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

IPTEL WG
Lennox/Schulzrinne
Columbia University

June 25, 1999
Expires: December 1999

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. Internet telephony, however, provides an opportunity to open up the service creation process to end users or third-party service designers. To accomplish this however, 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*.

2 Motivating examples

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.

3 Architecture

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.

3.1 Network components

In the view of the Call Processing Language, an Internet telephony network consists of two types of components. End systems originate and/or receive signalling information and media; network systems relay or control signalling information. While in actual networks other devices exist, such as mixers, media gateways, or firewalls, the CPL does not deal with them directly, and they will not be discussed here.

Network systems, in SIP, are proxy servers, redirect servers, or registrars; in H.323 they are gatekeepers. On the abstract level the CPL deals with, the functionality of two protocols is largely equivalent, and this document will generally use SIP terminology. Network systems 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.

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

3.2 Network model

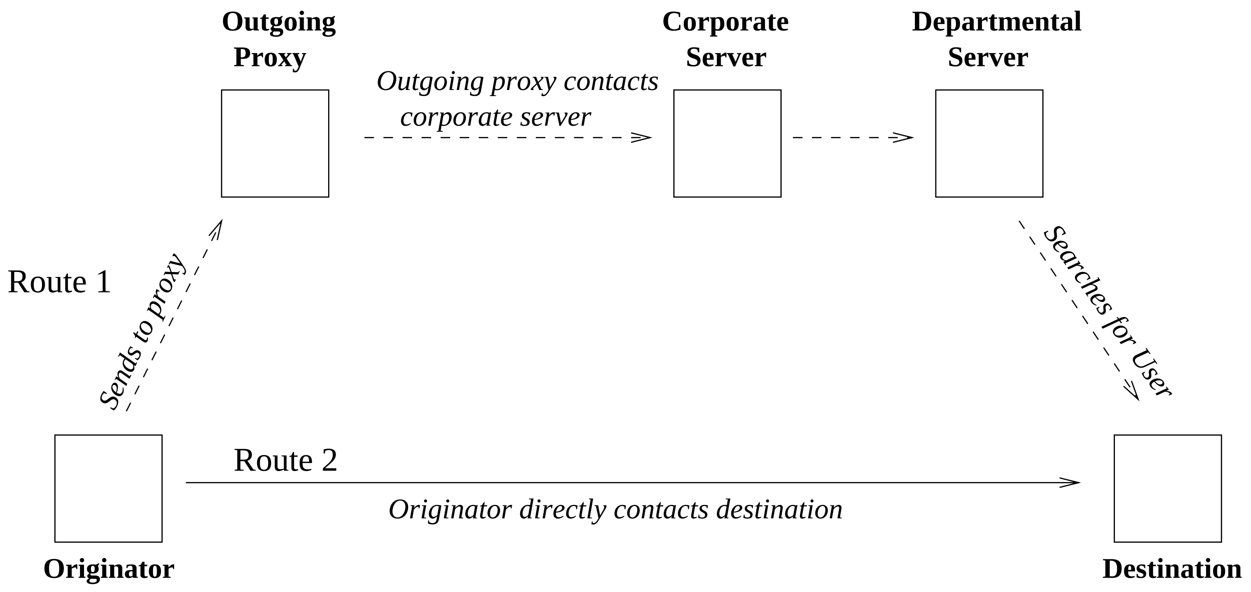

Figure 1: Possible paths of call setup messages

An Internet telephony network contains a number of network systems and a number of user agents. Call establishment requests can pass through a series of network systems, and user agents can be contacted by any of a number of network systems, or directly by other user agents.

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 network systems 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.

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 above 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 3.6.3 for some implications of a scenario like this.

3.3 Role of a CPL script

A CPL script runs in a network system, 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 network systems, or to describe features on a "Global Functional Plane" as in the Intelligent Network architecture.

CPL scripts are associated with a particular Internet telephony address. When a call establishment request arrives at a network system which is a CPL server, that server associates the address specified in the request with its database of CPL scripts; if one matches, that corresponding script is executed. CPL scripts may be associated either with the originator address or the destination address of the call establishment request. For some discussion of what happens if, for instance, a server has scripts for both an originating and destination address, see section 3.6.2.

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.

By controlling basic call set-up actions, a user can achieve a number of services. Many common services are implemented using a CPL script for incoming calls to a user address. These include: searching for the user's current location in some specialized way; specifying what happens when this initial search fails, either because it received some sort of negative response (e.g., busy) or did not receive any definitive response within a fixed time period (e.g., no answer); or handling certain originating addresses specifically, for instance by informing the caller that the call was refused. Services that can be implemented by a script triggered by an outgoing user address are somewhat more limited, but one example is to translate a user's abbreviated addresses into addresses specified with a fully-qualified domain name.

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 systems until changed or deleted, unless they are specifically given an expiration time; a network system 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 file upload, SIP REGISTER message payloads, remote method invocation, SNMP, ACAP, 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 can 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.

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 the server allows calls to wait for a call pick-up without imposing a timeout.

Abstractly, a script can be considered 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. In some circumstances, additional actions can be taken based on the consequences of the first action, and possibly on additional conditions. If no condition matches the invitation, the signalling server's standard action should be taken.

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.

3.6.1 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.

3.6.2 Script-to-script interactions

Script-to-script interactions can arise if multiple scripts are invoked for a single call. This can occur in a number of possible 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.

In the first two of these cases, the correct behavior is fairly obvious: the server should first execute the actions specified by the "first" script. In the first case, this is the originator's script; in the second case, this is the script which triggered the request. 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 script arrived over the network. Note that for the second type of these interactions, script forwarding can be recursive; a CPL server much be careful to prevent forwarding loops.

The correct behavior for the third type of script-to-script interaction depends on the scope of the administrative script. Typically, 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.

3.6.3 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.

3.6.4 Signalling ambiguity

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

4.1 IN service creation environments

The ITU’s IN series describe, on an abstract level, service creation environments [5]. 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.

4.2 SIP CGI

SIP CGI [6] 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.

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 actions 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:

  – 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 [7]. 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.

  – 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 (“branching” the request), 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 branched 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 its message fields

    We should be able to consider the same fields of a response as we consider in the initial request.
  - relay it on to the requestor

    If the response is satisfactory, it should be returned to the sender.
  - for a branch, choose one of several responses to relay back

    If we branched 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 5.5).

- 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 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.

5.4 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 5.3, 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.

5.5 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 5.5. Some considerations about the execution of the language are discussed in section 5.1.

7 Changes from previous versions

7.1 Changes from `draft-ietf-iptel-cpl-requirements-00`

The changebars in the Postscript version of this document indicate changes from this version.

- Changed the title of the draft from "...Requirements" to "...Framework and Requirements," and changed the draft name, to better reflect the content.
- Deleted a number of overambitious service examples that aren't supported in the CPL as it has developed.
- Deleted discussion of end systems, media devices, and other items that aren't supported in the CPL as it has developed.
- Reorganized the Architecture section.
- Clarified the Network Model section.
- Added Related Work section.
- Added requirement to support caller preferences.
- Deleted many requirements for higher-level and end-system features that are not supported in the CPL as it has developed.
- Re-worded many sections for clarity.
- Added To Do / Open Issues section.

8 To Do / Open Issues

- Add Terminology section.
- How do users find out which servers they should upload their scripts to?
- Flesh out the Related Work sections, particularly describing the different roles of CPL and SIP CGI. (As in [8].)
- The Control section needs to be fleshed out considerably.
- The entire document should be reorganized for clarity.

9 Acknowledgments

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

10 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, "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, 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.

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

[5] 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.

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

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

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

Full Copyright Statement