Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-iptel-cpl-requirements-00.ps

IPTEL WG
Lennox/Schulzrinne
Lucent Bell Labs/Columbia University
July 30, 1998
Expires: February 1999

Call Processing Language 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 architecture for such a method, 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 of these services may reside on end devices. A broad set of services, however — those involving user location, call distribution, behavior-on-busy, and the like — are independent of a particular end device, or need to be operational even when an end device is unavailable. These still best reside in a network device rather than an end system. To allow user control over such devices, we need a standardized way for end-users to specify the precise behavior of the servers. This document proposes an architecture in which network devices or end systems respond to call signalling events by triggering user-created programs which control the reaction to the events.

For reasons discussed in section 3.7, this document proposes a relatively static, non-expressively-complete language to solve this problem. We call this a *call processing language.* However, most of the requirements this document lists apply equally well to a library of call processing routines for an existing language.

2 Motivating examples

These are some specific examples of services which we want to be able to create programatically. They are arranged roughly in order of increasing requirements they impose. 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 superviser, in which case it should be proxied to the user's cellphone if it has registered.

- Administrative screening — firewall

  An outgoing call should be rejected if it is going to any destination that is on a "banned" list. Otherwise, it should be forwarded on to the appropriate destination; if the destination accepts the call, the firewall should be told to open up the UDP host/port pairs the two endpoints specified for their media. The same thing should be done for incoming calls, checking the origination address.

- Central phone server

  If a call comes in for a specific person, it should be redirected to the locations where they can currently be found. If a call comes in for the general "information" address we've advertised, 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 gets a recorded message indicating what times they can call.

- Intelligent user location

  When a call comes in, it should ring at every station from which the user has registered. 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.

- Intelligent user location with mixer (home phone)

  When a call comes in, it should ring at every station from which the user has registered. If the call is picked up from more than one station, the media from each station should be transparently mixed together and sent to the caller.

- Third-party registration control

  When a registration arrives for a user, make sure that the registration was authenticated, the person performing the registration has permission to perform the registration for the specified user, and the location registered is allowed for the registered user. If so, enter it in the registration database; if not, reject it.

- Calendarbook access

  When a call comes in, the user's on-line calendar should be consulted. If it specifies that the user has a meeting scheduled for this time, the caller should get a busy indication. Otherwise, the call should be directed to the user's office telephone.

- Client billing allocation — lawyer's office

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

- End system busy

  When a new call comes in, if the user is currently in a call, a call-waiting tone is generated, unless one of the calls currently in progress is with the user's boss or he or she has set "Do Not Disturb" in the user interface, in which case the caller gets a busy indication.

- Phone bank (call distribution/queueing)

  Incoming calls should be distributed to the phone-bank workers, so that each worker handles approximately the same total number of calls. If all the phone-bank workers are busy, calls should be queued until someone is available. Calls coming from preferred customers should get priority in the queue. If the length of the queue grows to twice the size of the phone bank, calls should be directed to management as well until the queue length has decreased again. Each caller should be given an approximate indication of waiting time or number of calls ahead of them in the queue as they wait. Callers should be given the option of listening to the music-on-hold of their choice.

3 Architecture

3.1 Network components

A network which supports Internet telephony consists of two types of components: end systems, which originate and/or receive media, and network systems, which relay signalling information.

End systems are either user agents, which reach actual people, or automated systems, which do not; this document will deal primarily with the former. Network systems, in SIP, are proxy servers, redirect servers, or registrars; in H.323 they are gatekeepers. The functionality between the two protocols is largely equivalent, and this document will generally use the SIP names. Proxy servers are network systems which receive a request and forward it on. Redirect servers tell the originating location an alternate location to try. Registrars track users' current locations; they will usually be the same devices as proxy or redirect servers, but do not necessarily have to be. See the illustrations in [1]. End systems may also have some properties of network systems, most likely the ability to perform redirection.

3.2 Model of normal use

Internet telephony addresses can be divided into two broad categories: terminal addresses and permanent user addresses. A terminal address is one that refers to a particular device, whose network-level (IP) address does not change. A permanent user address, on the other hand, refers on a more abstract level to an individual user, whose current location and network address may change. When a user becomes available at a location, his or her end system registers itself with a network server indicating this fact. (In SIP, users register via the `REGISTER` message; in H.323, via `RRQ` and related RAS messages.) A user may register from more than one location simultaneously.

Figure 1 shows how call signalling may flow. Calls may be placed either to terminal or permanent user addresses. Calls to terminal addresses may contact the corresponding device directly, or may travel through some signalling server. Calls to permanent user addresses must pass through the signalling server, which locates the user and proxies or redirects the call to the appropriate terminal addresses which the user has registered.

Signalling servers may also be used on the originating side. Rather than locate a call's destination on its own, an end system, when originating a call, may have been configured to transfer all its call requests

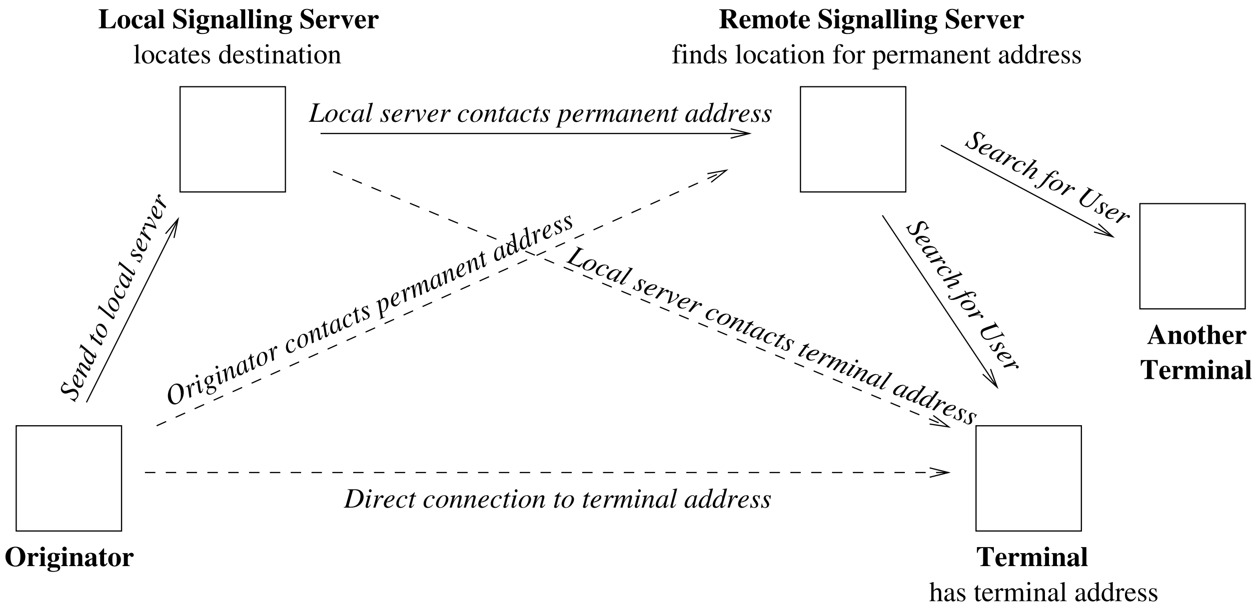

Figure 1: Illustration of call signalling messages

through a single, presumably local, server. This server can then perform the somewhat complex task of actually locating the destination, as well as other tasks such as firewall penetration or encryption of signalling information.

Call requests may be forwarded between multiple signalling servers on both the origination and destination ends of a call. For example, a corporation could have a large company-wide server which forwards incoming call requests to individual departmental servers, which then perform the task of actually locating the desired user. This is similar to a typical configuration of e-mail forwarding.

Different call invitations for a particular end system might travel through a different set of signalling servers; for instance, a user with several addresses might register his current end system with several different servers. Similarly, an end system placing a call might have several different outgoing signalling servers through which it could place the call. Thus, in general, a signalling server does not see all the signalling events for a particular end system; and so it does not have enough information to be able to determine the end system's state.

3.3 Purpose of a call processing language

A call processing language (CPL) is primarily intended to allow the user to modify the way an Internet telephony system handles call events. Call events include signalling events such as call setup, termination, or parameter changes, and also, for servers with an appropriate media path, in-band events such as DTMF tones. The user can modify either incoming or outgoing calls.

The most common sort of modification will be for incoming call setups. Some ways a user might want to alter the call setup process include: to search terminal addresses for a given user address in an alternate way; to specificy what happens when the initial search fails, either when it receives some sort of negative response (e.g., busy), or does not receive any definitive response within a fixed time period (e.g., no answer); or to handle certain origination addresses specially, for instance by informing the caller that the call was refused.

The useful changes to the outgoing call setup are somewhat more limited in scope, but one example is to translate a user's abbreviated addresses into an address specified with a fully-qualified domain name. The transformations to parameter changes or call terminations are generally only useful to complete the actions begun at call setup time; see for instance the lawyer's office example in section 2.

Once a language with this level of power has been introduced, other applications of it present themselves. An administrator might wish to perform administrative restrictions on users' calls, for instance blocking incoming or outgoing calls from certain domains. The language could also be scripted on an end system; with minimal extensions, behavior specific to end systems, such as the specifics of how the user is alerted to incoming calls, could also be made programmable.

3.4 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 network systems. Scripts persist in network devices until changed or deleted, unless they are specifically given an expiration time; a network device which supports CPL scripting will need stable storage.

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 access, SIP REGISTER message payloads, remote method invocation, SNMP, LDAP, and remote file systems such as NFS.

Creation of a CPL script may be through the creation of a text file; or for a simpler user experience, a graphical user interface which allows the manipulation of some basic rules.

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

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's calls will pass through multiple local signalling servers which know about that user (as discussed in section 3.2), the user may choose to upload scripts to any or all of those servers. These scripts can be entirely independent; see section 3.6 for some implications of this.

If, as discussed in section 3.3, the call processing language is extended to control end systems, the script-creation mechanism described above should also be able to create such end-system scripts. It may be possible that the end system on which the script executes (the simple telephone mentioned before) is not the same device as the end system on which the script is created; in this case, the script should be transmitted from the script creation site to the end system in the same way it is transmitted from creation sites to network systems.

3.5 Execution process of a CPL script

When a call event arrives, a CPL server considers the information in the request and determines if any of the scripts it has stored are applicable to the call in question. If so, it performs the actions corresponding to the matching scripts.

The most common type of script defines a set of actions to be taken for the entire process of call set-up — from the time a call request is initially received, to the time that (from the point of view of this device) the call is either definitively accepted or definitively rejected. This could be near-instantaneous, if, for instance, the script decides to reject the call; or it could be an arbitrarily long time, if we are waiting for a call pick-up without a timeout.

Generally, we expect a script to be structured as a list of condition/action pairs; if an incoming invitation matches a given condition, then the corresponding action (or more properly set of actions) will be taken. Whether this should be explicit in the language or just implicit in the normal usage remains to be seen. If no condition matches the invitation, the signalling server's standard action should be taken.

Other types of scripts may define sets of actions to be taken for other call events: call termination; changes to media format or other call parameters (re-invitations, in SIP); or in-band call events, such as a user sending DTMF tones. However, it is important to note that many, if not most, network servers cannot expect to be able to observe such events; subsequent signalling information may short-cut past the server, as media information almost certainly will.

While many of the uses of a CPL script are specific to one particular user, there are a number of circumstances in which an administrator of a signalling server would wish to provide a script which applies to all users of the server, or a large set of them. For instance, a system might be configured to prevent calls from or to a list of banned incoming or outgoing addresses; these should presumably be configured for everyone, but users still need to be able to have their own custom scripts as well. Similarly, an administrative script might perform the necessary operations to allow media to traverse a firewall; but individual users' scripts should not have permission to perform these operations. See the next section for some implications of this.

3.6 Feature interaction behavior

Feature interaction is the term used in telephony systems when two or more requested features produce ambiguous or conflicting behavior [3]. 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.

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.

Script-to-script interactions can arise if both an originator and a destination have scripts specified on the same signalling server, or if an administrative script and a user's script are both specified. In the former case, the correct behavior is fairly obvious: a server should first execute the originator's script, and then, if that script placed a call to a destination, call the destination script with the appropriate conditions.

The correct behavior in the latter case depends on the scope of the administrative script; however, normally, the administrator's script should run after origination scripts, intercepting any proxy or redirection decisions, and before recipient scripts, to avoid a user's script evading administrative restrictions.

The third case — server-to-server interactions — is the most complex of these three. Many such problems are unsolvable in an administratively heterogeneous network, even a "lightly" heterogeneous network such as current telephone systems. The canonical example of this is the interaction 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.

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.

As an aside, [3] 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.

3.7 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 5.1, 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. The second disadvantage with expressively complete languages is that they make automatic generation and parsing very difficult; an analogy can be drawn with the difference between markup languages like HTML or XML, which can easily be manipulated by smart editors, and powerful document programming languages such as Latex or Postscript which usually cannot be.

4 Related work

A future revision of this document will discuss such items as decision tree languages, AT&T's TOPS language, the IN service creation language, timed state diagrams, the Java servlet API, cgi-bin, and active networks.

5 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.

5.1 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.

5.2 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 an action script when a call request arrives

  See section 3, particularly 3.5.

- 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:

  - Event type

    It should be possible to handle call invitations, call terminations, user registrations, OPTIONS requests, and other distinct call events separately.

  - Originator address

    We want to be able to do originator-based screening or routing.

  - Destination address

    Similarly, we want to be able to do destination-based screening or routing. Note that in SIP we want to be able to filter on any or all of the addresses in the To header, the Location header, and the Request-URI.

  - 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.

  - Media description

    Requests specify the types of media that will flow, their bandwidth usage, their network destination addresses, etc.

  - Authentication/encryption status

    Requests 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.

- Should be able to take action based on a request

  There are a number of actions we can take in response to an incoming 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).

  - send a provisional response to it

    While a call request is being processed, provisional responses such as "Trying," "Ringing," and "Queued" are sent back to the caller. It is not clear whether the script should specify the sending of such responses explicitly, or whether they should be implicit in other actions performed.

  - redirect it

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

  - proxy it

    We should be able to send the request on to another location, or to several other locations, 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 request

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

  – consider all its message fields

    This consists of a similar set of fields as appear in a request.

  – relay it on to the requestor

    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 a request, 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).

5.3 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 should use the same mechanism as the script server uses to report language errors to the user (see section 5.9).

- 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 3.2). 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 the script should be able to 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.

- Creation of and access to local state

  A CPL script may wish to store state information, so that scripts invoked for future transactions related to this call or this user can have access to decisions made by an earlier invocation. For instance, a SIP re-invitation should be proxied to the same location as accepted the original invitation, regardless of the usual forwarding sequence; a server may wish to log the termination of a call in the same way it logged its initiation; or a user might want to limit the number of concurrent calls or calls per day allowed. The persistence of this state information for a call should be time-limited, either explicitly or by default. See section 3.5 for some caveats for a network system of expecting to receive events other than call initiation.

5.4 Higher-level features

There are some, more complex services which it would be quite useful to be able to describe with a CPL, but which require considerably more maintenance of state, elaborate inter-call event triggering, and so forth, than the features described earlier.

It is not clear whether these features should be specified as primitives of the language, or whether they should be assembled from lower-level features. In the latter case, the language may need to have additional low-level features added, beyond those specified in the previous sections, so these features can be constructed.

- Queueing

  Calls which should go to end systems which aren't accepting calls currently can be queued to await delivery. This requires inter-call synchronization to know when to take calls off the queue.

  It should be possible for the CPL to specify a priority for a queue entry.

- Call distribution

  Calls can be spread to a number of end systems, in a "phone bank" style set-up. Calls need to be directed to exactly one, currently available destination, in some fair manner (e.g., hierarchical, round-robin, randomly distributed, or weighted fair queueing). In many cases, this means that the proxy system needs to be able to track the state of end systems.

  This should also be able to interface with queueing — if all end systems are busy, the call is queued, and when one becomes free, the call is taken off the queue.

5.5 Contingent features

Some features are only useful if other network entities are available.

- Access to media servers

  We want to be able to connect a remote call to recorded audio (or video) messages.

- Firewall control

  If we are working in an environment with a firewall, we need to be able to tell it to open up a specific host/port 5-tuple for our media to flow through.

  We should be able to specify authentication so the firewall knows it can trust the proxy.

- Mixer/translator control

  If we have a media mixer or translator available, we want to be able to tell it to mix media between several addresses, with fine-grained control over what media flows to where, in what formats.

5.6 End-system specific features

Some features can only be implemented in end-systems, either because some end-system state is not generally communicated over the network, or because there is no protocol to signal that actions need to be performed. If we want these features to be implementable with the CPL, these additional operations will be necessary.

- Access to current calling state

  We want to know how many other calls are in progress, who they are with, etc.

- Access to additional user interface state

  The end system's user interface might present options which the user could specify on a per-call basis (for example, setting "do not disturb").

- Control of user notification UI

  This will allow such features as custom distinctive ringing, call-waiting tones (in combination with the call-state query), "reminder ring" for call forwarding, etc.

5.7 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.

- 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.

5.8 Future features

A number of services which don't exist yet or aren't widely deployed in the Internet will be relevant for Internet telephony. Once they are available, a CPL script should be able to control them.

- Ability to specify quality-of-service

  Certain calls — either on the basis of importance, or for known-troublesome destinations — should be able to have their desired quality of service specified by the language.

- Access to wide-area service location information

  A proxy doing call distribution might want to locate the service "closest" (in any one of a number of senses) to the caller; or we might want to find a PSTN gateway close to the destination of a PSTN-style call. In either case a script should be able to control these operations.

- Control of payment authorization

  If any kind of per-call billing is required, a CPL might want to be able to decide whether to accept charges. This is obviously a rather delicate operation from a security standpoint.

5.9 Control

As described in section 3.4, 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.

6 Security considerations

The security considerations of transferring CPL scripts are discussed in sections 3.4 and 5.9. Some considerations about the execution of the language are discussed in section 5.1.

7 Acknowledgments

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

8 Authors' Addresses

Jonathan Lennox
Lucent Technologies, Bell Laboratories
Rm. 4F-520
101 Crawfords Corner Road
Holmdel, NJ 07733
USA
electronic mail: lennox@dnrc.bell-labs.com

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

References

[1] M. Handley, H. Schulzrinne, and E. Schooler, "SIP: session initiation protocol," Internet Draft, Internet Engineering Task Force, May 1998. Work in progress.

[2] International Telecommunication Union, "Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service," Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, May 1996.

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

[4] T. Showalter, "Sieve – a mail filtering language," Internet Draft, Internet Engineering Task Force, Jan. 1998. Work in progress.

Full Copyright Statement