XML for creating 3G Services (CPL)

Jose Costa Requena

Jose@tct.hut.fi
Laboratory of Telecommunications Technology
Helsinki University of Technology, P.O. Box 3000, FIN-02015 HUT, Finland

Abstract   –
This paper discusses the problem of defining new tools for the implementation of services in the third generation of mobile networks. For service implementation, as the main tool, we have used the XML language. XML is considered one of the best languages for describing complex data relationships. We have also chosen XML because it is easily extended flexible and because it has a text-based syntax. In concrete and defining a more complete programming language is used the Call Processing Language, which is based in XML. Thus, CPL gets the flexibility and extensibility of XML applied to define a logical and methodic tool for programming new services.

Keywords: 3G; XML; CPL; SIP;

INTRODUCTION

The Call Processing Language (CPL) is a language that can be used to describe and control Internet telephony services. The CPL is designed to be implemented either on network servers or user agent servers. It is meant to be simple, extensible, easily edited by graphical clients, and independent of operating system or signalling protocol. It is suitable for running on a server where users may not be allowed to execute arbitrary programs, as it has no variables, loops, or ability to run external programs.
There other alternatives for implementing services but they either require more resources or are not under the control of the provider.

CPL is not tied to any particular signalling architecture or protocol; it is anticipated that it will be used with both SIP [1] and H.323 [2]. Basically, SIP and H.323 are the IP Telephony signalling protocols that will be used as the mechanism for transporting the CPL scripts. SIP is the signalling protocol chosen for UMTS networks. Thus it will be the protocol used in 3G mobile phones.

The following sections will describe the relation between the XML and CPL. It also shows the entities of the CPL language within a Telephony network for defining new services. Furthermore, it shows some scripts examples and the parser structure for analysing the script and executing the service. Finally, it is depicted the procedure for registering the service for being run during following requests.

CPL BASIS

In this section it is described the basis of the CPL language. Basically, the CPL is a programming language itself. It is structured and it contains its own switches and triggers to perform different actions. In this section are presented the language hierarchy. The CPL as a language is formed by pre-defined words that have a specific meaning and cannot used for any other purpose. Hence, the word "switch" means that at this point the server that is running the script should take a decision. There are different types of switches based on time, location, priority or address and they are depicted in the following paragraph. Thus, when the script is executed and the parser is running through the script, it will be performed different type of actions based on the switches. Therefore, complex actions can be achieved based on the switch selection. The decision for going inside different nodes is set according to the user based on the time, location or address requirements.

The rest of the paragraph describes the structure of the CPL script based on those actions and switches.
Basically, a CPL script consists of two types of information: ancillary information about the script, and call processing actions.
Abstractly, a call processing action is described by a collection of nodes, which describe actions that can be performed or choices, which can be made. A node may have several parameters, which specify the precise behaviour of the node; they usually also have outputs, which depend on the result of the condition or action.

Nodes are arranged in a tree, starting at a single root node; outputs of nodes are connected to additional nodes. When an action is run, the action or condition described by the top-level node is performed; based on the result of that node, the server follows one of the node's outputs, and that action or condition is performed; this process continues until a node with no specified outputs is reached.

Some nodes have specific default actions associated with them; for others, the default action is implicit in the underlying signalling protocol, or can be configured by the administrator of the server.

Syntactically, CPL scripts are represented by XML documents. XML consists of a hierarchical structure of tags; each tag can have a number of attributes.

XML tags represent both nodes and outputs in the CPL; parameters are represented by XML tag attributes. Enclosing the tag representing the pointed-to node inside the tag for the outer node's output represents the connection between the output of a node and another node. Convergence (several outputs pointing to a single node) is represented by sub-actions. The higher-level structure of a CPL script is represented by tags corresponding to each piece of meta-information, sub-actions, and top-level actions, in order.

A CPL script list which appears as a top-level XML document is identified with the formal public identifier "-//IETF//DTD RFCxxxx CPL 1.0//EN". If this document is published as an RFC, "xxxx" will be replaced by the RFC number.

A CPL embedded as a fragment within another XML document is identified with the XML namespace identifier "http://www.ietf.org/internet-drafts/draft-ietf-iptel-cpl-03.txt". If this document is published as an RFC, the namespace identifier will be "http://www.rfc-editor.org/rfc/rfcxxxx.txt", where xxxx is the RFC number.

```
<?xml version="1.0" ?>
  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

  <cpl>
```

Language structure and tags

A call processing action is a structured tree that describes the decisions and actions that a telephony signalling server performs on a call set-up event. There are two types of call processing actions:
1. top-level actions are actions that are triggered by signalling events that arrive at the server.
    Two top-level action names are defined:
    *   incoming, the action performed when a call arrives whose destination is the owner of the script.
    *   outgoing, the action performed when a call arrives whose originator is the owner of the script.

2. Sub-actions are actions, which can be called from other actions.

Call processing actions, both top-level actions and sub-actions, consist of nodes and outputs. Nodes and outputs are both described by XML tags. There are four categories of CPL nodes: switches , which represent choices a CPL script can make; location modifiers , which add or remove locations from the location set; signalling actions , which cause signalling events in the underlying protocol; and non-signalling actions, which take an action but do not effect the underlying protocol.

*   Switches represent choices a CPL script can make, based on either attributes of the original call request or items independent of the call. All switches are arranged as a list of conditions that can match a variable. Each condition corresponds to a node output; the output points to the next node to execute if the condition was true. The conditions are tried in the order they are presented in the script; the output corresponding to the first node to match is taken.
    1. Address switches allow a CPL script to make decisions based on one of the addresses present in the original call request.

Node:	address-switch	
Outputs:	address	Specific addresses to match
Parameters:	field	origin, destination, or original-destination
	Subfield	address-type, user, host, port, tel, or display, (also: password and alias-type)
Output:	address	
Parameters:	is	exact match
	Contains	substring match (for display only)
	subdomain-of	sub-domain match (for host, tel only)

2. Time switches allow a CPL script to make decisions based the time and/or date the script is being executed. Time switches are independent of the underlying signalling protocol. Time switches are based on a large subset of how recurring intervals of time are specified in the Internet Calendaring and Scheduling Core

3. Object Specification (iCal COS), RFC 2445 [13].

Node:	time-switch	
Outputs:	time	Specific time to match
Parameters:	tzid	RFC 2445 Time Zone Identifier
	Tzurl	RFC 2445 Time Zone URL
Output:	time	
Parameters:	dtstart	Start of interval (RFC 2445 DATE-TIME)
	dtend	End of interval (RFC 2445 DATE-TIME)
	duration	Length of interval (RFC 2445 DURATION)
	freq	Frequency of recurrence (one of "daily", "weekly", "monthly", or "yearly")
	interval	How often the recurrence repeats
	until	Bound of recurrence (RFC 2445 DATE-TIME)
	byday	List of days of the week
	bymonthday	List of days of the month
	byyearday	List of days of the year
	byweekno	List of weeks of the year
	bymonth	List of months of the year
	wkst	First day of workweek

4. Priority switches allow a CPL script to make decisions based on the priority specified for the original call.

Node:	priority-switch	
Outputs:	priority	Specific priority to match
Parameters:	none	
Output:	priority	
Parameters:	less	Match if priority is less than specified
	greater	Match if priority is greater than specified
	equal	Match if priority is equal to specified

- Location modifier. The behavior of several of the signalling actions is dependent on the current location set specified. Location nodes add or remove locations from the location set.

1. Explicit location nodes specify a location literally and are dependent on the underlying signalling protocol.

Node:	location	
Outputs:	any node	
Parameters:	url	URL of address to add to location set
	Priority	Priority of this location (0.0-1.0)
	clear	Whether to clear the location set before adding the new value

2. Location lookup. Locations can also be specified up through external means, through the use of location lookups.

Node:	lookup	
Outputs:	success	Action if lookup was successful
	Notfound	Action if lookup found no addresses
	Failure	Action if lookup failed
Parameters:	source	Source of the lookup
	Timeout	Time to try before giving up on the lookup
	use	Caller preferences fields to use
	ignore	Caller preferences fields to ignore
	clear	Whether to clear the location set before adding the new values
Output:	success	
Parameters:	none	
Output:	notfound	
Parameters:	none	

Basically, these are the different nodes and elements defined in the CPL naming. Furthermore, using these nodes and tags the user can define his service.

CPL IMPLEMENTED IN XML

After the introduction of the CPL basis we can easily see the similarity between CPL and XML. In fact the CPL is a programming language that completely fits into the XML structure. Furthermore, we could say that CPL is a XML structured language where are defined specific tags for achieving concrete actions.
Nowadays, other known programming languages such as PERL, TCL, HTML, SGML and C/C++ could have been chosen. So, why XML?

XML is more than a markup language it is a metalanguage. This means that XML is a language that allows you to describe languages.

XML lets developers to set standards defining the information that should appear in a document, and in what sequence. XML makes it possible to define the content of a document separately from its formatting, making it easy to reuse that content in other applications or for other presentation environments.

XML provides a basic syntax that can be used to share information between different kinds of computers, different applications, and different organizations without needing to pass through many layers of conversion.

XML provides a simple format that is flexible enough to accommodate diverse needs. Even developers performing tasks on different types of applications with different interfaces and different data structures can share XML formats and tools for parsing those formats into data structures that the applications can use. XML offers many advantages, including:

- **Simplicity**. XML documents are built upon a core set of basic nested structures. While the structures themselves can grow complex as layers and layers of detail are added, the mechanisms underlying those structures require very little implementation effort, from either authors or developers. Furthermore, XML rigid set of rules helps make documents more readable to both humans and machines.

- **Extensibility**. XML is extensible in two senses. First, it allows developers to create their own Document Type Definition (DTD), effectively creating 'extensible' tag sets that can be used for multiple applications. Second, XML itself is being extended with several additional standards that add styles (e.g. XSL), linking, and referencing ability to the core XML set of capabilities.

- **Interoperability**. XML can be used on a wide variety of platforms and interpreted with a wide variety of tools. Because the document structures behave consistently, parsers that interpret them can be built at relatively low cost in any of a number of languages such as C++, C, JavaScript, Tcl, and Python.

- **Openness**. XML documents are considerably open and anyone can parse a well-formed XML document.

- **Applications.** XML can be used in a couple of different ways. One is for data interchange between humans and machines, such as from a Web server to a user's browser. The other is for data exchange between applications, or from machine to machine.

  Linking possibilities could be also included in advantages list. To illustrate this, **Error! Reference source not found.** presents a script that rings a call at a standard location and, if the recipient is not available there, forwards the call to a voicemail server instead. Since we want the same action to occur on busy as on no answer, we define a link on one node. This allows other nodes to reference that link rather than repeat parts of the script.

```
<?xml version="1.0" ?>
  <!DOCTYPE call SYSTEM "ex.dtd">
  <call>
        <!-- Proxy the call to jose -->
      <location url="sip: jose@ josepc.example.com">
        <proxy timeout="8s">
              <!-- When busy, forward to voicemail -->
          <busy>
            <location url="sip: jose@voicemail.example.com" merge="clear"    id="voicemail" >
                 <proxy />
            </location>
          </busy>
              <!-- When there is no answer, jump to the voicemail link above and also forward to voicemail -->
          <noanswer >
                 <link ref="voicemail" />
          </noanswer >
        </proxy>
      </location>
  </call>
```

Figure 1. XML example

Document Type Definition (DTD) is a well-known self-describing and structured information required at the beginning of the validated scripts. The DTD identifies the root element of the document and may contain additional declarations. All XML documents must have a single root element that contains all of the content of the document. Additional declarations may come from an external DTD, called the external subset, or be included directly in the document, the internal subset, or both.

Writing in XML seems to be quite easy but also different technologies can be followed in this area: Document Type Definitions and XML schemas. Although, neither they are strictly required for XML development, both DTDs and XML Schemas are important parts of the XML toolbox. DTDs have been around for over twenty years as a part of SGML, while XML Schemas are relative newcomers. Though they use very different syntax and take different approaches to the task of describing document structures, both mechanisms definitely occupy the same turf. The W3C seems to be grooming XML Schemas as a replacement for DTDs, but it is not yet clear how quickly the transition will be made. DTDs are here-and-now, while XML Schemas, in large part, are just *coming*. Brother languages such as Perl, Tcl and so on could also wake up the attention of people describing data.

CPL AS NETWORK ELEMENT

After getting familiar with the CPL, its structure and how does it fits into XML, in this section is presented the CPL in a network environment.

The CPL is powerful enough to describe a large number of services and features, but it is limited in power so that it can run safely in Internet telephony servers. The intention is to make it impossible for users to do anything more complex (and dangerous) than describing Internet telephony services. The language is not Turing-complete, and provides no way to write loops or recursion.

The CPL is also designed to be easily created and edited by graphical tools. It is based on XML, so parsing it is easy and many parsers for it are publicly available. The structure of the language maps closely to its behavior, so an editor can understand any valid script, even ones written by hand. The language is also designed so that a server can easily confirm scripts' validity at the time they are delivered to it, rather that discovering them while a call is being processed.

Implementations of the CPL are expected to take place both in Internet telephony servers and in advanced clients; both can usefully process and direct users' calls. This document primarily addresses the usage in servers. A

mechanism will be needed to transport scripts between clients and servers; this document does not describe such a mechanism, but related documents will.

The Call Processing Language (CPL) is designed to be implementable on either network servers or user agent servers (UAS). It is simple, extensible, easily edited by graphical clients, and independent of operating system or signaling protocol. It is suitable for running on a server where users may not be allowed to execute arbitrary programs, as it has no variables, loops, or ability to run external programs.

Syntactically, CPL scripts are represented by XML documents, so parsing them is easy and many parsers for them are publicly available. The structure of the language maps closely to its behavior, so an editor can understand any valid script, even ones written by hand. The language is also designed so that a server can easily confirm scripts validity at the time they are delivered to it, rather that discovering them while a call is being processed.

The CPL Network Model

In this model, an Internet telephony network contains two types of components: end systems and signaling servers. End systems are devices, which originate and/or receive signaling information and media. An end system can originate, accept, reject a call, or forward incoming calls. Signaling servers are devices, which relay or control signaling information. In SIP, they are proxy servers, redirect servers, or registrars.

Signaling servers can perform three types of actions on call setup information. They can forward it on to one or more other network or end systems, returning one of the responses received (*proxy it*). They can also return a response informing the sending system of a different address to which it should send the request (*redirect it*). Finally, they can inform the sending system that the setup request could not be completed (*reject it*).

When an end system places a call, the call establishment request can proceed by a variety of routes through components of the network. To begin with, the originating end system must decide where to send its requests. There could be, for example, two possibilities: the originator may be configured so that all its requests go to a single local server; or it may resolve the destination address to locate a remote signaling server or end system to which it can send the request directly.

Once the request arrives at a signaling server, that server uses its user location database, its local policy, DNS resolution, or other methods, to determine the next signaling server or end system to which the request should be sent. A request may pass through any number of signaling servers: from zero (in the case when end systems communicate directly) to every server on the network.

SCRIPTS: What, Which, Where and How

In this section, we answer some questions to clarify scripts behavior:

1) **What** does a script do?

    Specifically, a script replaces the user location functionality of a signaling server. Signaling server typically maintains a database of locations where a user can be reached; it makes its proxy, redirect, and rejection decisions based on the contents of that database. A CPL script replaces this basic database lookup functionality; it takes the registration information, the specifics of a call request, and other external information it wants to reference (e.g. services requested), and chooses the signaling actions to perform.

    Abstractly, a script can be considered as a list of condition/action pairs; if some attribute of the registration, request, and external information matches a given condition, then the corresponding action is taken.

2) **Where** can users have scripts?

    Users can have CPL scripts on any network server which their call establishment requests pass through and with which they have a trust relationship. Scripts would typically perform different functions, related to the role of the server on which they reside.

3) **Which** script is executed in the server? And when?

CPL scripts are usually associated with a particular Internet telephony address. When a call establishment request arrives at a signaling server which is a CPL server, that server associates the source and destination addresses specified in the request with its database of CPL scripts; if one matches, the corresponding script is executed.

Once the script has been executed, if it has chosen to perform a proxy action, a new Internet telephony address will result as the destination of that proxying. Once this has occurred, the server again checks its database of scripts to see if any of them are associated with the new address; if one is, that script is also executed.

In general, in an Internet telephony network, an address will denote one of two things: either a user, or a device. A user address refers to a particular individual, for example *sip:jose@example.com*, regardless of where that user actually is or what kind of device she is using. A device address, by contrast, refers to a particular physical device, such as *sip:x26063@phones.example.com*.

IMPLEMENTING NEW SERVICES WITH CPL

After the initial overview of the CPL, its relationship with XML and the description of CPL as a network element, this section gives an overview of a real service implementation. This is a practical point of view for see a service implementation using the CPL and a XML parser. It shows the utilization of a XML parser at the server where it is executed the service. This part gives a technical picture of the parser execution, the elements and function used in the XML parser for going through the CPL script and extract the various switches on it. The parser is invoked and it will convert the script and the tags defined in the above section to other structures that are managed by the server to perform the signaling based on that.

To invoke the parser the first thing to do is to read an XML document. The parser accepts to parse both memory mapped documents or direct files. The functions are defined in "parser.h":

❑ *xmlDocPtr xmlParseMemory(char *buffer, int size);*

    parse a zero terminated string containing the document.

❑ *xmlDocPtr xmlParseFile(const char *filename);*

    parse an XML document contained in a file.

This returns a pointer to the document element or NULL in case of failure.
The way of traversing the tree to explore the complete XML document consists in jumping from one node to the others using the *xmlNodePtr*. Depending on the branch taken the final *text* output is different. There are multiple possibilities, following we present some examples.

    Document->root element->childs->childs

    Document->root element ->childs->next->childs->childs

    Document->root element->childs->next->next->childs

    Functions to modify the tree are also available.

❑ *xmlAttrPtr xmlSetProp(xmlNodePtr node, const xmlChar *name, const xmlChar *value);*

    This function sets or changes an attribute carried by an *element* node.

❑ *const xmlChar *xmlGetProp(xmlNodePtr node, const xmlChar *name);*

    This function returns a pointer to the property content.

    Two functions are used to read and write the text associated with elements:

❑ *xmlNodePtr xmlStringGetNodeList(xmlDocPtr doc, const xmlChar *value);*

> This function takes an "external" string and converts it to one text node or possibly to a list of entity and text nodes. All non-predefined entity references will be stored internally as an entity node; hence the result of the function may not be a single node.

❑ *xmlChar *xmlNodeListGetString(xmlDocPtr doc, xmlNodePtr list, int inLine);*

> This is the dual function, which generates a new string containing the content of the text and entity nodes When the script is received in the SIP server, the XML parser translates all the information about the user requirements to data structures. Those will be used to perform the right decision when another request is coming.
>
> Figure 2 depicts the steps followed by the parser to explore the *icw.xml* file. This file is represented in Figure 1 where is implemented a call waiting service. Looking at the top of the structure, the pointer *xmlDocPtr* stores the memory address where the xml script is deposited. The functions *xmlParseFile* and *xmlParseMemory* return this pointer. From this point the parser starts analysing the script. The *xmlNodePtr* guides to the memory addresses where the next nodes of the tree are stored. The *doc->root* function gets the root node of the tree, in our case the *<call>* tag. From the root, the parser can follow different branches according to the *next* and *childs* functions. The latter goes down one level in the *parents-children* hierarchy. Hence, from *<call>* the parser could take two possible *childs* nodes: *<proxy>* and *<response>*. Following the same criteria, the *<proxy>* node presents four possible *childs* branches (*<icw>*, *<busy>*, *<noanswer>* and *<failure>*) and, at the same time, the *<icw>* has three *childs* more (*<forward>*, *<success>* and *<reject>*). The structure continues like this with two *childs* more; *< link>* for *<forward>* and *<location>* for *<success>*.
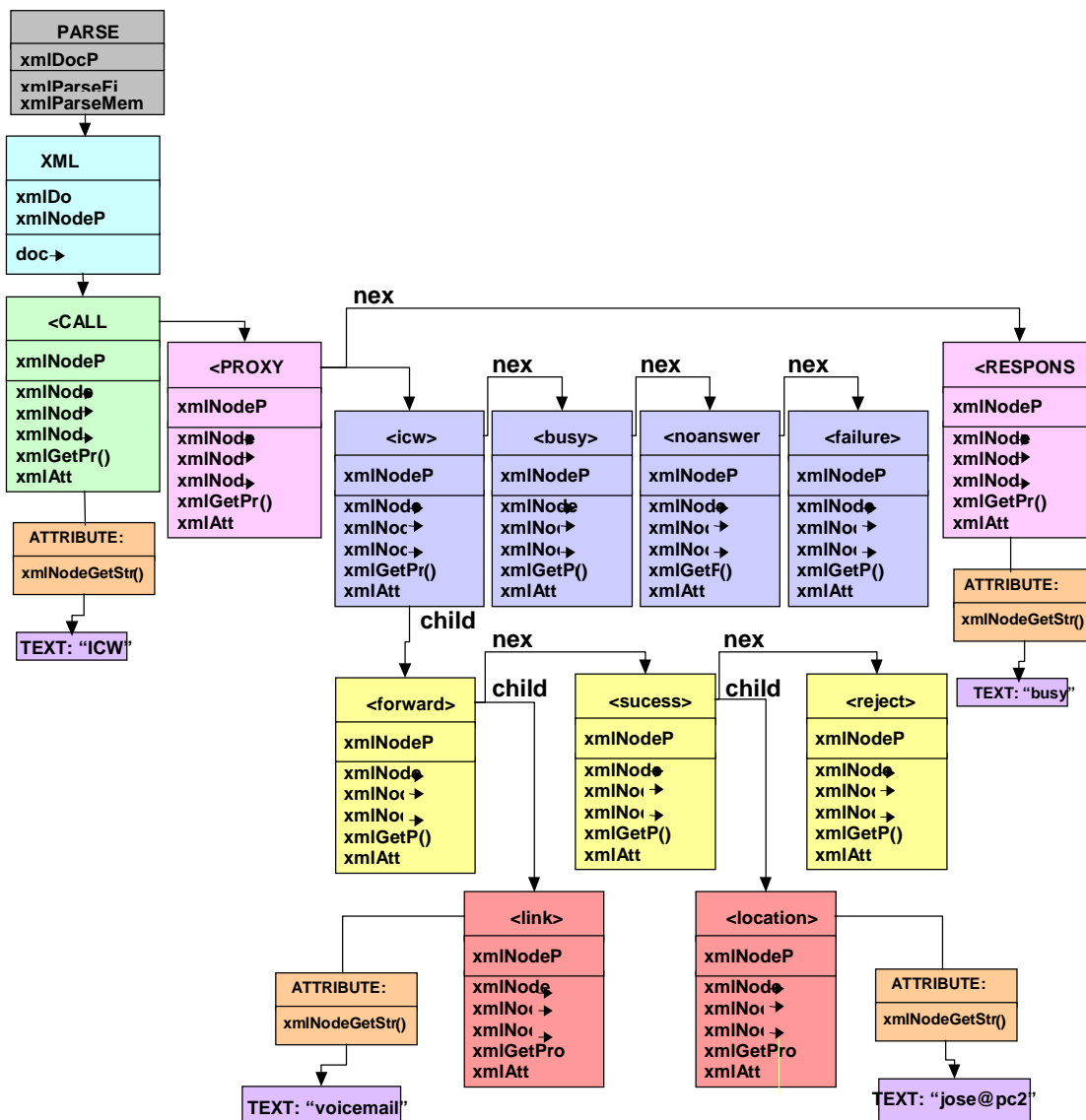

Figure 2. A tree built by the XML parser

Furthermore, *xmlGetProp()* function gives the content of the each node property, e.g. the content of *TYPE* in *<call>* would be *"ICW"*. On the other hand, *xmlNodeGetString()* function returns the attribute information between the same node tags, e.g. in *<reject>* node would be *The user is Busy now.*

USER TAILORED SERVICES

Thus, using CPL the user defines his own services within the CPL limitations but with it flexibility. The user will create the service will store it in the network server and the provider will execute it with enough security and under his control. At this point there is a trade off between the user freedom for designing the service and the provider for controlling the service execution. In other approaches the user can have more autonomy for creating the service but it is necessary and additional security mechanism for trusting the user. Otherwise, the provider does not rely on the script for executing in the server unless it is trusted previously. The next section presents few basic examples of telephony services.

Some examples of services already implemented

Example of Call redirect Unconditional Script.

```
<call>
            <location url="sip:smith@phone.example.com">
                        <redirect />
            </location>
</call>
```

Example time of day routing script:

```
<call>
            <time-switch>
            <!-- During the work week, contact the user at his registered locations -- >
                        <time day="1-5" timeofday="0900-1700">
                                    <lookup source="registration">
                                                <success>
                                                            <proxy/>
                                                </success>
                                    </lookup>
                        </time>
            <!-- The rest of the time, forward the call to voicemail -- >
            <otherwise>
                        <location url="sip:jones@voicemail.example.com">
                                    <proxy/>
                        </location>
            </otherwise>
            </time-switch>
</call>
```

USER LINKED SERVICES (Service Portability)

The main advantage of using CPL for defining new services is that the services are directly attached to the user. The user defines the specific characteristics that he wants to apply to his services. Furthermore, those descriptions are traduced to a XML script, which is stored at the user's device. Therefore, the user can move any place and his script that will be stored at the local provider with CPL capabilities. Thus, after the user registers in a new network the provider will interpret the script provided by the user and will execute the service based on the user requirements. The next section describes the process of registration and execution of the service in a SIP network.

Service registration

In a SIP based network the service works as follows. Firstly, the user has to send the registration information to the server. That information contains the XML script defining the concrete service, which will be stored in the SIP server to manage future calls. The following Figure 3 shows the process of the service registration. It depicts the

mechanism used by the user A where is running a SIP Client called IPtele, to register his service profile. The REGISTER is a method used by the signaling protocol (SIP) for registering the user in a new network. Thus the body of that message contains the CPL script. The message arrives to the SIP server that extracts the CPL script from the body and stores it in the Database. Thus, the CPL resides in the Database and will be run when a new call arrives addressed to that user. Then the script will drive the action to be taken with that call according to the requirements specified by the user in the script.

User A registers the Internet Call Waiting service sending the REGISTER message to the server (with the script defining the service in the body).

```
REGISTER sip:ericy.com SIP/2.0
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:queca@helsinkiu.ericy.com
To: sip:queca@helsinkiu.ericy.com
Call-ID: 6088@helsinkiu.ericy.com
Cseq: 1 REGISTER
Contact:<sip:queca@helsinkiu.ericy.com:5060;transport=udp>
Expires: 7200
Content-Length: 350
Content-Type: application/hut-cpl
<?xml version="1.0"?>
<call Type="ICW">
......
</call>
```
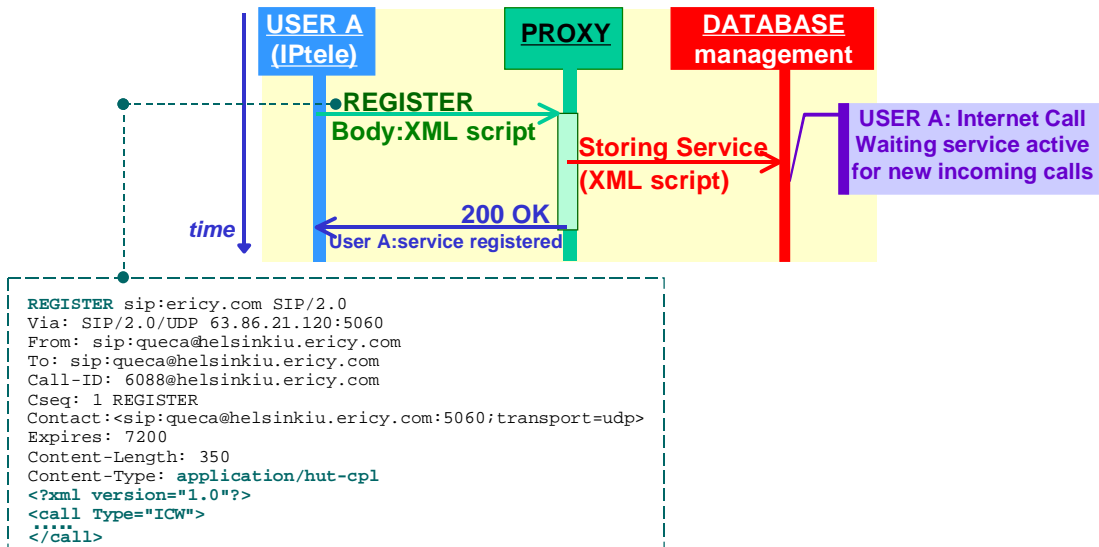
Figure 3.CPL script registration using SIP protocol

Thus, finally when the script is received in the SIP server, the XML parser translates all the information about the user requirements to data structures. Those will be used to perform the right decision when another request is coming. This case is presented in Figure 4, where a new call arrives meanwhile the user is already within an ongoing call. That user registered already a CPL script where it was indicated that between certain interval of time he does not want to be disturbed. Thus, based in the CPL service the new call will be reject automatically by the script without any interaction with the called user.
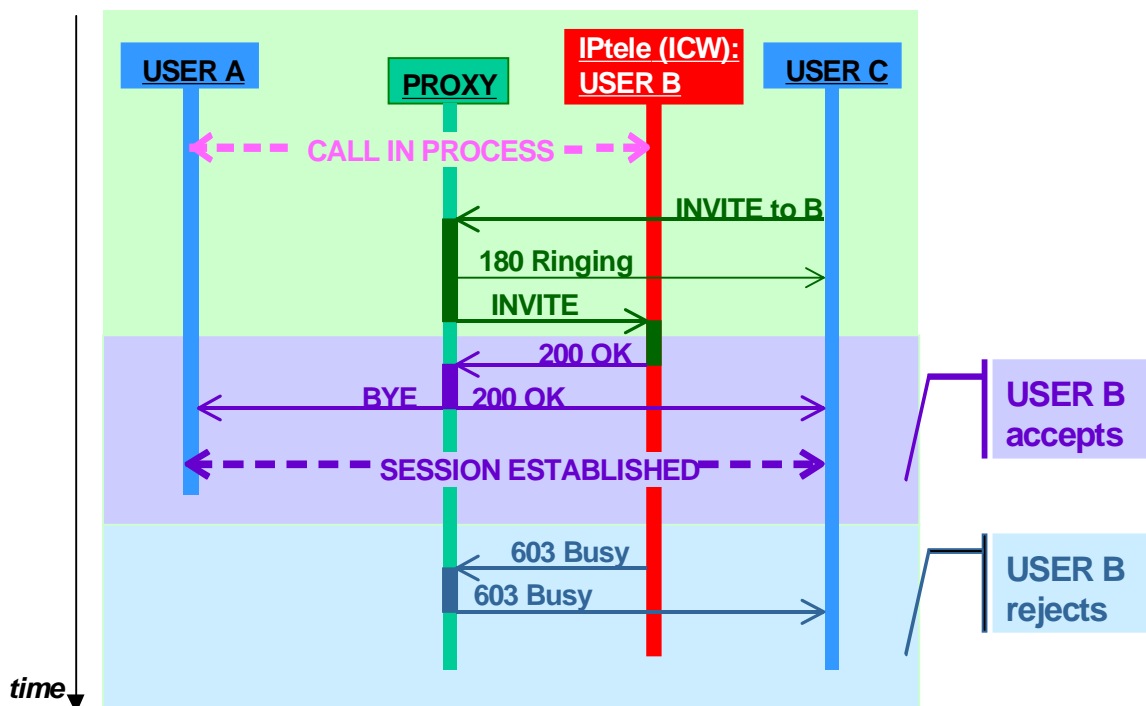
Figure 4. Execution of CPL script when a new incoming call arrives

In this case during the service registration it is also posible to use CGI scripts that basically contains similar features.
In this sense, both are viewed as different approaches for creating VoIP services. Both are written offline, and both are executed when messages arrive in order to execute features.

The difference is in the applicability. CPL is designed for end user service creation. Its purposefully limited in capabilities; its not a general purpose programming language. Its execution on a server is
generally very fast. CGI is more powerful - you can do nearly anything. Its programming language independent. It incurs a process-spawning overhead, so its less efficient than CPL, (CPL is usually executed in the same process as the server). As a service provider, I would not want to execute CGI scripts sent to me by end users. However, I would prefer to use CGI to develop my own services.
he question has to do with generic means for capabilities exchange between two devices, using a common set of semantics.

Thus the CPL is preferable in terms of security and efficiency at the execution level in the server executing the services.

REFERENCES

[1] CPL: A Language for User Control of Internet Telephony Services. draft-ietf-iptel-cpl-02.txt

[2] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. World Wide Web Consortium, 1998. (See http://www.w3.org/TR/REC-xml.)

[3] CPL service creation environments. http://www.cs.columbia.edu/~library/TR-repository/reports/reports-1999/cucs-010-99.pdf.

[4] RFC 2445. Internet Calendaring and Scheduling Core Object Specification (iCalendar)

[5] XML DTD for Roaming Access Phone Book, M. Riegel, G. Zorn, draft-ietf-roamops-phonebook-xml-04.txt

[6] I Espigares,J M Costa Requena, and R Kantola: New Tools for programming IP Telephony Services, accepted to IPTel2000.