Programming Internet Telephony Services

Jonathan Rosenberg
Bell Laboratories
jdrosen@bell-labs.com

Jonathan Lennox
Columbia University
lennox@cs.columbia.edu

Henning Schulzrinne
Columbia University
hgs@cs.columbia.edu

March 18, 1999

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, use instant messages in place of call waiting notifications, or have call logs reported via email. 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 a 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 different services from different providers.

In this paper, we consider this problem in more detail. Since the services are ultimately realized through Internet telephony signaling protocols, we discuss 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. We propose our solution for service creation by trusted users — the Common Gateway Interface (CGI) — in section 4 and for service creation for untrusted users — the Call Processing Language (CPL) — in section 5. We then summarize in section 6 and highlight some of the open issues.

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-enabling 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 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 gateways, 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 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 3.5), for developing services. Finally, SIP's 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 give an example SIP call setup here for illustrative purposes.

A typical SIP transaction is depicted in Figure 1. The caller (also known as the *User Agent Client*, or UAC) creates an INVITE request for some user, `sip:joe@company.com`. This request is forwarded to a local SIP proxy server (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 redirect server for `company.com` knows about the user `joe`, but this user is currently logged in as `j.user@university.edu`. The redirect server at `company.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. The caller then acknowledges receipt of the response (not shown), and media can flow directly between the parties.

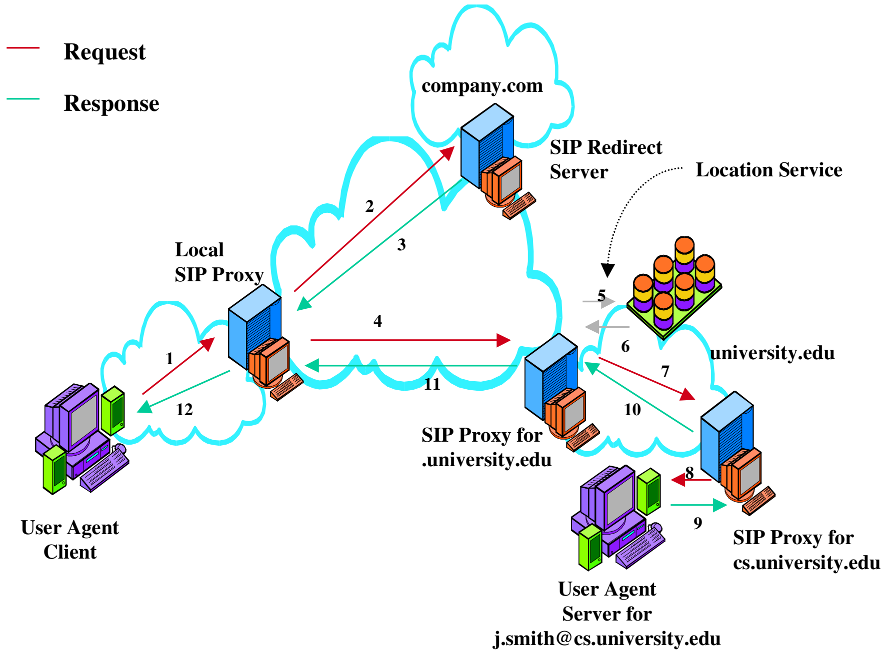

Figure 1: SIP Operation

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 responses 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 server's 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 (i.e., end system software). 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).

The basic model for providing logic for SIP services is shown in Figure 2. 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?

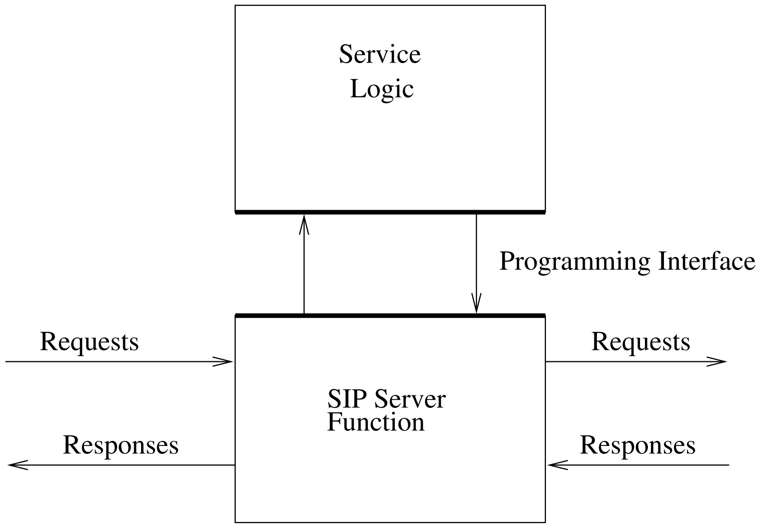

Figure 2: Model for Programming SIP Services

- 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 (as it may be with consumer-defined logic), 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. For trusted users (such as administrators or privileged users in a corporate environment), the trust levels are higher, and greater flexibility is warranted.

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 loss, delay, 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 [12]) 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 System (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 along with a copy of the server state.

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).

3.5 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) [13], 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. When a call setup message arrives at a telephone switch, the switch contacts a separate device, a *Service Control Point*, or SCP, to receive instructions about call progress. The IN standards define a *basic call state model*, or BCSM, which is used to define the controls the SCP has over the switch. This model contains the basic states for a call, and events that cause the model to move between states. The switch is configured with a number of *decision points*, or DPs, which are state changes at which the switch should ask for input from the SCP. When a DP is reached, the current call state and relevant information is reported to the SCP. The SCP can then make a decision about what to do next. It exercises control over the switch by instructing it to proceed to a particular state. The SCP can also instruct the switch to arm or disarm DPs for the remainder of the call.

The concept of separating service logic from servers also exists in the web. Web servers separate the generation of content (the services) from the detailed protocol handling, using CGI, Java servlets, Active Server Pages (ASP) or server-side Javascript. 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 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. The server also sets a number of environment variables before spawning the co-process. These variables are used to pass information, such as request details and user information, to the script. The body of the request is written to the standard input of the script. The script writes the response to be sent to its standard out, and terminates. The server reads the output, and sends it to the web browser.

4 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 [14] is the most flexible mechanism for creating dynamic content. This is because it possesses the following 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 control all aspects of the response, including headers, response codes and reason phrases, in addition to the content. In SIP, we will need this level of flexibility, since services are defined largely through headers in the responses.

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 network services accessible by the service logic is unlimited.

The similarity of HTTP and SIP makes CGI's 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 are familiar 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.

4.1 Basic Operation

Like traditional HTTP CGI, a SIP CGI script is first 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. The implication is that after generating its output, the script must somehow persist, and continue interacting with the server to process subsequent responses. One option was to keep the script process active for the duration of the transaction. This departs substantially from the HTTP CGI model, however. Instead, after processing a message, the script passes a state token (called a *Script Cookie*) to the server through a SIP CGI *meta-header* (A meta-header is a directive passed from the script to the server inside a SIP message. Meta-headers are removed by the server before forwarding requests). 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.

Not all services require the CGI script to be executed for each message that is received. For example, a call-forward unconditional service only requires special logic to be executed when the INVITE is first received. Responses to the proxied INVITE can be processed based on the rules in the SIP specification. To avoid the needless execution of scripts in these cases, the script is capable of instructing the server not to have itself re-executed when subsequent messages arrive. This feature (which is similar to event DPs in the IN) is also implemented by means of a SIP CGI meta-header. The meta-header contains a description of the conditions under which the script should be executed in the future, applicable to this call only.

In addition to proxying requests, creating new requests, and generating a response, the script can 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. When the script is executed, the identifier of the response which triggered execution is passed to the script through environment variables. The script can store these identifiers in the script cookie, so that they can be recalled at a later time. When the script is re-executed at a later time (upon the arrival of another response), the script can instruct the server to return a response previously received. The server does this by placing a special message in the script output. This message contains the identifier for the response in the Request-URI field of the message.

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 in the messages it outputs. 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 [15].

4.2 Example SIP CGI Operation

Assume the following request was received, triggering the execution of a SIP 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@physics.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 short messages, separated by blank lines. The first instructs the server to proxy the received request to `b.jacobs@physics.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@physics.university.edu SIP/2.0
Via: SIP/2.0/UDP lab2.university.edu
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 Call Processing Language

While SIP CGI is an ideal tool for service creation for trusted users, it is too flexible for service creation by untrusted users, such as consumers. We have therefore developed a new scripting language, called the Call Processing Language (CPL), which untrusted users can use to define services. CPL scripts can be uploaded by users to network servers, the logic can be read in and verified, and the service instantiated *instantly*. In this section, we overview the requirements for a language that can be used in this fashion, describe its design, and discuss its primitive constructs.

5.1 Language Requirements

Because CPL scripts are generated by untrusted parties, and run on a service provider's platform, some requirements are 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 calls to external services without timeouts) cannot be present. If these constructs were present, guaranteeing completion 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.

It is also desirable for the language to be readable and producible by both humans and machines. This facilitates the use of automatic authoring tools, but still allows hand-authoring for advanced users.

Unlike SIP CGI, where the details of messages are exposed to the script, only select information and control are made available to the CPL. To support this, the language itself provides a set number of commands which allow it access to information or control of the server. These commands are SIP independent, since they define services at a sufficiently high level. This means the CPL is portable across different signaling protocols and servers.

5.2 Language design

Based on the requirements, we chose to follow the IN Service Creation Models and design a language that represents services in a *decision graph*. The individual nodes of the decision graph are the primitives of the language. They are specific decisions to be made or actions to be taken in the course of specifying the service. These decisions and actions are arranged in a directed acyclic graph (DAG), which defines the service. 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 that node. Node outputs then lead down the tree to further actions or decisions. It is possible for some or all outputs of a node to be left unspecified. This 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 3.

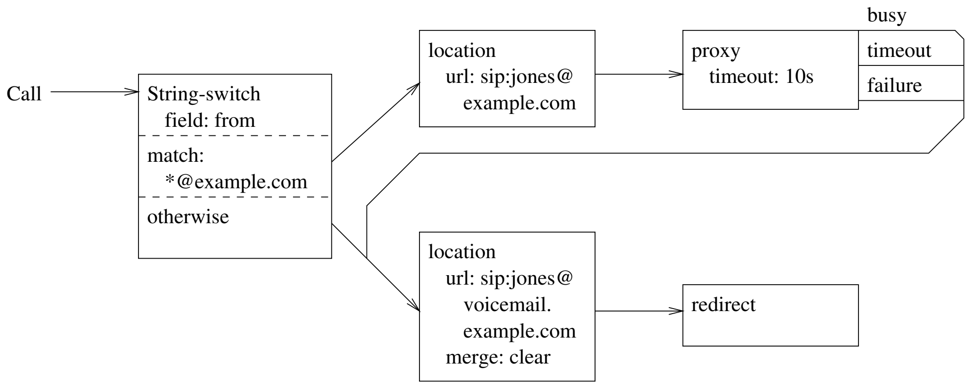

Figure 3: Example CPL Decision Graph

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.

We can also guarantee 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 a CPL can be checked by searching for cycles in the graph it represents, and computing the maximum depth of the tree. To ensure bounds on the running time, each action must be restricted in the amount of time it requires for execution. This means that actions which interface with external resources (such as database queries) must have timeouts. We have also removed generalized programming constructs, like looping, recursion and variables.

We decided to represent the decision graphs using an Extended Markup Language (XML) [16] based scripting language. XML is similar in flavor to HTML; it contains tags which describe the data in the document. We considered the usage of traditional scripting languages, like Perl [17], Tcl [18], or Python [19], portable programming languages, like Java, and application specific languages, like sieve [20], which is used for email filtering. However, we chose XML because it had several important features. XML documents are perfect for representing structured data, and in particular tree structures with optional links. This is exactly the structure needed for representing the DAGs that define call services. In addition, since XML contains no specific keywords, we were able to define a precise set of keywords representing control primitives and information accesses. XML was also useful since it allows the syntax and semantics of a script to be verified using XML validation against a Document Type Definition (DTD). XML is producible and readable by both humans and machines, satisfying another design goal of CPL.

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.

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.

5.3 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 [21].

6 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 as HTTP CGI. 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, SIP CGI scripts cannot easily provide asynchronous directives to the server. We are looking at modifications to SIP 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. Finally, means for transporting SIP CGI and the CPL must be developed, allowing users to upload these scripts to servers. Issues such as authenticity and privacy are paramount here. An initial proposal is to use SIP REGISTER messages for this purpose [22].

SIP CGI has been implemented as part of SIP server implementations at both Lucent Bell Laboratories and Columbia University; the CPL is being integrated into both implementations.

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] R. Braden, Ed., 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, Sept. 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," Request for Comments (Proposed Standard) 2543, Internet Engineering Task Force, Mar. 1999.

[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 *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] 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.

[13] I. Faynberg, L. R. Gabuzda, M. P. Kaplan, and N. J. Shah, *The Intelligent Network Standards*. New York: McGraw-Hill, 1997.

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

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

[16] 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.

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

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

[19] M. Lutz, *Programming Python*. Sebastopol, California: O’Reilly, 1996.

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

[21] 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.

[22] J. Lennox and H. Schulzrinne, “Transporting user control information in SIP REGISTER payloads,” 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 4. 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. The script is commented to explain its operation.

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

Figure 5 shows a simple script which unconditionally redirects calls to a particular address.

B.2 Call Forward Busy/No Answer

The script in Figure 6 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. The script starts by providing a location, `jones@jonespc.example.com`, which is then inherited by the proxy request. The proxy request can return either an indication of “busy” or “noanswer”. The “busy” branch provides the location of the voice mail server; the “noanswer” branch refers to it via a link. This illustrates XML’s linking possibilities, which.

```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 4: Perl script for Call Forwarding

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

Figure 5: 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 6: Call Forward Busy/No Answer Script

allows other nodes to reference a link rather than repeat parts of the script. (This is roughly equivalent to a named “goto”.)

B.2.1 Time-of-Day routing

Figure 7 illustrates a condition based on a non-call-related criterion, namely the time of day. The `time-switch` element acts like a C-language switch statement, with a set of time values and a final `otherwise` element. Since no elements follow the `proxy` element, the server simply takes the default action specified by the signaling protocol and terminates.

```
<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 7: Time of Day Routing Script