Tech-Report Number CUCS-010-99

Programming Internet Telephony Services

Jonathan Rosenberg	Jonathan Lennox	Henning Schulzrinne
Bell Laboratories	Columbia University	Columbia University
`jdrosen@bell-labs.com`	`lennox@cs.columbia.edu`	`hgs@cs.columbia.edu`

Abstract

Internet telephony enables a wealth of new service possibilities. Traditional telephony services, such as call forwarding, transfer, and 800 number services, can be enhanced by interaction with email, web, and directory services. Additional media types, like video and interactive chat, can be added as well. One of the challenges in providing these services is how to effectively program them. Programming these services requires decisions regarding where the code executes, how it interfaces with the protocols that deliver the services, and what level of control the code has. In this paper, we consider this problem in detail. We develop requirements for programming Internet telephony services, and we show that at least two solutions are required — one geared for service creation by trusted users (such as administrators), and one geared for service creation by untrusted users (such as consumers). We review existing techniques for service programmability in the Internet and in the telephone network, and extract the best components of both. The result is a Common Gateway Interface (CGI) that allows trusted users to develop services, and the Call Processing Language (CPL) that allows untrusted users to develop services.

1 Introduction

Internet telephony enables a wealth of new service possibilities [1]. Traditional telephony services, such as call forwarding, transfer, and 800 number services, can be enhanced by adding integration with email, web, presence, instant messaging and directory services. Users can have calls redirected to web pages, use streaming media tools to record voicemail, or use instant messages in place of call waiting notifications. IP telephony can offer improved speech quality through advanced speech and audio codecs. Communications can encompass not just voice, but video, shared applications, and even virtual reality. Much more powerful user interfaces can allow these services to be made easily accessible. Gateways to the Public Switched Telephone Network (PSTN) can allow these services to extend to traditional landline phones, cellular phones, and pagers.

With such a wide range of services possible, it becomes critical to provide means by which these services can be rapidly conceived, developed, and deployed. It should not be necessary to add new network elements for each new service, nor should it be necessary to reinvent the interfaces to existing elements for each new service. In addition, it should be possible for third parties to create new services easily. By third parties, we mean individuals or organizations besides the ones that own or build the routers, hubs, switches, and servers which actually implement the service. This separation allows end users flexibility: they can purchase certain services from one provider, and other services from other providers. It also opens new markets for service providers.

This notion of service programmability has been a goal for the telephone network for some time, although it never fully materialized. However, the Internet provides a new opportunity to realize this goal. Its end-to-end architecture, which allows any host to send any data to any other host, is the ideal platform on which this service model can be achieved. .

In this paper, we consider this problem in more detail. Since the services are ultimately realized through Internet telephony signaling protocols, we provide a quick review of them in Section 2. We then discuss, in section 3 the design decisions that must be made in the development of a programming mechanism, and argue that two solutions are required. To develop solutions, we consider how the problem has been resolved in the telephone network (using the Intelligent Network) and in the World Wide Web (WWW), in sections 4.1 and 4.2 respectively. We then propose our solution for service creation by trusted users — CGI — in section 5 and for service creation for untrusted users — CPL — in section 6. We then conclude in section 7.

2 Signaling Protocol

There is a strong connection between programming IP telephony services and the protocols used to deliver these services. A number of different protocols are needed for delivery of IP telephony services. These include transport protocols, such as the Real Time Transport Protocol (RTP) [2], used to carry voice on IP networks, QoS protocols, such as the Resource Reservation Protocol (RSVP) [3] and Differentiated Services [4], used to provide low delay and loss for voice transport, and directory access protocols, used for user location and policy. Most importantly, it includes signaling protocols. Signaling protocols are used to setup and teardown the call, carry information required for the call to progress (such as media codecs and addresses), locate the user to be called, negotiate capabilities, and invoke services like hold, mute, and transfer. Since these protocols are what ultimately provide services, understanding them is key to programming services.

There are three main protocols currently used for signaling IP telephony services. These are H.323 [5], the Media Gateway Control Protocol (MGCP) [6], and the Session Initiation Protocol (SIP) [7]. H.323 was developed in the International Telecommunications Union (ITU). It was originally conceived for multimedia conferencing on a LAN, but has since been extended to cover Internet telephony. It provides call control, conferencing functions, call management, capabilities negotiation, and supplementary services. MGCP is a control protocol, allowing a central coordinator to monitor events in IP phones, and instruct them to send media to specific addresses. SIP was developed in the Internet Engineering Task Force (IETF), and is a protocol engineered for lightweight, distributed call control and capabilities negotiation. A detailed comparison of SIP and H.323 can be found in [8].

We chose to use SIP as a platform for programming telephony services. Its clean request-response model is amenable to simple programming. Its textual formatting and simple header structure makes it easy to use text processing languages, such as Perl, and textual interfaces, such as CGI (see Section 4.2), for developing services. The control it provides over the amount of state maintained at a server is very useful for service programming. Finally, its ability to work in a fully distributed fashion, avoiding routing loops and maintaining consistent behavior across servers, helps in avoiding feature interactions when programming services. A more detailed description of SIP can be found in [9, 10, 11]. We briefly overview it here, touching on the features relevant for our discussions.

2.1 Session Initiation Protocol

SIP is a client-server protocol, similar in both syntax and semantics to the HyperText Transfer Protocol (HTTP) [12]. However, it defines its own methods and headers for providing the functions required in IP telephony signaling. Requests are generated by one entity (the client), and sent to a receiving entity (the server) that processes them, and then sends responses. A request and the responses which follow it are called a *transaction*. As a call participant may either generate or receive requests, SIP-enabled end systems include a protocol client and server (generally called a user agent client and user agent server, respectively). The user agent server responds to the requests based on human interaction or some other kind of input. Furthermore, SIP requests can traverse many *proxy servers*, each of which receives a request and forwards it towards a next hop server, which may be another proxy server or the final user agent server. As a result, proxy servers are primarily responsible for *call routing*. A server can make use of any means at its disposal to determine where to route a call, including database queries or local program execution.

A server may also act as a *redirect server*, informing the client of the address of the next hop server, so that the client can contact it directly. There is no protocol distinction between a proxy server, redirect server, and user agent server; a client or proxy server has no way of knowing which it is communicating with. The distinction lies only in function. A proxy or redirect server cannot accept or reject a request, whereas a user agent server can. This is similar to the HTTP model of clients, origin and proxy servers. A single host may well act as client and server for the same request.

As in HTTP, the client requests invoke *methods* on the server. Requests and responses are textual, and contain header fields which convey call properties and service information. SIP reuses many of the header fields used in HTTP, such as the entity headers (e.g., Content-type) and authentication headers.

SIP defines several methods: INVITE invites a user to a call, BYE terminates a connection between two users in a call, OPTIONS solicits information about capabilities, but does not set up a connection, ACK is used for reliable message exchanges for invitations, CANCEL terminates a search for a user, and REGISTER conveys information about a user's location to a SIP server.

```
INVITE sip:joe@example.com SIP/2.0
Via: SIP/2.0/UDP 10.2.33.88:5067
From: Bob Jones <sip:bob@university.edu>
Subject: Internet telephony standards
To: Joe Smith <sip:joe@example.com>
Call-ID: 1997234505.56.78@10.2.33.88
Content-type: application/sdp
CSeq: 4711 INVITE
Content-Length: 187

v=0
o=user1 53655765 2353687637 IN IP4 128.3.4.5
s=Mbone Audio
i=Discussion of Mbone Engineering Issues
e=mbone@somewhere.com
c=IN IP4 224.2.0.1/127
t=0 0
m=audio 3456 RTP/AVP 0
```

Figure 1: Example SIP Message with SDP

A call is set up by issuing an INVITE request. This request contains header fields used to convey information about the call. Examples of some of the header fields are To, which lists the callee, From which lists the caller, Subject, which identifies the subject of the call, Call-ID which contains a unique call identifier, Contact which lists addresses where a user can be contacted, and Require, which allows for feature negotiation. The call control extensions [13] defines an additional header, called Also, which allows for simple third-party call control. This can enable services like transfer and multi-party conferencing.

At any time after the call is set up, either party may send a new INVITE request to the other to change parameters of the call. This includes changing media codecs, adding parties to the call, or changing addresses (for mobility support). Once the call is over, either side may hang up by sending a BYE request to the other.

Like HTTP, SIP requests and responses carry bodies, which can be any defined MIME [14] type. The body conveys information about the session between the parties. Normally, the Session Description Protocol (SDP) [15] is used. SDP describes multimedia sessions, including codec types, port numbers, addresses, and session start and stop times.

An example SIP message with an SDP payload is shown in Figure 1. The SIP message is an INVITE request, from Bob Jones to Joe Smith. The message contains a call identifier in the Call-ID field, and a sequence number in the CSeq field. The body is of type `application/sdp`. It describes a multicast session on “Mbone Engineering Issues”, on multicast group 224.2.0.1. The `m=audio 3456 RTP/AVP 0` line indicates the session is audio only, using codec 0 (which is PCM μ-law) on UDP port 3456.

SIP runs on top of either UDP or TCP, providing its own reliability mechanisms when used with UDP. For addressing, SIP makes use of uniform resource identifiers (URI’s) [16], which are generalizations of Uniform Resource Locators (URL’s), in common usage in the web. SIP defines its own URI, but its header fields can carry other URI’s, such as http, mailto [17], or phone [18]

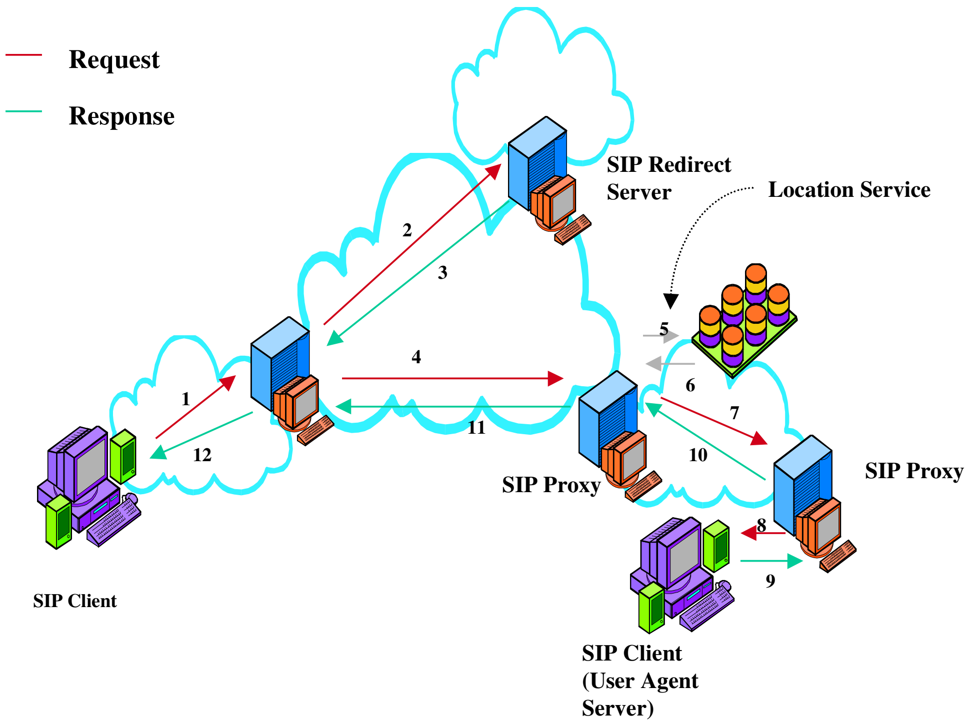

Figure 2: SIP Operation

URL's.

Like HTTP, a proxy or redirect server can destroy all transaction state after the transaction is complete. SIP transactions can complete even if a server crashes and reboots in the middle, losing all transaction state. SIP messages contain sufficient information to allow a rebooted server to treat the message correctly. This also means a server can safely clean up old state which has collected to due unusual failures or cases where the caller lets the phone ring for a long time. In addition, SIP allows subsequent transactions (such as a call termination, or feature invocation) to occur directly between the caller and callee without traversing through the same proxy or redirect servers. However, a proxy can insist on being in the signaling path for subsequent transactions for the same call through means of the Route and Record-Route header fields.

A typical SIP transaction is depicted in Figure 2.

The caller creates an INVITE request for some user, `sip:joe@company.com`. This request is forwarded to a local proxy (1). This proxy looks up `company.com` in DNS, and obtains the IP address of a server handling SIP requests for this domain. It then proxies the request to this server (2). The server for `company.com` knows about the user `joe`, but this user is currently logged in as `j.user@university.edu`. The server at `example.com` can know this through a static configuration, database entry, or a dynamic binding set up by the user using a SIP REGISTER message. So, the server redirects the proxy (3) to try this address. The local proxy looks up `university.edu` in DNS, and obtains the IP address of its SIP server. The request is then proxied there (4). The university server consults a local database (5), which indicates (6) that `j.user@university.edu` is known locally as `j.smith@cs.university.edu`. So, the main university server proxies the request to the computer science server (7). This server knows the IP address where the user is currently logged in, so it proxies the request there (8). The user accepts the call, and the response is returned through the proxy chain (9),(10),(11),(12) to the caller.

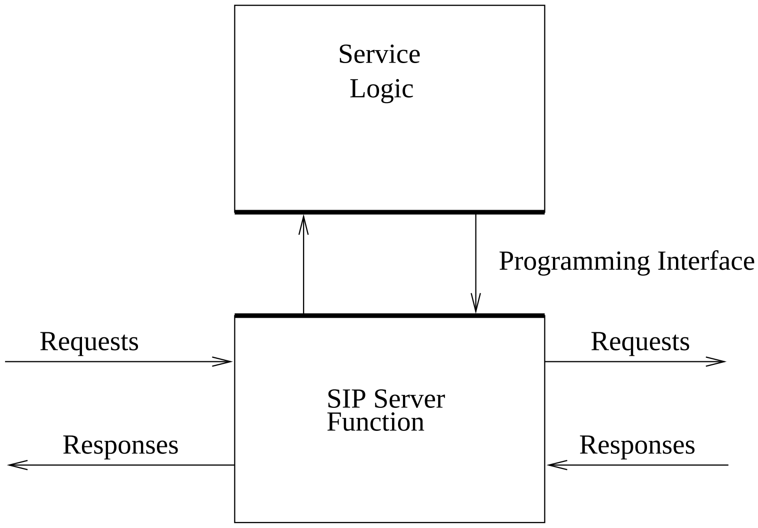

Figure 3: Model for Programming SIP Services

3 Programming SIP Services

The key to programming Internet telephony services with SIP is to add logic that guides behavior at each of the elements in the system. In a SIP proxy server, this logic would dictate where the requests are proxied to, how the packet should be formatted, and how the results are processed. For example, a simple service, such as call forwarding based on time of day, would require logic in the SIP server to obtain the time when a call setup arrives, and based on it, proxy the request to one destination or another. In general, the logic can direct the servers actions based on all sorts of inputs — time of day, caller, call subject, session types, call urgency, media composition, data obtained from web pages, and data obtained from directories. The logic may also instruct the server to generate new requests or responses.

Logic can also be added to user agents. However, since user agents are usually owned by end users, not network service providers, providing logic for them is a different problem. The breadth of platforms used, the security implications, and the trust models, are substantially different. For this reason, we consider only network servers for the remainder of this paper. However, the approaches for programming services advocated in this paper could be implemented in a user agent server (UAS) as well.

The basic model for providing logic for SIP services is shown in Figure 3. The figure shows a SIP server that has been augmented with *service logic*, which is a program that is responsible for creating the services. An interface exists between the two. When requests and responses arrive, the server passes information up to the service logic. The service logic makes some decisions based on this information, and other information it gathers from different resources, and passes instructions back to the server. The server then executes these instructions.

In order to define the details of this model, a number of issues must be resolved. These are:

- Where does the logic reside?
- When does the logic execute?
- What are the restrictions on the resources available to the program?

- What information about the SIP messages are provided to the program?
- What level of control does the program have over the server's execution?

There is no one solution for each of these issues. In particular, the solution for the last three issues depends very much on the level of trust between the server and the program. If the level of trust is low, very specific, structured information should be passed from the server to the program, and a very narrowly defined set of controls should be exposed to the program from the server. This restricts the set of services that can be defined, but provides a greater level of security. The server can be sure that the program cannot perform malicious operations, or cause the server itself to crash.

The amount of trust depends on who is creating the service logic. This can be the owner of the server, some third-party service provider, or even an end user. In the former two cases, the service logic is created by a network administrator, who has access to testing facilities, and can verify the service logic before it is placed in the network. In the latter case, the logic is created by a regular end user. There may be thousands or millions of end users. A network administrator cannot possibly test each piece of logic ahead of time, under all conditions; nor can the network administrator trust that end users will provide bug free, non-malicious code. The differing trust models demand different solutions.

3.1 Program Location

The service logic can either reside on the servers themselves, or in special computers separate from the servers. In the latter case, some protocol is needed for the interface between the server and the service logic. This can be a special purpose protocol, or can be some form of Remote Procedure Call (RPC). Distributed computing platforms, such as CORBA and DCOM, can also be used. This allows the location of the service logic to be independent of the interface itself. When the service logic and server are co-resident, their interface can be a simple API. Placing the logic in an external server has numerous advantages. It increases security. Malicious or buggy code which crashes has less effect on the server, since they are physically separated. There can be multiple computers executing the logic for a single server. This provides load balancing and improves scalability. On the other hand, executing the service logic on the same server simplifies the interface. Network issues, such as losses, delays, and encryption, can be ignored. Execution time for the logic is also improved, since it is not necessary to traverse a network.

3.2 Program Invocation Times

Not all services require the service logic to be consulted for every event or message that is received. A large class of services require the logic to be executed only when the initial INVITE message is received. Subsequent message processing rules can follow standard procedures defined by the protocol itself. Furthermore, some calls won't require any services at all. The SIP server should behave as it normally would, and not consult the service logic at any point. It is therefore necessary to have some means to specify at what point service logic is executed. The execution points can be defined by some administratively set policy, or they can be controlled dynamically by the service logic itself.

A related issue is whether the service logic is persistent or not. If the service logic runs as a separate process, it can remain active for the duration of the call (and beyond), and therefore be persistent. This would mandate an asynchronous interface between the logic and the server. It also introduces cleanup issues. Protocol or server errors may cause the service logic process for a particular call to remain active long after the call is over. Some means for cleanup is then needed to destroy these old processes. The advantage is that the service logic can pass control instructions back to the server at any time, rather than depending on the server to execute the service logic only on specific events. This enables numerous services (such as the click-to-dial service defined in [19]) which would otherwise be impossible to support.

As an alternative, the service logic can be executed synchronously. When the server receives a message, it begins execution of the service logic. The logic passes the control information back to the server, and ceases execution. This is most easily accomplished by having the service logic executed as a function call from the server. However, the service logic can also be executed as a separate process, but terminate once the control information is passed back to the server.

3.3 Resource Restrictions

The service logic can have access to a large number of resources. On the Internet, this includes name services (i.e., the Domain Name Service (DNS)), web pages, directories, mail servers, media servers, QoS controls, policy repositories, presence systems, and instant messaging services, to name a few. The logic can also have access to resources on other networks, such as the telephone network. The ability to query 800-number databases, for example, would allow migration of freephone services to the Internet.

With a breadth of resources comes a wide range of failure modes. The likelihood of bugs, malicious actions, unusual and untested scenarios increases. The right operating point, as we have indicated above, depends on the level of trust between the server and the logic. For end user defined services, access to resources will often need to be restricted. For administrator defined services, they should be more flexible.

3.4 Interface

The server will need to pass information about the SIP transaction, including message information and call states, to the service logic. This information can range from very abbreviated to very verbose. In the abbreviated case, only the message types (whether it is an INVITE, ACK or BYE for requests, and the response code for responses) and current state might be passed. In the verbose case, the entire message might be passed.

The right operating point, once again, depends on the level of trust and desired amount of flexibility. Verbosity lends to flexibility, but increases complexity and the possibility of error.

Information must also be passed from the service logic back to the server. This information is control data, instructing the server what to do next. This can also range from simple (a list of URI's to proxy to), to complex (an entire message to be sent).

4 Existing Models

The concept of separating the service logic from the server is certainly not new. This idea is at the heart of the Intelligent Network (IN) [20], a key component of the telephone network. The IN arose out of the need for separating services from the telephone switches, enabling rapid development of new services. The idea also exists in the web. Web servers separate the generation of content (the

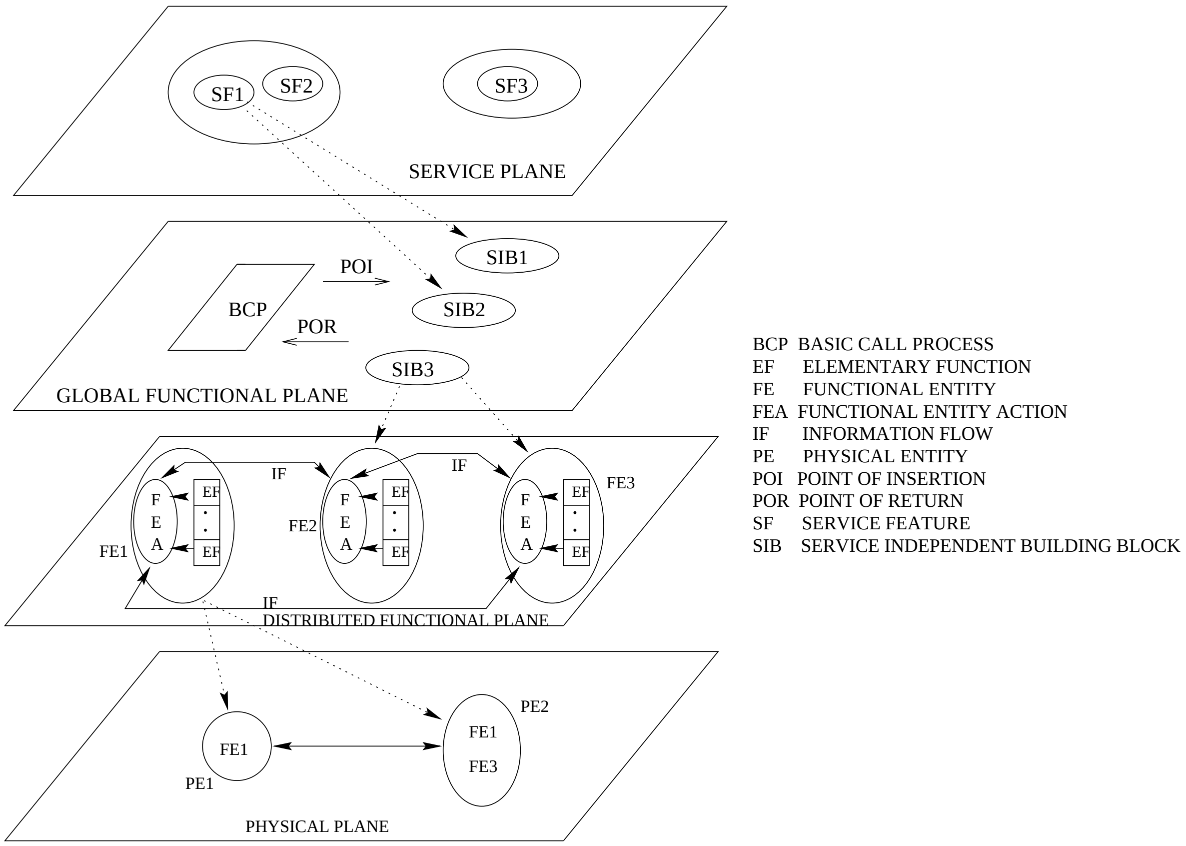

Figure 4: IN Conceptual Model

services) from the detailed protocol handling, using the Common Gateway Interface (CGI) [21], Java servlets, or server side Javascript.

The mechanisms used in these environments can provide valuable insight on how a solution for Internet telephony might be constructed. We therefore briefly review them in the next two sections.

4.1 Intelligent Network

The Intelligent Network is a collection of standards codified by the International Telecommunications Union, Telecommunications Sector (ITU-T). It is first introduced in Recommendation Q.1201 [22], which provides the model for IN, duplicated in Figure 4.

The figure shows that the IN can be modeled as a number of distinct planes. The first is the Service Plane, which contains services (such as call-forward and freephone). The Global Functional Plane (GFP) defines these services in terms of atomic operations, called Service-Independent Building Blocks (SIB's). A SIB can be thought of as a "network level" machine instruction inside a computer. It defines a basic function on a piece of data. Some of the SIBs are *Compare* for comparing two values, *Queue* for queuing a call, and *Screen* for screening calls. The GFP also defines the interaction between the telephone switch and the service logic on the general purpose computer, by means of a state machine on the switch (the Basic Call Process, or BCP), and remote procedure calls to the computer. The telephone switch has certain points in the progress of a call where remote calls can be made (Points of Initiation (POI)), and points in the call where the service logic can instruct the switch to return to (Points of Return (POR)). Examples of POI's are *Address*

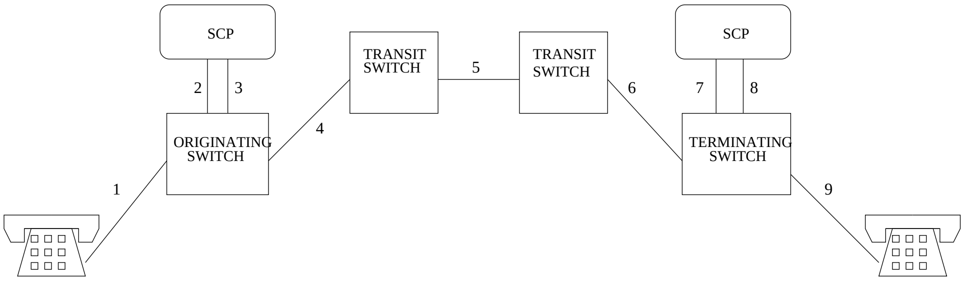

Figure 5: Call flow for a phone call

Collected and *Busy*, which occur in the BCP when the digits for the call are collected, and when the remote party signals busy, respectively. Examples of POR's are *Initiate Call* and *Clear Call*.

The next plane is the Distributed Functional Plane (DFP), which maps the abstract GFP into functional blocks, and then defines the required flow of information between them. Finally, the DFP is realized in the Physical Plane, which maps the functional entities into real devices. The IN defines a number of devices, including a Service Switching Point (SSP), which is a telephone switch, a Service Control Point (SCP) which is a general purpose computer that can execute the remote procedure calls from the switch, and a Service Node (SN), which executes service logic, but also has a switch fabric so that it can generate tones, play announcements, and provide additional services. The physical plane also defines the protocols among these entities. Most important among these is the IN Application Protocol, or INAP [23], which carries the information for the information flows defined in the DFP. INAP can be thought of as a special-purpose remote procedure call protocol.

A simplified example of a call flow for a phone call that makes use of IN services is shown in Figure 5. In this figure, a call originates at the calling phone, which dials an 800 number. The call signaling arrives at the originating switch (1). Since the switch does not know how to handle the 800 number directly, it asks the SCP for further instructions (2), and the response (3) specifies the further actions the switch should take, including a routing number where the call should be connected to (4). The call passes through another transit switch (5) and eventually arrives at the terminating switch. This switch also consults an SCP for instructions (6), and the response (7) tells it to complete the call to a subscriber line (8).

The IN has been standardized incrementally, starting with a baseline set of services and associated call models and protocols (call IN capability set 1, or CS1). Increasingly more complex services, models, and protocols have been developed as part of capability set 2, and more recently capability set 3. The services enabled with CS1 include freephone, televoting, follow-me, call screening, and call forwarding, among others. IN Capability Set 2 (CS2) supports more powerful features, such as call transfer, call waiting, message store and forward, and conference calling.

4.2 Web Models

There are a number of mechanisms which provide "services" (dynamic content) in the WWW. These can be broken into two types — server side, and client side. Server side mechanisms allow the content of the web response to be generated by either a separate process (in the case of the Common Gateway Interface (CGI)), or by the web server (in the case of server side Javascript, Java servlets, or Active Server Pages (ASP)). In the case of client side programming, the web page returned by the server can contain programs which execute to either interact with the user, or provide dynamic content. Java and Javascript are examples of this.

In the case of CGI, the generation of the content for the response is performed by a separate process. When the web server receives a request, it spawns a separate process (called a co-process) to execute the script. The standard output of the script process is connected to a handle on the server, as is the standard input. This means that when the script process reads from its standard input, or writes to its standard output, the data actually comes from or goes to the server.

Before spawning the process, the server also sets a number of environment variables. These environment variables are data that are accessible to the script process. The server uses the environment variables to pass information to the script. The information that is passed includes:

Request Headers: The values of various headers in the HTTP request are passed to the script. This includes information on who is making the request, what browser they are using, etc.

Server Information: Information about the server is passed to the script. This includes the type of server and the version of CGI.

Requested Data: The Uniform Resource Identifier (URI) that has been requested in the request is passed to the script process.

Body Information: Information on the length and type of the body in the request are passed to the script process.

User Information: Information on the IP address and identity of the requesting user are passed to the script.

In addition to setting these environment variables, the server passes the body of the request (often web form data) to the script process through the standard input of the script process. The script then processes the data, and returns a result.

The result is in the form of a HTTP response to be sent to the client, usually a success message with a web page of some sort in the body. This response is passed through the standard output of the script process, and is read by the server. The server fills in any necessary message headers, and sends to the response to the client. The script process then terminates.

5 SIP CGI

We concluded in section 3 that two mechanisms are needed for a complete service programming solution — a flexible, general purpose one for trusted users, primarily targeted at administrators, and a simpler, more restricted one targeted at untrusted users, such as consumers. In the web, CGI is the most flexible mechanism for creating dynamic content. This is because of several characteristics:

Language independence: CGI works with perl, C, VisualBasic, tcl, and many other languages. This provides maximum flexibility.

Exposes all headers: CGI exposes the content of all of the headers in an HTTP request to the CGI application through environment variables. An application can make use of these if it sees fit, and ignore those it doesn't care about. Since the encoding of SIP messages is similar to the encoding of HTTP messages, the environment variable approach can be directly applied to SIP.

Creation of Responses: CGI is advantageous in that it can create all of the content of a response, including headers, response codes and reason phrases, in addition to content, through standard out. In SIP, we will need this level of flexibility as well.

Access to any resources: Since the CGI script is a general purpose program, it can use existing API's to access any desired network service.

Because of these properties, CGI makes an ideal starting point for service creation in an IP telephony context. The interface between the server and the service logic is very flexible (enabling entire packets to be sent back and forth), and the set of services accessible by the service logic is unlimited.

The similarity of HTTP and SIP makes its application to Internet telephony straightforward. Usage of CGI for Internet telephony service creation has a number of other advantages:

Component Reuse: Much of the CGI componentware allows for easy reading of environment variables and parsing and generation of header fields. Since SIP reuses the basic syntax of HTTP, all of these tools are immediately applicable to SIP CGI.

Familiar Environment: Many web programmers have familiarity with CGI.

Ease of extensibility: Since CGI is an interface and not a language, it becomes easy to extend and reapply to other protocols, such as SIP.

In the sections below, we discuss SIP CGI in more detail.

5.1 Basic Operation

Like traditional HTTP CGI, a SIP CGI script is invoked when a SIP request arrives at a server. The server passes the body of the message to the script through its standard input, and sets environment variables containing information on the message headers, user information, and server configuration. The script performs some processing, and generates some data which is written to the standard output of the script. This data is then read by the server, and the script terminates.

Unlike HTTP-CGI, however, the output of the script need not be the response to send. A script can also instruct the server to proxy a request, or to create an entirely new request. In fact, the script can instruct the server to generate multiple messages. This is accomplished by using the message multiplexing rules in SIP to place several messages in the script output.

Another important difference between SIP CGI and HTTP CGI is the persistence model. In HTTP CGI, a request arrives, the script executes, a response is generated, and the script terminates. The server generates a response and the transaction is complete. In SIP, however, a script can cause requests to be proxied. This means that the server will eventually receive responses to these requests, and these responses must be passed to the script for processing. This means that after generating its output, the script must somehow persist, and continue interacting with the server to process subsequent responses.

There are two ways in which this persistence can be accomplished:

- Allow the script to be executed once, and then continue running for the duration of the transaction. This would require new IPC mechanisms, since the use of environment variables for server to script CGI wouldn't support asynchronous notification of packet events.

- Stick with the current HTTP-CGI model, so that the script is called, some output is generated, and then it terminates. We can achieve persistence by re-executing the script on the next event with a state token.

While the first option has lower process overhead, it departs significantly from the clean model presented in HTTP-CGI. Therefore, we chose the second approach. A state token (called a *Script Cookie*) is passed from the script to the server through a SIP-CGI meta-header. When the script is re-executed at some later point, the server passes the cookie back to it through environment variables. This token is opaque to the server, and can contain anything of use to the script. In essence, the execution of the script is a procedure call, with the particular procedure dependent on the semantic of the cookie.

Since the script does not need to be executed on the arrival of every message for every service, the script is capable of instructing the server not to execute the script for subsequent messages. This avoids the needless overhead of starting a new process for each message. This feature is accomplished by having the script pass *meta-headers* in the messages from the script to the server. These meta-headers contain instructions for the server, and are removed before the packet is sent. This provides a function similar to triggers in the IN model.

The persistence mechanism allows the script to issue new instructions as responses to its new and proxied requests are received. In addition to proxying requests, creating new requests, and generating a response, the script can also instruct the server to forward a response, previously received, upstream towards the caller. Each response that is received is stored in the server until the transaction is complete. The server associates a unique identifier with each response, and this is passed to the script through environment variables. At any point, the script can instruct the server to return a response by generating a special message type that contains the identifier of the response to return.

In addition to controlling which messages get sent, and when, a script can control the headers in these messages. By default, a SIP server will fill in all of the headers in proxied requests, forwarded responses, or generated responses, according to the rules in the SIP specification. If this behavior is acceptable, the script need not specify any headers. However, a script can optionally instruct the server to place a specific header in a message, replace a header in a message with a new one, or delete a header from a message. The script also has controls on whether the body of the message should be copied, updated, or removed.

Details on the operation of SIP CGI can be found in [24].

5.2 Example CGI Operation

Assume the following request was received, triggering the execution of a CGI script:

```
INVITE sip:astronomer@lab2.university.edu SIP/2.0
Via: SIP/2.0/UDP ganymede.university.edu
Subject: Io's orbit
From: sip:physicist@university.edu
To: sip:astronomer@university.edu
Call-ID: 089y30n0983h2f0@112.34.55.2
CSeq: 1 INVITE
Contact: sip:j.smith@ganymede.university.edu
```

The script outputs the following:

```
CGI-PROXY-REQUEST sip:b.jacobs@lab2.university.edu SIP/2.0
Contact:
Subject: Earth's rotation

SIP/2.0 180 Ringing

CGI-SCRIPT-COOKIE asd-9unas SIP/2.0
```

The script output contains three messages. The first instructs the server to proxy the received request to `b.jacobs@lab2.university.edu`. The Contact header with no value instructs the server to remove the Contact header from the proxied request. The Subject header instructs the server to replace the Subject header in the proxied request with the one specified. The server will perform these operations and then generate the following proxied request:

```
INVITE sip:b.jacobs@lab2.university.edu SIP/2.0
Via: SIP/2.0/UDP ganymede.university.edu
Subject: Earth's rotation
From: sip:physicist@university.edu
To: sip:astronomer@university.edu
Call-ID: 089y30n0983h2f0@112.34.55.2
CSeq: 1 INVITE
```

The second message in the script output instructs the server to generate a ringing response towards the caller. The server will fill in all of the required header fields and send the following response:

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP ganymede.university.edu
From: sip:physicist@university.edu
To: sip:astronomer@university.edu
Call-ID: 089y30n0983h2f0@112.34.55.2
CSeq: 1 INVITE
```

The last message in the script output instructs the server to set the script cookie to `asd-9unas`, a string with meaning only for the script. Next time the script is invoked (when a response to the proxied request arrives), this cookie is passed back to the script in an environment variable.

The header processing rules provide a script flexibility in choosing its level of control. A simple script can let the server handle all header processing. A more complex script can completely manage the server processing by generating all of the headers.

5.3 Services Enabled

SIP CGI provides a powerful tool for developing Internet telephony services rapidly. Many of the traditional telephone services, such as automatic call distribution (where the calls are forwarded to operators based on some specified logic), call forward busy, call forward unconditional, and follow-me, are easily supported with small perl scripts. See Appendix A for a perl script for unconditional call forwarding.

However, these are just the tip of the iceberg. Since the CGI script is a normal process, it can have access to network and system resources, such as email, web, database, and file store. Consider some of the possibilities:

Redirection to web: When a call arrives at a server, the CGI script can lookup the called user in a corporate database, and using the information found there (such as picture, phone number, or email) generate a web page dynamically containing reach information for that user. This web page can be returned to the caller in a SIP response.

Instant Message when busy: When a call arrives at a server, the CGI script accesses an IN database to find a phone number for the user. The script finds a telephony gateway to complete the call over the phone network. If the user's phone is busy, the script assumes this is because they are logged in, and sends an instant message to them over the Internet, indicating who called. The script then generates a busy message for the caller.

Forward to Email: When a call arrives at a server, the CGI script looks up the callee in a database and obtains their email address. The script then sends email to the user, containing the caller, time of call, and call subject. The caller is redirected to a voicemail server to record a message.

Access to email, web, and instant messaging services are already available through API's in existing languages. These can be used directly to provide these services for IP telephony.

6 Call Processing Language

We found that the IN conceptual model has a number of properties that make it a good starting point for end user defined services. IN defines a very restricted interface between the switch and service logic. The model whereby services are constructed by the interconnection of a fixed set of building blocks is also very attractive. This limits the set of resources available to the program. It also limits the ways in which programs can be constructed. Since they are restricted to interconnection of SIB's in a directed acyclic graph (DAG), automatic tools can be applied to bound running times. This property is extremely attractive for end user service creation.

To instantiate this concept for IP telephony, we have defined a new language, called a Call Processing Language (CPL). The language was designed to mimic the structure provided in IN service creation. Users can write scripts in this language to define their own services. Alternatively, the CPL can be generated through graphical tools which collect user preferences in a user-friendly manner. These scripts can then be uploaded to the service provider, and automatically placed in the network. This makes the service instantly available.

6.1 Language Requirements

Because CPL scripts run on a service provider's platform, there are numerous requirements imposed on the language:

Verifiability: The service provider must be able to verify automatically that a user-described service is well-formed and can be successfully executed by its server. It needs to be able to verify this at the time the script is submitted, as discovering it only during execution can lead to a user not being able to receive calls. Of course, it is not possible to *guarantee* successful execution at submission time — unexpected network failures, for example, can cause a service to be unsuccessful — but a server can confirm that it is able and willing to carry out all the parts of the specified service.

Completion: It must be possible to determine, at submission time, that the service specified in the CPL will be completely executed in a finite amount of time. This implies that the language in which services are specified cannot be Turing-complete, and in particular certain constructs, such as generalized looping or non-timed calls no external services, cannot be present; otherwise, this would be an undecidable problem.

Safety of execution: The service description should not be able to represent unsafe actions, such as modifying other users' data, or examining arbitrary files on the server. Furthermore, it should not be possible for it to interfere with the operation of the server by using large amounts of CPU time, memory, storage, network bandwidth, or other resources.

Standardized representation: Because customers and service providers may well have software from different vendors, it is important that the service descriptions be compatible between different tools.

Readable and producible by humans and machines: It is unreasonable to expect the average end-user to be able to write a description of a complex service by hand. It needs to be possible to create an authoring tool which can generate service descriptions. However, more advanced users, or independent vendors writing "template" outlines of services, will want to have the added flexibility that hand-authoring provides. Furthermore, it is important that authoring tools be able to understand hand-generated services completely, and that humans can read, debug, and edit descriptions created with authoring tools.

Protocol-independence: For at least the medium term, the Internet Telephony world will have to contend with at least the three protocols described in the introduction: SIP, H.323, and MGCP. The Call Processing Language describes the services at a sufficiently high level that they can control servers running any of the three protocols. Service descriptions can therefore be portable between different servers or vendors.

Ease of transport: Because the user and the service provider are physically separated, there needs to be a way for service descriptions to be transported easily and reliably from the user to the server, and ideally back again (so services can be examined or modified). This speaks for single file representations.

Security of transport: Since call processing languages may reveal user preferences and other sensitive information (such as passwords, usernames, and phone numbers) they need to be transported securely to the service provider. The service provider must also ensure that only the user or other authorized entities modify call handling instructions. This makes both encryption and authentication key requirements for transport. The use of a SIP REGISTER message has been proposed as a transport for both SIP CGI and the CPL [25].

6.2 Server Interaction Model

In the case of SIP-CGI, the details of message exchanges are exposed to the script. This requires the script to know details of the protocol operation. In addition, the interaction between the script and the server is "event driven." The server invokes the script when certain events (namely message arrivals) occur. This requires the service definition within the script to be based on a state machine model of protocol operation. In addition, the CGI script maintains state for just a single transaction.

This model is inappropriate for the CPL. One of the goals stated above was for the script to be protocol independent. This means a detailed knowledge of messages and elaborate state machines cannot be assumed. In addition, since CPL scripts are written by end users, they cannot be assumed to understand how a protocol operates. Finally, since many services require knowledge of call state, the persistence of CGI scripts is not sufficient for the CPL.

The model used in the CPL abstracts the operation of the protocol to providing a basic set of services, and being able to generate a basic set of information. The language contains primitives which cause various messages to be sent, and those primitives return results based on the protocol message exchanges. Which messages are sent, and how long it takes for the information to be obtained, are hidden from the CPL. This is much in line with the IN model, which separates the functional and physical planes.

6.3 Language design

For the actual structure of the Call Processing Language, a number of possible approaches were considered. Among these were:

- Describing services with a restricted subset of an existing scripting language, such as Perl [26], Tcl [27], or Python [28].
- Having services be implemented with user-created programs which run in a virtual machine such as the Java Virtual Machine.
- Designing a syntax for a new programming language, similar to, for example, the e-mail filtering language Sieve [29].
- Basing service descriptions on the Extended Markup Language, XML [30].

After considering all these, we concluded that, considering the language requirements described above, an XML-based language was the best choice. XML is a powerful, flexible text-based syntax for describing complex data relationships. It is a subset of SGML, and is visually and syntactically very similar to HTML. One of the goals of XML is to be readable and createable by both humans and machines, a goal shared by the CPL. This feature is difficult to support in other programming languages. While it is certainly possible for a visual front-end to generate, for example, Java code, it is not generally possible for such a front-end to understand existing code (generated by some other tool, or by a human) well enough to edit it meaningfully on a semantic level.

A natural way of describing services is with a *decision graph*. Decision graphs are collections of choices to be made and actions to be taken in the course of performing the service. These decisions and actions are arranged in a directed acyclic graph (DAG); control begins at a single root node, and each node can have several outputs, depending on the result of the choice or action taken at

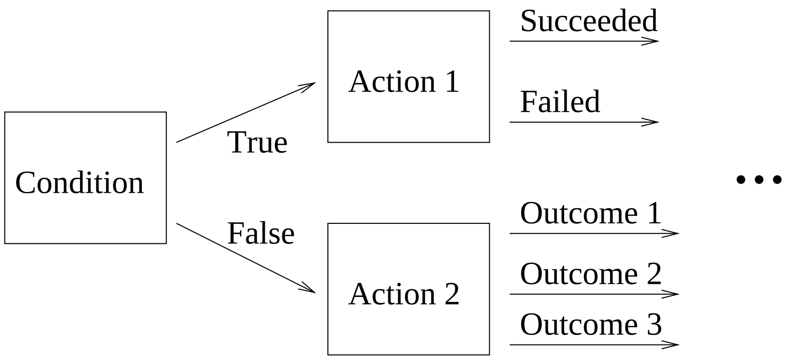

Figure 6: CPL Decision Model

that node. Node outputs then lead down the tree to further actions or decisions. An example of a decision graph is shown in figure 6. Decision graphs are common in Intelligent Network service creation environments. XML documents represent tree structures with optional links — exactly this sort of directed graph. Thus XML maps easily to a very natural and convenient representation of services using the CPL.

This representation of services as a directed acyclic graph also implicitly guarantees most of the CPL's safe execution requirements. As the flow of control moves only "downward" in a decision tree, we can conclude (if the decision tree is well-formed) that the service must eventually reach a leaf node and terminate, and that the resources the service can use are finite and proportional to the length of the longest branch of the tree in the worst case. This means the safety of an XML based CPL can be checked by searching for cycles in the graph it represents, and computing the maximum depth of the tree. The syntax and semantics can be verified using XML validation against a Document Type Definition (DTD). Any of the alternative approaches which used existing languages would not be able to make this guarantee. Common features of general languages, such as general looping or recursion, would make such a guarantee (and indeed, thanks to the undecidability of the halting problem, *any* such guarantee) impossible. However, removing these features would cripple any existing language beyond recognition; so attempting to base service specification on these languages is probably inappropriate.

XML is also a good choice since it is easily extended. Every tag and attribute has a name explicitly specified; thus, a parser can immediately determine whether it can support all the requested features, and decide what to do if it cannot support them. Furthermore, XML has built-in mechanisms for the additions of new tags and attributes, which can come from namespaces specified in the head of the document.

XML is by no means perfect. It tends to be verbose, requiring relatively long programs for simple services. In addition, since XML is not a programming language, but rather a syntax, inclusion of certain language features (such as variable assignment) are awkward. However, its limited flexibility is more of an advantage than a disadvantage in this application.

6.4 Language structure

As mentioned above, the CPL consists of a decision graph describing the service. The individual nodes of the decision graph are the primitives of the language, specific choices to be made or actions to be taken. The complex service is assembled out of these primitives. Nodes have outputs, which lead to further nodes. It is possible for some or all outputs of a node to be left unspecified. This

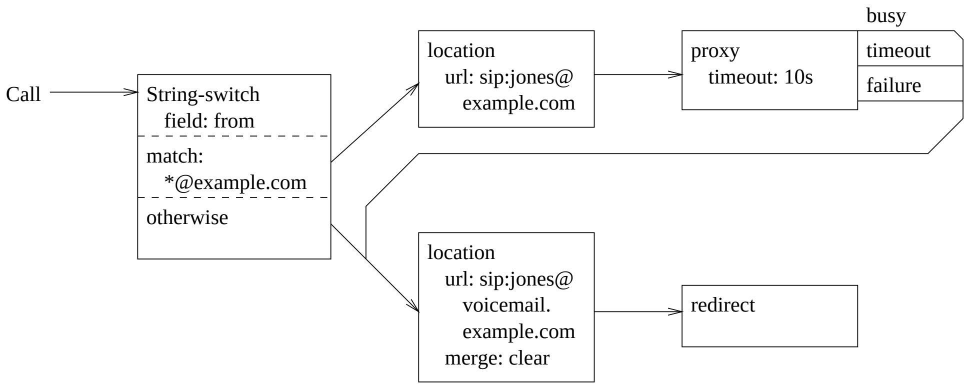

Figure 7: Example CPL Decision Graph

means that the server should take whatever normal or default action it would take in the current call state, in the absence of a CPL script. Similarly, many parameters to conditions or actions can be left unspecified, also taking default values. An example graph for simple caller based forward or redirect service is shown in Figure 7.

The mapping of the CPL onto XML is straightforward. There is an enclosing XML tag named `call` which contains an entire CPL script, indicating the point where execution begins. Both nodes and their outputs are represented as XML tags; parameters are represented as XML tag attributes. Node tags typically contain output tags, and vice-versa, representing descent down the decision tree. Convergence (where several outputs point to a single node) is represented with links.

6.5 Language primitives

There are four broad classes of language primitives in the CPL. First, there are switch nodes, which represent decisions a script can make. Secondly, location nodes indicate the locations where users can be found, either directly or by reference. Signaling actions are the core of the language; they control the behavior of the underlying signaling protocol. Finally, non-signaling actions allow non-call actions to be taken. We discuss each of these types of primitives in turn.

Switch nodes: Switch nodes allow the CPL script to make decisions which determine the future actions to perform. Two types of decisions exist. The first type of decisions depend on the parameters of the original call which triggered the script, such as its sender, its recipient, the types of media involved, the total bandwidth required, and so forth. The other sort of decisions are those which are based on global state independent of the call; the only current example of this is whether the current date or time falls within a given range.

Location nodes: Location nodes specify the locations which subsequent signaling actions should contact. Locations can be specified in two ways: directly, as literal URLs, or indirectly. Indirect lookup allows the server to retrieve a list of locations to contact from an external source; for example, a database server or a SIP registrar associated with the CPL server.

A CPL script always has a set of locations specified as an implicit global variable. Location nodes modify this implicit variable, either by adding to the set, or by clearing the set and adding new values to it.

Signaling actions: Signaling actions form the core of the CPL. They control the broad behavior of the underlying signaling protocol. There are three basic signaling actions: **proxy**, **redirect**, and **response**. Proxy is the most powerful of these; it causes the CPL server to forward the call to the currently specified location set, and waits for responses from it. The server automatically picks the best response of these. If the best response was success (i.e. the call was picked up), the script terminates, as call setup is complete. If not, some output of the node, such as **busy**, **noanswer**, or **failure**, is indicated, and the subsequent nodes pointed to by that output are executed.

Redirect and response are both simpler actions. They immediately terminate the execution of the script, since both these actions imply that this call server is done handling the call. Redirect sends a redirection request to the current set of locations; response allows the server to send a failure condition or reject the call.

Non-signaling Actions: Non-signaling actions allow a script to record events, or notify a user of them. For instance, a record could be stored in a log at the server, allowing a user to categorize calls they receive. Alternately, an action could send electronic mail or an instant message to a user when some event occurs; this allows a script to warn users about failure conditions, when a malfunctioning script might prevent them from receiving phone calls, or alternatively to alert them of incoming calls when they are not in a position to be reached by telephone.

Because XML is easily extended, adding additional primitives or additional parameters to existing primitives is simple, and does not harm backward compatibility. A number of example CPL scripts are given in Appendix B. Details on the CPL can be found in [31].

7 Conclusion

Internet telephony is much more than point to point voice transport on the Internet. It has the potential for combining the best of traditional telephony services and Internet applications. This will enable new classes of services which don't exist in either network today. However, such flexibility introduces new challenges. How are such services to be programmed? How can existing tools for programming Internet services be leveraged for Internet telephony?

We have investigated this problem from two perspectives — programming services as a trusted user (such as an administrator), and as an untrusted user (such as a consumer). The requirements for both are quite different — flexibility is paramount in the former case, and security in the latter. In recognition of this, we have developed the SIP Common Gateway Interface (CGI) for programming services in the former case, and the Call Processing Language (CPL) for the latter. SIP CGI is based on the successful web CGI model, and affords the same flexibility. The CPL is an XML-based language, which can be verified automatically to provide security.

A number of open issues remain. SIP CGI has a number of drawbacks we are working to resolve. First, the script must run on the same machine as the server. Second, CGI scripts cannot easily provide asynchronous directives to the server. We are looking at modifications to CGI to rectify these problems. The Call Processing Language is still under definition; choosing the right set of primitives is a complex issue. For both CGI and the CPL, *feature interaction* issues require further study. We have already identified a few, but more investigation is needed.

SIP CGI has been implemented on two separate SIP servers, and the CPL is under development on both.

References

[1] H. Schulzrinne, "Re-engineering the telephone system," in *Proc. of IEEE Singapore International Conference on Networks (SICON)*, (Singapore), Apr. 1997.

[2] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: a transport protocol for real-time applications," Request for Comments (Proposed Standard) 1889, Internet Engineering Task Force, Jan. 1996.

[3] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation protocol (RSVP) – version 1 functional specification," Request for Comments (Proposed Standard) 2205, Internet Engineering Task Force, Oct. 1997.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," Internet Draft, Internet Engineering Task Force, Oct. 1998. Work in progress.

[5] 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.

[6] M. Arango, A. Dugan, I. Elliott, C. Huitema, and S. Pickett, "Media gateway control protocol (MGCP)," Internet Draft, Internet Engineering Task Force, Feb. 1999. Work in progress.

[7] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: session initiation protocol," Internet Draft, Internet Engineering Task Force, Jan. 1999. Work in progress.

[8] H. Schulzrinne and J. Rosenberg, "A comparison of SIP and H.323 for internet telephony," in *Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, (Cambridge, England), July 1998.

[9] H. Schulzrinne and J. Rosenberg, "Signaling for internet telephony," in *Proc. of 6th IEEE International Conference on Network Protocols (ICNP)*, (Austin, Texas), Oct. 1998.

[10] H. Schulzrinne and J. Rosenberg, "The session initiation protocol: Providing advanced telephony services across the internet," *Bell Labs Technical Journal*, vol. 3, pp. 144–160, October-December 1998.

[11] H. Schulzrinne and J. Rosenberg, "Internet telephony: Architecture and protocols – an IETF perspective," *Computer Networks and ISDN Systems*, vol. 31, pp. 237–255, Feb. 1999.

[12] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," Request for Comments (Proposed Standard) 2068, Internet Engineering Task Force, Jan. 1997.

[13] H. Schulzrinne and J. Rosenberg, "SIP call control services," Internet Draft, Internet Engineering Task Force, Feb. 1998. Work in progress.

[14] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies," Request for Comments (Draft Standard) 2045, Internet Engineering Task Force, Dec. 1996. (Obsoletes RFC1521); (Updated by RFC2184); (Updated by RFC2231).

[15] M. Handley and V. Jacobson, "SDP: session description protocol," Request for Comments (Proposed Standard) 2327, Internet Engineering Task Force, Apr. 1998.

[16] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifiers (URI): generic syntax," Request for Comments (Draft Standard) 2396, Internet Engineering Task Force, Aug. 1998.

[17] P. Hoffman, L. Masinter, and J. Zawinski, "The mailto URL scheme," Request for Comments (Proposed Standard) 2368, Internet Engineering Task Force, July 1998.

[18] A. Vaha-Sipila, "URLs for telephone calls," Internet Draft, Internet Engineering Task Force, Feb. 1999. Work in progress.

[19] S. Petrack and L. Conroy, "The PINT profile of SIP and SDP: a protocol for IP access to telephone call services," Internet Draft, Internet Engineering Task Force, Nov. 1998. Work in progress.

[20] I. Faynberg, L. Gabuzda, M. Kaplan, and N. Shah, *Intelligent Network Standards: their Application to Services*. New York: McGraw Hill, 1997.

[21] K. Coar and D. Robinson, "The WWW common gateway interface version 1.1," Internet Draft, Internet Engineering Task Force, Dec. 1998. Work in progress.

[22] International Telecommunication Union, "Principles of intelligent network architecture," Recommendation Q.1201, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, 1992.

[23] International Telecommunication Union, "Intelligent network interfaces," Recommendation Q.1218, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, 1995.

[24] J. Lennox, J. Rosenberg, and H. Schulzrinne, "Common gateway interface for SIP," Internet Draft, Internet Engineering Task Force, Nov. 1998. Work in progress.

[25] J. Lennox and H. Schulzrinne, "Transporting user control information in SIP REGISTER payloads," Internet Draft, Internet Engineering Task Force, Mar. 1999. Work in progress.

[26] L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl*. Sebastopol, California: O'Reilly, 2nd ed., 1996.

[27] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Reading, Massachusetts: Addison-Wesley, 1994.

[28] M. Lutz, *Programming Python*. Sebastopol, California: O'Reilly, 1996.

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

[30] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0," W3C Recommendation REC-xml-19980210, World Wide Web Consortium (W3C), Feb. 1998. Available at http://www.w3.org/TR/REC-xml.

[31] 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.

A Example CGI Script — Call Forwarding

A perl script providing a call forwarding service is shown in Figure 8. The script makes use of a Berkeley DB database file, and maps it to an associative array. When a request arrives, causing the script to be invoked, the script looks up the user in the To field, and obtains the forwarding location from the database. The call is then proxied to that location, or if there was no match in the database, an error is returned.

B Example CPL Scripts

We show here some example CPL scripts. These examples illustrate the types of services that are envisioned for the CPL, and the nature of their descriptions.

B.1 Call Redirect Unconditional

One simple script is one which unconditionally redirects calls to a particular address, depicted in Figure 9.

B.2 Call Forward Busy/No Answer

The script in Figure 10 attempts to ring a call at a standard location, and if the recipient is not available there, forwards the call to a voicemail server instead.

This example also illustrates XML's linking possibilities: since we want the same action to occur on busy as on no answer, we define a link on one node. This allows other nodes to reference that link rather than repeat parts of the script. (When a script is submitted, the server would verify that the links do not give rise to any cycles.)

B.2.1 Time-of-Day routing

Figure 11 illustrates a condition based on a non-call-related criterion. Note also the default dest value for the first proxy statement, and lack of specified outputs for either one.

```perl
#!/usr/bin/perl -w

# We store the list of users in DB_File, a Berkeley database.
use DB_File;

# This subroutine gets invoked when there is an error condition.
# It takes a status code (the SIP numerical response code) and
# reason phrase as arguments, and prints a line to standard out.
# This line is a CGI action for returning an error response

sub fail {
    my($status, $reason) = @_;
    print "SIP/2.0 $status $reason\n\n";
    exit 0;
}

# We now define an associative array, called addresses, and
# bind it to the database file addresses.db. This means that
# $addresses{'sip:user@example.com'} contains the forwarding address
# for user@example.com

tie %addresses, 'DB_File', 'addresses.db'
    or fail("500", "Address database failure");

# Retrieve the To field of the SIP request through the environment
# variable HTTP_TO.

$to = $ENV{'HTTP_TO'};

# If it wasn't defined, return an error to the caller

if (! defined( $to )) {
    fail("400", "Missing Recipient");
}

# Obtain the forwarding address from the database

$destination = $addresses{$to};

# If there was no forwarding address, return a error to the caller

if (! defined( $destination )) {
    fail("404", "No such user");
}

# Forward the call using the CGI-PROXY-REQUEST-TO CGI action,
# and instruct the server never to execute the script again for
# this transaction

print "CGI-PROXY-REQUEST-TO $destination SIP/2.0\n";
print "CGI-Reexecute-On: never\n\n";
untie %addresses; # Close db file
```

Figure 8: Perl script for Call Forwarding

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

Figure 9: Call Redirect Unconditional Script

```
<call>
  <!-- Proxy the call to jones -->
  <location url="sip:jones@jonespc.example.com">
     <proxy timeout="8s">

       <!-- When busy, forward to voicemail -->
       <busy>
         <location url="sip:jones@voicemail.example.com" merge="clear"
                   id="voicemail" >
            <proxy />
         </location>
       </busy>

       <!-- When there is no answer, jump to the voicemail link above
            and also forward to voicemail -->
       <noanswer>
         <link ref="voicemail" />
       </noanswer>
     </proxy>
  </location>
</call>
```

Figure 10: Call Forward Busy/No Answer Script

```
<call>
  <time-switch>

    <!-- During the work week, contact the user at his
 registered locations -->
    <time day="1-5" timeofday="0900-1700">
      <lookup source="registration">
        <success>
          <proxy />
        </success>
      </lookup>
    </time>

    <!-- The rest of the time, forward the call to voicemail -->
    <otherwise>
      <location url="sip:jones@voicemail.example.com">
        <proxy />
      </location>
    </otherwise>
  </time-switch>
</call>
```

Figure 11: Time of Day Routing Script