Programmable End System Services Using SIP

Xiaotao Wu
Department of Computer Science
Columbia University
New York, New York 10027
Email: xiaotaow@cs.columbia.edu

Henning Schulzrinne
Department of Computer Science
Columbia University
New York, New York 10027
Email: hgs@cs.columbia.edu

Abstract— **In Internet telephony, end systems can take a much larger role in providing services than in traditional telephone systems. We analyze the importance of end system services and describe the services and the Service Logic Execution Environment (SLEE) implemented in our SIP user agent, SIPC. Since we believe that end system services differ in their requirements from network services, we define a new service creation scripting language called Language for End System Services (LESS). Compared with other service creation languages, LESS is extensible, can be easily understood by non-programmers and contains commands and events for direct user interaction and the control of media applications.**

I. INTRODUCTION

One of the key promises of Internet telephony lies in the ability of developing and deploying innovative services rapidly and efficiently. Internet telephony services are not limited to those performed in servers operated by service providers; end systems can play a much larger role in providing services than in traditional telephone networks. In Internet telephony systems, end systems could be PCs or embedded SIP (Ethernet) phones with CPU and memory. End systems can execute programs to perform call control and other telecommunication services. For example, our SIP user agent, SIPC, can execute SIP CGI [11] programs or Call Processing Language (CPL) [9] [10] scripts to automatically handle SIP requests and responses. The Pingtel Xpressa SIP phones allow users to upload Java class files to perform services such as caller-specific ring tones.

We define end system services as the services that are performed in IP telephony end systems such as workstations, PDAs, Ethernet phones or Internet-connected appliances and network services as the services that are performed in network servers such as SIP [13] proxy servers.

Unlike network services, end system services can directly control media applications and interact with users. This allows them to completely automate all aspects of a call. For example, only an end system service can automatically accept a call based on address information.

Enabling end system services not only provides additional user convenience, but also encourages service innovation. In general, while it is difficult for subscribers to modify network services that are operated by carriers, they can install and modify services on end systems they own.

The development of end system services depends on the underlying protocols used for call signaling. SIP promotes end system services because it allows end-to-end operations: two SIP user agents (UAs) can talk to each other directly. SIP header fields, which can be inserted and modified by end systems, can control services in end systems without interference from proxy servers. The simplicity of SIP also makes it easy to develop services in end systems since end systems often have limited computational capabilities. In Section II, we show the services and Service Logic Execution Environment (SLEE) provided in our SIP user agent, SIPC.

Since traditional service creation methods address the needs of carriers with trained personnel, we defined a new XML-based script language called Language for End System Services (LESS) specifically for end system service creation (Section III). We then compare it with other languages.

II. END SYSTEM SERVICES AND SERVICE LOGIC EXECUTION ENVIRONMENT IN SIPC

SIPC is a SIP user agent developed at Columbia University. It can be used for Internet telephony calls, multimedia conferences, instant messaging, shared web browsing and control of network appliances.

A. End system services in SIPC

SIPC offers call control and presence-related services, emergency services, network appliance control and shared web browsing. We describe them in detail below.

Call control services: Call control services define how incoming and outgoing signaling messages related to a call session are handled. The call control services implemented in SIPC include many services defined in ITU-T recommendation Q.1211 [6] such as abbreviated dialing (ABD), automatic call back (ACB), call forwarding on busy/don't answer (CFC), customized ringing (CRG), originating call screening (OCS), terminating call screening (TCS), just to name a few.

Presence-related services: SIPC is a presence user agent (PUA) that can notify others about the user's presence state, such as online or offline, and receive notifications. With the support of CPL extensions for presence [19], SIPC can automatically set up calls when a subscribed-to user comes online.

Emergency services: An end system can connect to devices, for example, a fire detector or a temperature monitor, to detect emergency events and use the SIP event notification architecture for emergency notification [14]. In addition, the end system can issue device control commands to handle the emergencies. In SIPC, we use SOAP messages [15] as

0-7803-7802-4/03/$17.00 (C) 2003 IEEE

the content of emergency notifications. When SIPC gets an emergency event, it can, for example, execute a script to flash lights or close fire doors.

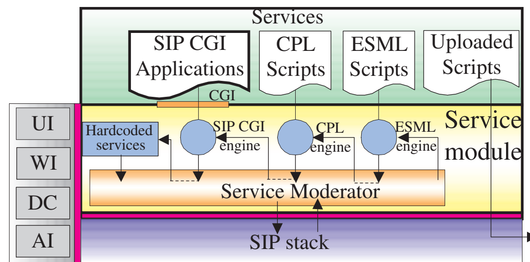
Integration with other Internet services: Several SIP headers can contain URIs that are resolved by the end system. For example, a URI in the Call-Info header describes the caller or callee. A HTTP URL in a Contact header can forward a call to a web page. In addition to using SIP headers to access call-related web information, we defined a mechanism to use the SIP MESSAGE method for shared web browsing [17]. Shared web browsing allows a group of people to visit the same web sites. SIPC can interact with the local web browser to retrieve the URL currently being viewed. The visited URL is then sent to the remote party via the SIP MESSAGE method. The remote SIPC will instruct its local web browser to visit the same URL.

Conference control: An end system can also cooperate with network servers to perform conference control services. We are developing floor control services using SIP events and SOAP [20]. With SIPC, the moderator can grant or revoke floors, participants can claim a floor and watch its status.

B. The service execution environment in SIPC

Several service interfaces and languages have been defined that allow the creation of services for SIP-based systems. These include SIP CGI, SIP servlets [7] and CPL. Because SIP servlets are built upon Java, while SIPC is implemented in C/C++ and Tcl/Tk, SIPC only supports SIP CGI and CPL. SIP CGI inherits the HTTP Common Gateway Interface model [3]. It uses the CGI to exchange information between service applications and SIP entities. For an incoming SIP message, the SIP entity invokes the service application and conveys message information to it through environment variables. The service application then instructs the SIP entity via commands written to standard output.

Since SIP CGI scripts allow general-purpose languages and imposes no restrictions on scripts, they are ill-suited for untrusted third parties. In contrast, CPL is an XML-based language that is intentionally limited in its capabilities, supporting neither loops nor variables nor recursion.



UI: User Interaction WI: Web Interaction
DC: Device Control AI: Application Interaction

Fig. 1. Architecture of end system service execution environment

Figure 1 shows the architecture of the end system service execution environment in SIPC. The services include SIP CGI applications, CPL scripts, LESS (see Section III) scripts and the scripts uploaded to SIP servers. By using the SIP PUBLISH method, a SIP UA can send service scripts to network servers such as SIP proxy servers, saving a local copy of the uploaded services. The service creation environment is responsible for examining all the services for potential conflicts, though the service execution engines will also check at runtime.

The service engines in SIPC do not interact with the signaling module directly, rather, they communicate with the service moderator. The service moderator is responsible for prioritizing services and resolving any conflicts. SIPC executes services in the order LESS, CPL, SIP CGI and finally hard-coded services. We chose this order so that the script most suitable for end systems is executed first.

III. Language for End System Services (LESS)

A. Motivation

Many existing service languages are designed for network services. The call model for end system services and network services differ. Figure 2 shows the models of a two-party call for a network service and an end system service. A network service establishes connections between multiple addresses, while end system services instruct the local application to send media to and receive media from remote addresses. The different call model implies different states, events and actions for services, thus a network service script is usually unsuitable for end systems and vice versa.

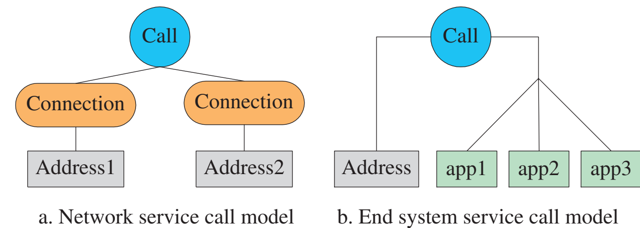


Fig. 2. Call models of network services and end system services

In addition, the two kinds of services have different developers. Network services are usually implemented by experienced programmers so the functional richness of the service language is more important than its simplicity. On the other hand, end system services are usually developed by non-programmers, making simplicity a requirement.

B. Requirements for an end system service language

An end system service language needs to be simple and easy to understand by non-programmers. Users would like to move services they created to different platforms, so a platform-neutral high-level language is called for. The language needs to be able to express user interactions and control media and other end system applications. It should be extensible to accommodate new services. Since it is restricted to a certain class of services, it does not have to be Turing-complete.

0-7803-7802-4/03/$17.00 (C) 2003 IEEE

C. Design strategy of LESS

The development of the Internet and the rise of the eXtensible Markup Language (XML) as a language standard have prompted proposals that XML-based scripting languages be used for creating telecommunications services. Among other advantages, XML is platform, network and technology neutral, independent of underlying programming languages, and readable by machines as well as humans. For these reasons, we based LESS on XML.

LESS inherits the tree-like structure from CPL. Like CPL, it avoids the use of loops, variables and recursion to allow program inspection and the back-and-forth translation between a graphical and textual representation.

To make LESS extensible, we use packages to group LESS events and actions. Initial packages support general multimedia call and user interaction.(Section III-E).

LESS is defined as an XML schema rather than a DTD. Schemas allow the derivation of new types from existing ones, so LESS can derive a new package from an existing package. For example, the triggers and actions in the **presence** package can be derived from the **event** package, which we will explain in detail in Section III-E.2. Also, XML schema provides predefined data types, such as datetime and list, making it easier to define LESS and validate LESS scripts.

```
<LESS:LESS
 xmlns:LESS="urn:ietf:params:xml:ns:LESS"
 xmlns:Generic="...:ns:LESS:generic"
 xmlns:Presence="...:ns:LESS:presence"
 xmlns:UI="...:ns:LESS:ui"
 xmlns:xsi="http://.../XMLSchema-instance"
 xsi:schemaLocation="......"
 name="xyzOnlineCall" priority="0.8">
 <subaction name="generalCall">
  <Generic:call/>
  <UI:alert message="Calling %address%"/>
 </subaction>
 <Presence:notification
   direction="incoming" package="presence">
  <Presence:presence-switch>
   <event package="presence" is="OPEN">
    <LESS:address-switch field="origin">
     <address is="sip:xyz@foo.com">
      <sub ref="generalCall"/>
     </address>
     <otherwise>
      <UI:alert message="%address% online"/>
     </otherwise>
    </LESS:address-switch>
   </event>
  </Presence:presence-switch>
 </Presence:notification>
</LESS:LESS>
```

Fig. 3. LESS service example

D. Elements of LESS

Figure 3 shows a simple LESS script. When `sip:xyz@foo.com` is online, the script automatically places an outgoing call to `sip:xyz@foo.com` and alerts the user. The `name` attribute in `LESS:LESS` tag provides a reference of the feature. Users can enable or disable features by names. The `priority` attribute in the `LESS:LESS` tag is used to solve feature conflicts (Section III-F). The `Presence:notification` tag indicates an incoming presence notification. The `Presence:presence-switch` and `event` tags check the status of the event. The `LESS:address-switch` and `address` tags check whether the notification is from `xyz@foo.com`. The `sub` tag refers to a `subaction` named generalCall. Reference to a `subaction` likes performing a function call in C language. The generalCall subaction places an outgoing call and alerts the user.

The example shows that LESS consists of three basic elements, namely triggers, switches, and actions. Triggers determine whether a LESS script should be executed. For example, in Figure 3, the `Presence:notification` trigger indicates that the script gets executed only for an incoming presence notification. Triggers can be invoked by call signaling, user interaction, timers or other applications. Switches represent choices a LESS script can make. For example, in Figure 3, the `LESS:address-switch` checks whom the notification is from and performs different actions accordingly. Actions place, redirect or reject calls or provide abstract user interface actions such as alerting or user decisions.

E. LESS packages

LESS groups triggers, switches and actions into packages. A LESS engine may support only a subset of packages. The namespace declarations in the `LESS:LESS` tag indicate which packages the script uses. Each package has its own namespace, avoiding naming conflicts for extensions.

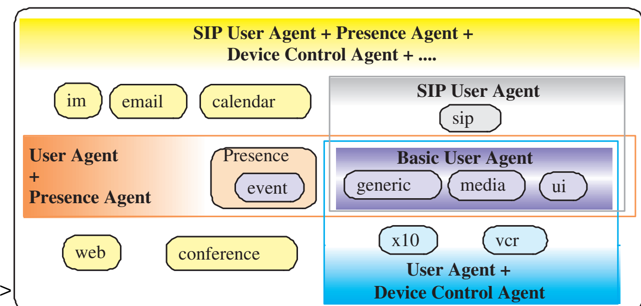


Fig. 4. LESS packages for a full-function end system

As shown in Figure 4, a simple user agent only needs to support the **generic**, **media** and **ui** packages. A SIP user agent adds the **sip** package. The **presence** and **im** packages support presence and instant messaging agents. The **presence** package is based on the **event** package, which provides general event notification handling. A device control agent needs device

packages. For example, with a serial-to-IR controller like the Slink-e, an end system can send infrared signal out and control devices, like VCR. The **vcr** package, containing commands such as play, record and fast forward, is then introduced for VCR control. With the CM11A computer interface, an end system can control X10 powerline devices and the **x10** package is introduced. Other packages include the **conference**, **web**, **email** and **calendar** packages, supporting conference control, interaction with web content, email and calendaring services, respectively.

Below, we briefly introduce the triggers and actions of the **generic**, **event** and **presence** packages and present how to derive the **presence** package from the **event** package.

1) GENERIC *package:* The **generic** package covers most call control services, such as accepting, forwarding, rejecting, terminating calls and placing outgoing calls. It can handle triggers for timers and incoming and outgoing calls,

2) EVENT *package and* PRESENCE *package:* The **event** package is based on the SIP event notification architecture [12]. It introduces two triggers, namely `subscription` and `notification`. The new triggers handle incoming or outgoing subscriptions and notifications respectively. It also defines a new switch named `event-switch` making choices based on the event value.

The **presence** package is derived from the **event** package. Figure 5 shows the basic TriggerType defined in the basic LESS schema. Figure 6 shows how the `notification` trigger in the **event** package extends the basic TriggerType with adding two new attributes. Figure 7 shows that in the **event** package's `notification` trigger, with restricting the `package` attribute to a fixed value `presence`, we get the **presence** package's `notification` trigger.

```
<xs:complexType name="TriggerType"
                abstract="true">
 <xs:group ref="Node"/>
</xs:complexType>
```

Fig. 5. TriggerType in the basic LESS schema

```
<xs:complexType name="NotificationType">
 <xs:complexContent>
  <xs:extension base="LESS:TriggerType">
   <xs:attribute name="direction"
    type="LESS:DirectionType"/>
   <xs:attribute name="package"
    type="xs:string" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Fig. 6. 'notification' trigger in **event** package

```
<xs:complexType name="NotificationType">
 <xs:complexContent>
  <xs:restriction
    base="LESSevent:NotificationType">
   <xs:group ref="LESS:Node"/>
   <xs:attribute name="direction"
    type="LESS:DirectionType"/>
   <xs:attribute name="package"
    type="xs:string" fixed="presence"/>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>
```

Fig. 7. 'notification' trigger in **presence** package

F. Handling feature interaction in LESS

The `priority` attribute in the `event` tag helps to resolve conflicts when multiple scripts subscribe to the same event. The script with highest priority gets executed first. If the actions defined in the higher-priority script cannot be performed (e.g., the switches cannot be matched), the LESS engine will continue to execute the lower-priority scripts.

G. Creating LESS services

Users can use a text editor to create LESS service scripts directly. However, it is more efficient to break the service creation process into two stages. In the first stage, a service template is created, represented in Figure 8 as `service.less`. The service template is written in LESS, but uses conventions for user-configurable values. For example, `<address is="${var}">` means that the address value is configurable. `${var}` is then replaced by user input. The second stage is to configure the service template and generate the LESS scripts. Users can use a graphical editor to fill in variables for the template or the template can be translated to HTML page by eXtensible Stylesheet Language (XSL) and XSL Transformations (XSLT). When performing translation, using the `xsl:if` tag, the XSLT script checks the value of each attribute to see whether it is configurable. For configurable attributes, the XSLT generates an HTML `input` tag in the HTML file so that users can input values. The CGI program, `translate.cgi`, then translates the HTML file into a user-configured LESS script.

H. Comparison with other service creation languages

Several XML-based call control languages have been proposed, such as the Call Policy Markup Language (CPML) [1], Telephony Markup Language (TML) [4], CallXML [2], Call Processing Language (CPL), Service Creation Markup Language (SCML) [5], and Call Control eXtensible Markup Language (CCXML) [16]. Space constraints prevent a complete survey of all these languages. We observe that the more mature and interesting of these proposals are CPL, being standardized by the Internet Engineering Task Force (IETF), SCML, being developed by JAIN forum, and CCXML, being developed by the Voice Browser working group in the World Wide Web Consortium (W3C). Our examination of CPL, SCML and CCXML concludes that none of the three approaches provide enough support for end system services.

Among CPL, SCML and CCXML, CPL is the only fully specified language. It is designed to run on a server where

0-7803-7802-4/03/$17.00 (C) 2003 IEEE

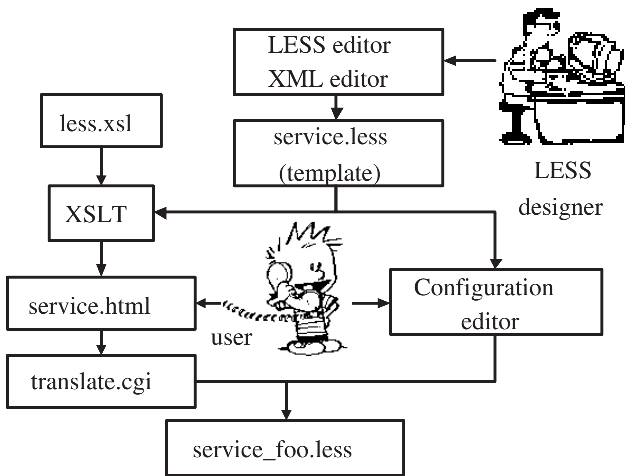


Fig. 8. Create LESS services

users may not be allowed to execute arbitrary programs, as it has no variables, loops, or ability to run external programs. These make it a suitable language for non-programmers. It can proxy, reject or redirect calls. However, CPL cannot originate a call, an important service for end systems. In addition, CPL cannot be activated through non-call events, such as timers.

LESS uses many switches defined in CPL. However, at the current stage, we cannot define LESS as a CPL extension. CPL is defined as a XML DTD and is not using the built-in data types in XML Schema, but LESS is defined as a XML Schema for extensibility and strict data type checking. In addition, CPL contains actions that cannot be executed in an end system, such as `proxy`. We are working on a CPL schema [18] that uses XML Schema built-in data types and separates the basic CPL elements, such as `address-switch`, `location`, into a basic CPL schema, and the elements specifically for proxy servers, such as `proxy`, into a CPL extension for proxy servers. With the modifications and the separation, we can define LESS as an extension of the basic CPL schema.

CCXML is designed to provide telephony call control support for dialog systems, such as VoiceXML systems, making it suitable for only a subset of end systems. The states and events for CCXML is in a lower level abstraction than those for LESS and CPL. For example, in CCXML, the call event is represented as sub-events such as `call.CALL_CONNECTED`, `call.CALL_ACTIVE`, `connection.CONNECTION_ALERTING`. Such signaling-derived events are too low-level for non-technical users.

At the moment, SCML is still a work in progress. SCML is developed by the JAIN forum. It is closely tied to the JAIN Java Call Control (JCC) API. and defined using an XML Schema that is derived from JCC. The object model of JCC [8] is the same as the network service call model in Figure 2.a. It focuses on how to build connections between addresses, not on how to instruct local applications. Unlike LESS, SCML has not defined how to extend the language for different end system applications.

IV. Conclusion and future work

In this paper, we have discussed the importance of end system services and described the services and service architecture in our SIP user agent, SIPC. We have developed SIP CGI and CPL as service interfaces in SIPC, and are designing a new scripting language, LESS, specifically for end system services. Compared with the other existing call control languages, LESS is based on a call model suited for end system services and offers simplicity, safety, extensibility and interaction with users, media applications and other end system applications. We plan to investigate how end system services are going to interact with existing network services and how to handle feature interactions between the end system and network services.

References

[1] Call policy markup language (cpml). http://xml.coverpages.org/cpml.html.
[2] Callxml. http://community.voxeo.com/docs/cxml/index.jsp.
[3] Common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi/interface.html.
[4] Telephony markup language (tml). http://xml.coverpages.org/tml.html.
[5] J.L. Bakker and Ravi Jain. Next generation service creation using xml scripting languages. In *Conference Record of the International Conference on Communications (ICC)*, New York City, New York, April 2002.
[6] International Telecommunication Union. General recommendations on telephone switching and signaling – intelligent network: Introduction to intelligent network capability set 1. Recommendation Q.1211, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, March 1993.
[7] JAIN. Sip servlet api. http://jcp.org/jsr/detail/116.jsp.
[8] Ravi Jain, Farooq M. Anjum, Paolo Missier, and Subramanya Shastry. Java call control, coordination, and transactions. *IEEE Communications Magazine*, 38(1), January 2000.
[9] J. Lennox and Henning Schulzrinne. Call processing language framework and requirements. RFC 2824, Internet Engineering Task Force, May 2000.
[10] J. Lennox and Henning Schulzrinne. CPL: a language for user control of Internet telephony services. Internet draft, Internet Engineering Task Force, November 2001. Work in progress.
[11] J. Lennox, Henning Schulzrinne, and J. Rosenberg. Common gateway interface for SIP. RFC 3050, Internet Engineering Task Force, January 2001.
[12] A. B. Roach. Session initiation protocol (sip)-specific event notification. RFC 3265, Internet Engineering Task Force, June 2002.
[13] J. Rosenberg, Henning Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.
[14] Henning Schulzrinne and Knarig Arabshian. Providing emergency services in Internet telephony. *IEEE Internet Computing*, 6:39–47, May 2002.
[15] W3C. Simple object access protocol (soap) 1.1. http://www.w3.org/TR/SOAP.
[16] W3C. Voice browser call control: Ccxml version 1.0. http://www.w3.org/TR/ccxml.
[17] X. Wu and Henning Schulzrinne. Use SIP MESSAGE method for shared web browsing. Internet draft, Internet Engineering Task Force, November 2001. Work in progress.
[18] X. Wu, Henning Schulzrinne, and J. Lennox. An extensible markup language schema for call processing language (CPL). Internet draft, Internet Engineering Task Force, November 2002. Work in progress.
[19] Xiaotao Wu, Henning Schulzrinne, Jonathan Lennox, and Jonathan Rosenberg. CPL extensions for presence. Internet draft, Internet Engineering Task Force, June 2001. Work in progress.
[20] Xiaoxin Wu et al. Use of session initiation protocol (SIP) and simple object access protocol (SOAP) for conference floor control. Internet draft, Internet Engineering Task Force, January 2003. Work in progress.

0-7803-7802-4/03/$17.00 (C) 2003 IEEE