Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-iptel-cpl-framework-02.ps

IPTEL WG
Lennox/Schulzrinne
Columbia University
January 25, 2000
Expires: July 2000

Call Processing Language Framework and Requirements

Status of this Memo

Copyright Notice

Abstract

A large number of the services we wish to make possible for Internet telephony require fairly elaborate combinations of signalling operations, often in network devices, to complete. We want a simple and standardized way to create such services to make them easier to implement and deploy. This document describes an architectural framework for such a mechanism, which we call a call processing language. It also outlines requirements for such a language.

Contents

1 Introduction

Recently, several protocols have been created to allow telephone calls to be made over IP networks, notably SIP [1] and H.323 [2]. These emerging standards have opened up the possibility of a broad and dramatic decentralization of the provisioning of telephone services so they can be under the user's control.

Many Internet telephony services can, and should, be implemented entirely on end devices. Multi-party calls, for instance, or call waiting alert tones, or camp-on services, depend heavily on end-system state and on the specific content of media streams, information which often is only available to the end system. A variety of services, however — those involving user location, call distribution, behavior when end systems are busy, and the like — are independent of a particular end device, or need to be operational even when an end device is unavailable. These services are still best located in a network device, rather than in an end system.

Traditionally, network-based services have been created only by service providers. Service creation typically involved using proprietary or restricted tools, and there was little range for customization or enhancement by end users. In the Internet environment, however, this changes. Global connectivity and open protocols allow end users or third parties to design and implement new or customized services, and to deploy and modify their services dynamically without requiring a service provider to act as an intermediary.

A number of Internet applications have such customization environments — the web has CGI [3], for instance, and e-mail has Sieve [4] or procmail. To create such an open customization environment for

Internet telephony, we need a standardized, safe way for these new service creators to describe the desired behavior of network servers.

This document describes an architecture in which network devices respond to call signalling events by triggering user-created programs written in a simple, static, non-expressively-complete language. We call this language a *call processing language.*

The development of this document has been substantially informed by the development of a particular call processing language, as described in [5]. In general, when this document refers to "a call processing language," it is referring to a generic language that fills this role; "the call processing language" or "the CPL" refers to this particular language.

2 Terminology

In this section we define some of the terminology used in this document.

SIP [1] terminology used includes:

invitation: The initial INVITE request of a SIP transaction, by which one party initiates a call with another.

redirect server: A SIP device which responds to invitations and other requests by informing the request originator of an alternate address to which the request should be sent.

proxy server: A SIP device which receives invitations and other requests, and forwards them to other SIP devices. It then receives the responses to the requests it forwarded, and forwards them back to the sender of the initial request.

user agent: A SIP device which creates and receives requests, so as to set up or otherwise affect the state of a call. This may be, for example, a telephone or a voicemail system.

user agent client: The portion of a user agent which initiates requests.

user agent server: The portion of a user agent which responds to requests.

H.323 [2] terminology used includes:

terminal: An H.323 device which originates and receives calls, and their associated media.

gatekeeper: An H.323 entity on the network that provides address translation and controls access to the network for H.323 terminals and other endpoints. The gatekeeper may also provide other services to the endpoints such as bandwidth management and locating gateways.

gateway: A device which translates calls between an H.323 network and another network, typically the public-switched telephone network.

RAS: The Registration, Admission and Status messages communicated between two H.323 entities, for example between an endpoint and a gatekeeper.

General terminology used in this document includes:

user location: The process by which an Internet telephony device determines where a user named by a particular address can be found.

CPL: A Call Processing Language, a simple language to describe how Internet telephony call invitations should be processed.

script: A particular instance of a CPL, describing a particular set of services desired.

end system: A device from which and to which calls are established. It creates and receives the call's media (audio, video, or the like). This may be a SIP user agent or an H.323 terminal.

signalling server: A device which handles the routing of call invitations. It does not process or interact with the media of a call. It may be a SIP proxy or redirect server, or an H.323 gatekeeper.

3 Example services

To motivate the subsequent discussion, this section gives some specific examples of services which we want users to be able to create programmatically. Note that some of these examples are deliberately somewhat complicated, so as to demonstrate the level of decision logic that should be possible.

- Call forward on busy/no answer

  When a new call comes in, the call should ring at the user's desk telephone. If it is busy, the call should always be redirected to the user's voicemail box. If, instead, there's no answer after four rings, it should also be redirected to his or her voicemail, unless it's from a supervisor, in which case it should be proxied to the user's cell phone if it is currently registered.

- Information address

  A company advertises a general "information" address for prospective customers. When a call comes in to this address, if it's currently working hours, the caller should be given a list of the people currently willing to accept general information calls. If it's outside of working hours, the caller should get a webpage indicating what times they can call.

- Intelligent user location

  When a call comes in, the list of locations where the user has registered should be consulted. Depending on the type of call (work, personal, etc.), the call should ring at an appropriate subset of the registered locations, depending on information in the registrations. If the user picks up from more than one station, the pick-ups should be reported back separately to the calling party.

- Intelligent user location with media knowledge

  When a call comes in, the call should be proxied to the station the user has registered from whose media capabilities best match those specified in the call request. If the user does not pick up from that station within four rings, the call should be proxied to the other stations from which he or she has registered, sequentially, in order of decreasing closeness of match.

- Client billing allocation — lawyer's office

  When a call comes in, the calling address is correlated with the corresponding client, and client's name, address, and the time of the call is logged. If no corresponding client is found, the call is forwarded to the lawyer's secretary.

4 Usage scenarios

A CPL would be useful for implementing services in a number of different scenarios.

- Script creation by end user

  In the most direct approach for creating a service with a CPL, an end user simply creates a script describing their service. He or she simply decides what service he or she wants, describes it using a CPL script, and then uploads it to a server.

- Third party outsourcing

  Because a CPL is a standardized language, it can also be used to allow third parties to create or customize services for clients. These scripts can then be run on servers owned by the end user or the user's service provider.

- Administrator service definition

  A CPL can also be used by server administrators to create simple services or describe policy for servers they control. If a server is implementing CPL services in any case, extending the service architecture to allow administrators as well as users to create scripts is a simple extension.

- Web middleware

  Finally, there have been a number of proposals for service creation or customization using web interfaces. A CPL could be used as the back-end to such environments: a web application could create a CPL script on behalf of a user, and the telephony server could then implement the services without either compotent having to be aware of the specifics of the other.

5 CPL creation

There are also a number of means by which CPL scripts could be created. Like HTML, which can be created in a number of different manners, we envision multiple creation styles for a CPL script.

- Hand authoring

  Most directly, CPL scripts can be created by hand, by knowledgable users. The CPL described in [5] has a text format with an uncomplicated syntax, so hand authoring will be straightforward.

- Automated scripts

  CPL features can be created by automated means, such as in the example of the web middleware described in the previous section. With a simple, text-based syntax, standard text-processing languages will be able to create and edit CPL scripts easily.

- GUI tools

  Finally, users will be able to use GUI tools to create and edit CPL scripts. We expect that most average-experience users will take this approach once the CPL gains popularity. The CPL will be designed with this application in mind, so that the full expressive power of scripts can be represented simply and straightforwardly in a graphical manner.

6 Network model

The Call Processing Language operates on a generalized model of an Internet telephony network. While the details of various protocols differ, on an abstract level all major Internet telephony architectures are sufficiently similar that their major features can be described commonly. This document generally uses SIP terminology, as its authors' experience has mainly been with that protocol.

6.1 Model components

In the Call Processing Language's network model, an Internet telephony network contains two types of components.

6.1.1 End systems

End systems are devices which originate and/or receive signalling information and media. These include simple and complex telephone devices, PC telephony clients, and automated voice systems. The CPL abstracts away the details of the capabilities of these devices. An end system can originate a call; and it can accept, reject, or forward incoming calls. The details of this process (ringing, multi-line telephones, and so forth) are not important for the CPL.

For the purposes of the CPL, gateways — for example, a device which connects calls between an IP telephony network and the PSTN — are also considered to be end systems. Other devices, such as mixers or firewalls, are not directly dealt with by the CPL, and they will not be discussed here.

6.1.2 Signalling servers

Signalling servers are devices which relay or control signalling information. In SIP, they are proxy servers, redirect servers, or registrars; in H.323, they are gatekeepers.

Signalling servers can perform three types of actions on call setup information. They can:

proxy it: forward it on to one or more other network or end systems, returning one of the responses received.

redirect it: return a response informing the sending system of a different address to which it should send the request.

reject it: inform the sending system that the setup request could not be completed.

RFC 2543 [1] has illustrations of proxy and redirect functionality. End systems may also be able to perform some of these actions: almost certainly rejection, and possibly redirection.

Signalling servers also normally maintain information about user location. Whether by means of registrations (SIP REGISTER or H.323 RAS messages), static configuration, or dynamic searches, signalling servers must have some means by which they can determine where a user is currently located, in order to make intelligent choices about their proxying or redirection behavior.

Signalling servers are also usually able to keep logs of transactions that pass through them, and to send e-mail to desinations on the Internet, under programatic control.

6.2 Component interactions

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 are two possibilities here: 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 signalling server or end system to which it can send the request directly.

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

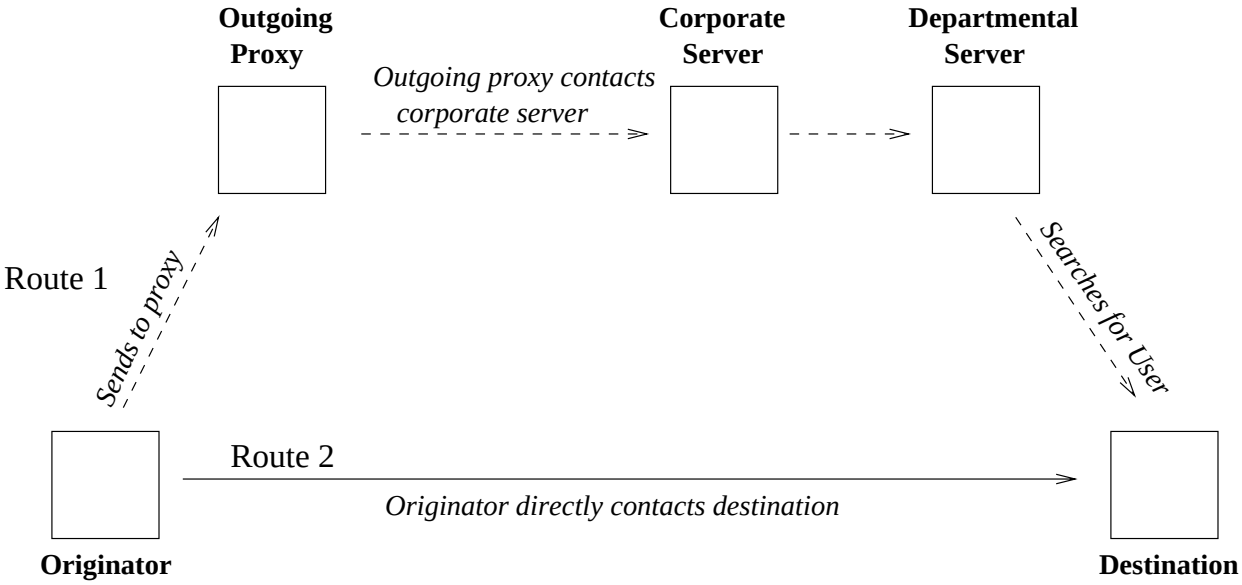

Figure 1: Possible paths of call setup messages

For example, in figure 1, there are two paths the call establishment request information may take. For Route 1, the originator knows only a user address for the user it is trying to contact, and it is configured to send outgoing calls through a local outgoing proxy server. Therefore, it forwards the request to its local server, which finds the server of record for that address, and forwards it on to that server.

In this case, the organization the destination user belongs to uses a multi-stage setup to find users. The corporate server identifies which department a user is part of, then forwards the request to the appropriate departmental server, which actually locates the user. (This is similar to the way e-mail forwarding is often configured.) The response to the request will travel back along the same path.

For Route 2, however, the originator knows the specific device address it is trying to contact, and it is not configured to use a local outgoing proxy. In this case, the originator can directly contact the destination without having to communicate with any network servers at all.

We see, then, that in Internet telephony signalling servers cannot in general know the state of end systems they "control," since signalling information may have bypassed them. This architectural limitation implies a number of restrictions on how some services can be implemented. For instance, a network system cannot

reliably know if an end system is currently busy or not; a call may have been placed to the end system without traversing that network system. Thus, signalling messages must explicitly travel to end systems to find out their state; in the example, the end system must explicitly return a "busy" indication.

7 Interaction of CPL with network model

7.1 What a script does

A CPL script runs in a signalling server, and controls that system's proxy, redirect, or rejection actions for the set-up of a particular call. It does not attempt to co-ordinate the behavior of multiple signalling servers, or to describe features on a "Global Functional Plane" as in the Intelligent Network architecture [6].

More specifically, a script replaces the user location functionality of a signalling server. As described in section 6.1.2, a signalling 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, and chooses the signalling 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 (or more properly set of actions) is taken. In some circumstances, additional actions can be taken based on the consequences of the first action and additional conditions. If no condition matches the invitation, the signalling server's standard action — its location database lookup, for example — is taken.

7.2 Which script is executed

CPL scripts are usually associated with a particular Internet telephony address. When a call establishment request arrives at a signalling 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 executed, if it has choosen to perform a proxy action, a new Internet telephony address will result as the destination of that proxying. Once this has occured, 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 as well is executed (assuming that a script has not attempted to proxy to an address which the server has already tried). For more details of this recursion process, and a description of what happens when a server has scripts that correspond both to a scripts origination address and its destination address, see section 9.2.

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:joe@example.com`, regardless of where that user actually is or what kind of device he or she is using. A device address, by contrast, refers to a particular physical device, such as `sip:x26063@phones.example.com`. Other, intermediate sorts of addresses are also possible, and have some use (such as an address for "my cell phone, wherever it currently happens to be registered"), but we expect them to be less common. A CPL script is agnostic to the type of address it is associated with; while scripts associated with user addresses are probably the most useful for most services, there is no reason that a script could not be associated with any other type of address as well. The recursion process described above allows scripts to be associated with several of a user's addresses; thus, a user script could specify an action "try me at my cell phone," whereas a device script could say "I don't want to accept cell phone calls while I'm out of my home area."

It is also possible for a CPL script to be associated not with one specific Internet telephony address, but rather with all addresses handled by a signalling server, or a large set of them. For instance, an administrator might configure a system to prevent calls from or to a list of banned incoming or outgoing addresses; these should presumably be configured for everyone, but users should still to be able to have their own custom scripts as well. Exactly when such scripts should be executed in the recursion process depends on the precise nature of the administrative script. See section 9.2 for further discussion of this.

7.3 Where a script runs

Users can have CPL scripts on any network server which their call establishment requests pass through and with which they have a trust relationship. For instance, in the example in figure 1, the originating user could have a script on the outgoing proxy, and the destination user could have scripts on both the corporate server and the departmental server. These scripts would typically perform different functions, related to the role of the server on which they reside; a script on the corporate-wide server could be used to customize which department the user wishes to be found at, for instance, whereas a script at the departmental server could be used for more fine-grained location customization. Some services, such as filtering out unwanted calls, could be located at either server. See section 9.3 for some implications of a scenario like this.

This model does not specify the means by which users locate a CPL-capable network server. In general, this will be through the same means by which they locate a local Internet telephony server to register themselves with; this may be thorugh manual configuration, or through automated means such as the Service Location Protocol [7]. It has been proposed that automated means of locating such servers should include a field to indicate whether the server allows users to upload CPLs.

8 Creation and transport of a call processing language script

Users create call processing language scripts, typically on end devices, and transmit them through the network to signalling servers. Scripts persist in signalling servers until changed or deleted, unless they are specifically given an expiration time; a network system which supports CPL scripting will need stable storage.

The end device on which the user creates the CPL script need not bear any relationship to the end devices to which calls are actually placed. For example, a CPL script might be created on a PC, whereas calls might be intended to be received on a simple audio-only telephone. Indeed, the device on which the script is created may not be an “end device” in the sense described in section 6.1.1 at all; for instance, a user could create and upload a CPL script from a non-multimedia-capable web terminal.

The CPL also might not necessarily be created on a device near either the end device or the signalling server in network terms. For example, a user might decide to forward his or her calls to a remote location only after arriving at that location.

The exact means by which the end device transmits the script to the server remains to be determined; it is likely that many solutions will be able to co-exist. This method will need to be authenticated in almost all cases. The methods that have been suggested include web file upload, SIP REGISTER message payloads, remote method invocation, SNMP, ACAP, LDAP, and remote file systems such as NFS.

Users can also retrieve their current script from the network to an end system so it can be edited. The signalling server should also be able to report errors related to the script to the user, both static errors that could be detected at upload time, and any run-time errors that occur.

If a user has trust relationships with multiple signalling servers (as discussed in section 7.3), the user may choose to upload scripts to any or all of those servers. These scripts can be entirely independent.

9 Feature interaction behavior

Feature interaction is the term used in telephony systems when two or more requested features produce ambiguous or conflicting behavior [8]. Feature interaction issues for features implemented with a call processing language can be roughly divided into three categories: feature-to-feature in one server, script-to-script in one server, and server-to-server.

9.1 Feature-to-feature interactions

Due to the explicit nature of event conditions discussed in the previous section, feature-to-feature interaction is not likely to be a problem in a call processing language environment. Whereas a subscriber to traditional telephone features might unthinkingly subscribe to both "call waiting" and "call forward on busy," a user creating a CPL script would only be able to trigger one action in response to the condition "a call arrives while the line is busy." Given a good user interface for creation, or a CPL server which can check for unreachable code in an uploaded script, contradictory condition/action pairs can be avoided.

9.2 Script-to-script interactions

Script-to-script interactions arise when a server invokes multiple scripts for a single call, as described in section 7.2. This can occur in a number of cases: if both the call originator and the destination have scripts specified on a single server; if a script forwards a request to another address which also has a script; or if an administrative script is specified as well as a user's individual script.

The solution to this interaction is to determine an ordering among the scripts to be executed. In this ordering, the "first" script is executed first; if this script allows or permits the call to be proxied, the script corresponding to the next address is executed. When the first script says to forward the request to some other address, those actions are considered as new requests which arrive at the second script. When the second script sends back a final response, that response arrives at the first script in the same manner as if a request arrived over the network. Note that in some cases, forwarding can be recursive; a CPL server must be careful to prevent forwarding loops.

Abstractly, this can be viewed as equivalent to having each script execute on a separate signalling server. Since the CPL architecture is designed to allow scripts to be executed on multiple signalling servers in the course of locating a user, we can conceptually transform script-to-script interactions into the server-to-server interactions described in the next section, reducing the number of types of interactions we need to concern ourselves with.

The question, then, is to determine the correct ordering of the scripts. For the case of a script forwarding to an address which also has a script, the ordering is obvious; the other two cases are somewhat more subtle. When both originator and destination scripts exist, the originator's script should be executed before the destination script; this allows the originator to perform address translation, call filtering, etc., before a destination address is determined and a corresponding script is chosen.

Even more complicated is the case of the ordering of administrative scripts. Many administrative scripts, such as ones that restrict source and destination addresses, need to be run after originator scripts, but before destination scripts, to avoid a user's script evading administrative restrictions through clever forwarding;

however, others, such as a global address book translation function, would need to be run earlier or later. Servers which allow administrative scripts to be run will need to allow the administrator to configure when in the script execution process a particular administrative script should fall.

9.3 Server-to-server interactions

The third case of feature interactions, server-to-server interactions, is the most complex of these three. The canonical example of this type of interaction is the combination of Originating Call Screening and Call Forwarding: a user (or administrator) may wish to prevent calls from being placed to a particular address, but the local script has no way of knowing if a call placed to some other, legitimate address will be proxied, by a remote server, to the banned address. This type of problem is unsolvable in an administratively heterogeneous network, even a "lightly" heterogeneous network such as current telephone systems. CPL does not claim to solve it, but the problem is not any worse for CPL scripts than for any other means of deploying services.

Another class of server-to-server interactions are best resolved by the underlying signalling protocol, since they can arise whether the signalling servers are being controlled by a call processing language or by some entirely different means. One example of this is forwarding loops, where user X may have calls forwarded to Y, who has calls forwarded back to X. SIP has a mechanism to detect such loops. A call processing language server thus does not need to define any special mechanisms to prevent such occurrences; it should, however, be possible to trigger a different set of call processing actions in the event that a loop is detected, and/or to report back an error to the owner of the script through some standardized run-time error reporting mechanism.

9.4 Signalling ambiguity

As an aside, [8] discusses a fourth type of feature interaction for traditional telephone networks, signalling ambiguity. This can arise when several features overload the same operation in the limited signal path from an end station to the network: for example, flashing the switch-hook can mean both "add a party to a three-way call" and "switch to call waiting." Because of the explicit nature of signalling in both the Internet telephony protocols discussed here, this issue does not arise.

10 Relationship with existing languages

This document's description of the CPL as a "language" is not intended to imply that a new language necessarily needs to be implemented from scratch. A server could potentially implement all the functionality described here as a library or set of extensions for an existing language; Java, or the various freely-available scripting languages (Tcl, Perl, Python, Guile), are obvious possibilities.

However, there are motivations for creating a new language. All the existing languages are, naturally, expressively complete; this has two inherent disadvantages. The first is that any function implemented in them can take an arbitrarily long time, use an arbitrarily large amount of memory, and may never terminate. For call processing, this sort of resource usage is probably not necessary, and as described in section 12.1, may in fact be undesirable. One model for this is the electronic mail filtering language Sieve [4], which deliberately restricts itself from being Turing-complete.

Similar levels of safety and protection (though not automatic generation and parsing) could also be achieved through the use of a "sandbox" such as is used by Java applets, where strict bounds are imposed on

the amount of memory, cpu time, stack space, etc., that a program can use. The difficulty with this approach is primarily in its lack of transparency and portability: unless the levels of these bounds are imposed by the standard, a bad idea so long as available resources are increasing exponentially with Moore's Law, a user can never be sure whether a particular program can sucessfully be executed on a given server without running into the server's resource limits, and a program which executes sucessfully on one server may fail unexpectedly on another. Non-expressively-complete languages, on the other hand, allow an implicit contract between the script writer and the server: so long as the script stays within the rules of the language, the server will guarantee that it will execute the script.

The second disadvantage with expressively complete languages is that they make automatic generation and parsing of scripts very difficult, as every parsing tool must be a full interpreter for the language. An analogy can be drawn from the document-creation world: while text markup languages like HTML or XML can be, and are, easily manipulated by smart editors, powerful document programming languages such as LaTeXor Postscript usually cannot be. While there are word processors that can save their documents in LaTeXform, they cannot accept as input arbitrary LaTeXdocuments, let alone preserve the structure of the original document in an edited form. By contrast, essentially any HTML editor can edit any HTML document from the web, and the high-quality ones preserve the structure of the original documents in the course of editing them.

11 Related work

11.1 IN service creation environments

The ITU's IN series describe, on an abstract level, service creation environments [6]. These describe services in a traditional circuit-switched telephone network as a series of decisions and actions arranged in a directed acyclic graph. Many vendors of IN services use modified and extended versions of this for their proprietary service creation environments.

11.2 SIP CGI

SIP CGI [9] is an interface for implementing services on SIP servers. Unlike a CPL, it is a very low-level interface, and would not be appropriate for services written by non-trusted users.

The paper "Programming Internet Telephony Services" [10] discusses the similarties and contrasts between SIP CGI and CPL in more detail.

12 Necessary language features

This section lists those properties of a call processing language which we believe to be necessary to have in order to implement the motivating examples, in line with the described architecture.

12.1 Language characteristics

These are some abstract attributes which any proposed call processing language should possess.

- Light-weight, efficient, easy to implement

In addition to the general reasons why this is desirable, a network server might conceivably handle very large call volumes, and we don't want CPL execution to be a major bottleneck. One way to achieve this might be to compile scripts before execution.

- Easily verifiable for correctness

  For a script which runs in a server, mis-configurations can result in a user becoming unreachable, making it difficult to indicate run-time errors to a user (though a second-channel error reporting mechanism such as e-mail could ameliorate this). Thus, it should be possible to verify, when the script is committed to the server, that it is at least syntactically correct, does not have any obvious loops or other failure modes, and does not use too many server resources.

- Executable in a safe manner

  No action the CPL script takes should be able to subvert anything about the server which the user shouldn't have access to, or affect the state of other users without permission. Additionally, since CPL scripts will typically run on a server on which users cannot normally run code, either the language or its execution environment must be designed so that scripts cannot use unlimited amounts of network resources, server CPU time, storage, or memory.

- Easily writeable and parseable by both humans and machines.

  For maximum flexibility, we want to allow humans to write their own scripts, or to use and customize script libraries provided by others. However, most users will want to have a more intuitive user-interface for the same functionality, and so will have a program which creates scripts for them. Both cases should be easy; in particular, it should be easy for script editors to read human-generated scripts, and vice-versa.

- Extensible

  It should be possible to add additional features to a language in a way that existing scripts continue to work, and existing servers can easily recognize features they don't understand and safely inform the user of this fact.

- Independent of underlying signalling details

  The same scripts should be usable whether the underlying protocol is SIP, H.323, a traditional telephone network, or any other means of setting up calls. It should also be agnostic to address formats. (We use SIP terminology in our descriptions of requirements, but this should map fairly easily to other systems.) It may also be useful to have the language extend to processing of other sorts of communication, such as e-mail or fax.

12.2 Base features — call signalling

To be useful, a call processing language obviously should be able to react to and initiate call signalling events.

- Should execute actions when a call request arrives

  See section 7, particularly 7.1.

- Should be able to make decisions based on event properties

  A number of properties of a call event are relevant for a script's decision process. These include, roughly in order of importance:

  - Destination address

    We want to be able to do destination-based routing or screening. Note that in SIP we want to be able to filter on either or both of the addresses in the To header and the Request-URI.

  - Originator address

    Similarly, we want to be able to do originator-based screening or routing.

  - Caller Preferences

    In SIP, a caller can express preferences about the type of device to be reached – see [11]. The script should be able to make decisions based on this information.

  - Information about caller or call

    SIP has textual fields such as Subject, Organization, Priority, etc., and a display name for addresses; users can also add non-standard additional headers. H.323 has a single Display field. The script should be able to make decisions based on these parameters.

  - Media description

    Call invitations specify the types of media that will flow, their bandwidth usage, their network destination addresses, etc. The script should be able to make decisions based on these media characteristics.

  - Authentication/encryption status

    Call invitations can be authenticated. Many properties of the authentication are relevant: the method of authentication/encryption, who performed the authentication, which specific fields were encrypted, etc. The script should be able to make decisions based on these security parameters.

- Should be able to take action based on a call invitation

  There are a number of actions we can take in response to an incoming call setup request. We can:

  - reject it

    We should be able to indicate that the call is not acceptable or not able to be completed. We should also be able to send more specific rejection codes (including, for SIP, the associated textual string, warning codes, or message payload).

  - redirect it

    We should be able to tell the call initiator sender to try a different location.

  - proxy it

    We should be able to send the call invitation on to another location, or to several other locations ("forking" the invitation), and await the responses. It should also be possible to specify a timeout value after which we give up on receiving any definitive responses.

- Should be able to take action based a response to a proxied or forked call invitation

  Once we have proxied an invitation, we need to be able to make decisions based on the responses we receive to that invitation (or the lack thereof). We should be able to:

- consider its message fields

  We should be able to consider the same fields of a response as we consider in the initial invitation.

- relay it on to the call originator

  If the response is satisfactory, it should be returned to the sender.

- for a fork, choose one of several responses to relay back

  If we forked an invitation, we obviously expect to receive several responses. There are several issues here — choosing among the responses, and how long to wait if we've received responses from some but not all destinations.

- initiate other actions

  If we didn't get a response, or any we liked, we should be able to try something else instead (e.g., call forward on busy).

12.3 Base features — non-signalling

A number of other features that a call processing language should have do not refer to call signalling per se; however, they are still extremely desirable to implement many useful features.

The servers which provide these features might reside in other Internet devices, or might be local to the server (or other possibilities). The language should be independent of the location of these servers, at least at a high level.

- Logging

  In addition to the CPL server's natural logging of events, the user will also want to be able to log arbitrary other items. The actual storage for this logging information might live either locally or remotely.

- Error reporting

  If an unexpected error occurs, the script should be able to report the error to the script's owner. This may use the same mechanism as the script server uses to report language errors to the user (see section 12.5).

- Access to user-location info

  Proxies will often collect information on users' current location, either through SIP REGISTER messages, the H.323 RRQ family of RAS messages, or some other mechanism (see section 6.2). The CPL should be able to refer to this information so a call can be forwarded to the registered locations or some subset of them.

- Database access

  Much information for CPL control might be stored in external databases, for example a wide-area address database, or authorization information, for a CPL under administrative control. The language could specify some specific database access protocols (such as SQL or LDAP), or could be more generic.

- Other external information

Other external information a script could access includes web pages, which could be sent back in a SIP message body; or a clean interface to remote procedure calls such as Corba, RMI, or DCOM, for instance to access an external billing database. However, for simplicity, these interfaces may not be in the initial version of the protocol.

12.4 Language features

Some features do not involve any operations external to the CPL's execution environment, but are still necessary to allow some standard services to be implemented. (This list is not exhaustive.)

- Pattern-matching

  It should be possible to give special treatment to addresses and other text strings based not only on the full string but also on more general or complex sub-patterns of them.

- Address filtering

  Once a set of addresses has been retrieved through one of the methods in section 12.3, the user needs to be able to choose a sub-set of them, based on their address components or other parameters.

- Randomization

  Some forms of call distribution are randomized as to where they actually end up.

- Date/time information

  Users may wish to condition some services (e.g., call forwarding, call distribution) on the current time of day, day of the week, etc.

12.5 Control

As described in section 8, we must have a mechanism to send and retrieve CPL scripts, and associated data, to and from a signalling server. This method should support reporting upload-time errors to users; we also need some mechanism to report errors to users at script execution time. Authentication is vital, and encryption is very useful. The specification of this mechanism can be (and probably ought to be) a separate specification from that of the call processing language itself.

13 Security considerations

The security considerations of transferring CPL scripts are discussed in sections 8 and 12.5. Some considerations about the execution of the language are discussed in section 12.1.

14 Acknowledgments

We would like to thank Tom La Porta and Jonathan Rosenberg for their comments and suggestions.

15 Authors' Addresses

Jonathan Lennox
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue, MC 0401
New York, NY 10027
USA
electronic mail: lennox@cs.columbia.edu

Henning Schulzrinne
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue, MC 0401
New York, NY 10027
USA
electronic mail: schulzrinne@cs.columbia.edu

References

[1] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: session initiation protocol," Request for Comments (Proposed Standard) 2543, Internet Engineering Task Force, Mar. 1999.

[2] International Telecommunication Union, "Packet based multimedia communication systems," Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, Feb. 1998.

[3] K. Coar and D. Robinson, "The WWW common gateway interface version 1.1," Internet Draft, Internet Engineering Task Force, Apr. 1999. Work in progress.

[4] T. Showalter, "Sieve: A mail filtering language," Internet Draft, Internet Engineering Task Force, Mar. 1999. Work in progress.

[5] J. Lennox and H. Schulzrinne, "CPL: a language for user control of internet telephony services," Internet Draft, Internet Engineering Task Force, Mar. 1999. Work in progress.

[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, Mar. 1993.

[7] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service location protocol, version 2," Request for Comments (Proposed Standard) 2608, Internet Engineering Task Force, June 1999.

[8] E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schure, and H. Velthuijsen, "A feature interaction benchmark for IN and beyond," *Feature Interactions in Telecommunications Systems, IOS Press*, pp. 1–23, 1994.

[9] J. Lennox, J. Rosenberg, and H. Schulzrinne, “Common gateway interface for SIP,” Internet Draft, Internet Engineering Task Force, May 1999. Work in progress.

[10] J. Rosenberg, J. Lennox, and H. Schulzrinne, “Programming internet telephony services,” Technical Report CUCS-010-99, Columbia University, New York, New York, Mar. 1999.

[11] H. Schulzrinne and J. Rosenberg, “SIP caller preferences and callee capabilities,” Internet Draft, Internet Engineering Task Force, Mar. 1999. Work in progress.

Full Copyright Statement