. Bertrand, “The IP Multimedia Subsystem in Next Generation Networks”, May 2007., http://www.rennes.enst-bretagne.fr/~gbertran/files/IMS_an_overview.pdf

[3] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana, “Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI”, IEEE Internet Computing, vol. 6, no. 2, pp. 86–93, Mar./Apr. 2002.

[4] M. N. Huhns, M. P. Singh, “Service-oriented computing: key concepts and principles”, IEEE Internet Computing, vol. 9, Issue 1, pp. 75–81, Jan./Feb. 2005.

[5] T. Earl, “SOA – Principles of Service Design”, Prentice Hall, Jul. 2007., ISBN: 9780132361132

[6] I. Benc, I. Skuliber, T. Stefanec, “System for dynamically adaptive multi-way conversion and coordination of SIP and SOAP”, patent pending

[7] I. Budiselic, G. Delac, D. Sego, T. Stefanec, “SIP/WS interworking triggering gateway”, Summer Camp 2007 Book of abstracts “New generation network applications & protocols”, ISBN: 9531841209

[8] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, T. Storey, S. Weerawaranna, “Modeling stateful resources with WebServices”, Version 1.1, Globus Alliance, May 2004.

[9] T. Stefanec, “ABNF parser generator” Summer Camp 2006 Book of abstracts “One step ahead: Advanced applications and network support functions”, ISBN: 9531841098

[10] H. Cai, W. Lu, B. Yang, L. Tang, “Session Initiation Protocol and Web Services for next generation multimedia applications,” IEEE Fourth International Symposium on Multimedia Software Engineering, Dec. 2002., pp.70

[11] F. Liu, W. Chou, L. Li, J. Li., “WSIP - Web service SIP endpoint for converged multimedia/multimodal communication over IP,”, IEEE International Conference on Web Services, Jul. 2004., pp. 690-697

[12] W. Chou, L. Li, F. Liu, “Web services methods for communication over IP”, Proceedings of the IEEE International Conference on Web Services, Jul. 2007., pp. 372-379

[13] ECMA international, “WS-Session - Web Services for Application Session Services”, 2nd edition, Jun. 2008., http://www.ecma-international.org/publications/standards/Ecma-366.htm

[14] G. Gehlen, F. Aijaz, Y. Zhu, and B. Walke, “Mobile P2P Web Service creation using SIP”, Fourth International Conference on Advances in in Mobile Computing and Multimedia, Dec. 2006, pp. 39-48