CINEMA: Columbia InterNet Extensible Multimedia Architecture

Kundan Singh, Wenyu Jiang, Jonathan Lennox, Sankaran Narayanan and Henning Schulzrinne
Department of Computer Science, Columbia University
{kns10,wenyu,lennox,sankaran,hgs}@cs.columbia.edu

Abstract

We describe the architecture and implementation of our Internet telephony system CINEMA (Columbia InterNet Extensible Multimedia Architecture, intended to replace the departmental PBX (telephone switch). It interworks with the traditional telephone networks via PSTN/IP gateways. It also serves as a corporate or campus infrastructure for existing and future services like web, email, video and streaming media. Initially intended for a few users, it will eventually replace the plain old telephones from our offices, due to the cost benefits and new services it offers. We also discuss common inter-operability problems between the PBX and the gateway. This paper is intended as a design document of the overall system.

keywords: Internet telephony deployment; VoIP test-bed; advanced IP telephony; PSTN/IP interoperability; SIP

Contents

1 Introduction

Internet telephony is defined as the transport of telephone calls over the Internet. Internet telephone calls can originate from traditional phone sets through gateways, PCs using software or embedded devices ("Ethernet phones"). Most of the interest in Internet telephony is motivated by cost savings and ease of developing and integrating new services. Internet telephony integrates a variety of services provided by the current Internet and the Public Switched Telephone Network (PSTN) infrastructure. Internet telephony employs a variety of protocols, including RTP (Real-time Transport Protocol [31]) for transport of multimedia data and SIP (Session Initiation Protocol [28, 34]) or H.323 [13] for signaling, i.e., establishing and controlling sessions.

SIP is designed to integrate with other Internet services, such as email, web, voice mail, instant messaging, multi-party conferencing and multimedia collaboration. We have implemented a SIP-based software suite called Columbia InterNet Extensible Multimedia Architecture (CINEMA) for Internet telephony and installed it within the Computer Science department at Columbia University, integrating it with the existing PBX infrastructure. The environment provides inter-operability with the PSTN, programmable Internet telephony services, IP-based voice mail, integration with web and email for unified messaging, multi-party multimedia conferencing, and inter-operability with existing multimedia tools. The setup allows us to extend our PBX capacity and eventually replace it, while keeping our existing phone numbers. The test-bed provides an environment where we can add new services and features, for example, accessing emails from a regular telephone, network appliance control, and support for instant messaging and presence. We believe that our setup can be readily used by other organizations.

Section 2 gives an overview of SIP. Section 3 details the architecture of our test-bed describing various components. Section 4 describes the user interface. Call handling by our SIP server is described in Section 5. PSTN Inter-operability is described in Section 6, whereas security issues are discussed in Section 7. Monitoring and accounting for IP telephony is important for billing purposes, and described in Section 8. Other advanced services are listed in Section 9. We describe various modules of our implementation in Section 10. Section 11 deals with some related work in this area. Finally, we summarize and point to future work in Section 12. Details of our installation are presented in Appendix A.

2 Overview of SIP

Before we look at our architecture it is helpful to know how SIP operates[1]. Readers familiar with SIP may skip this section.

For an Internet audio call, it is sufficient for a participant to know the audio codecs supported by the other participant and the IP address and port number to which audio packets should be sent. The problem with this is that IP addresses are hard to remember and may change if the user is mobile (terminal and personal mobility). SIP allows use of a more high level address of the form *user@domain* for user mobility. For instance, a user can call *bob@office.com* no matter what communication device, IP address or phone number he is using currently. The current locations of the users are maintained by the SIP location or registration servers. The user's communication devices register with registrar servers periodically by providing the address at which he/she can be reached.

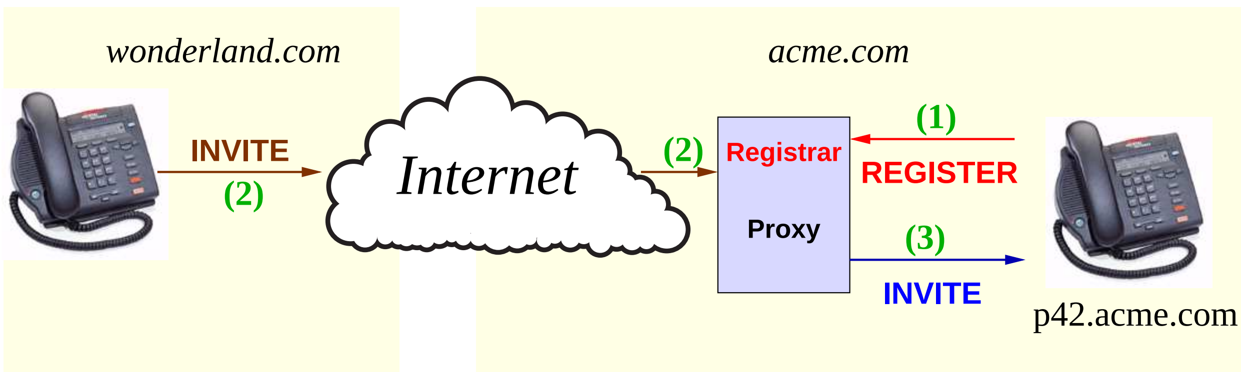


Figure 1: SIP call flow using proxy servers

Fig. 1 shows the steps involved when Alice, with address *sip:alice@wonderland.com* calls Bob, *sip:bob@acme.com*.

1. When the SIP phones are powered on they register their locations (IP addresses or host name) with the corresponding server. Thus Bob's phone tells the server at *acme.com* that the user *bob* is located at host *p42.acme.com*.

2. Since, the callee *sip:bob@acme.com* is located on the Internet, Alice's SIP phone sends the call to its "outbound" server, a SIP server that handles all calls destined for addresses outside *wonderland.com*.

3. The outbound server contacts the server *acme.com*, which knows Bob's current locations.

4. The server *acme.com* can either return Bob's location (in redirect mode) or can itself try to contact Bob at his current location (in proxy mode). Fig. 1 shows the proxy mode of operation. In the former case, the user agent retries the new location, while in the latter case the request is forwarded.

[1]More details at http://www.cs.columbia.edu/sip

It is possible to encounter multiple SIP servers (either in redirect or proxy mode) in a given call attempt. A *forking proxy* can fork the call request to more than one location, so that the first phone that is picked up gets the call, while all other phones stop ringing. Note that the outbound server is optional and may not be used in many campus environments or for all outgoing calls.

SIP calls can also use "tel" URLs that identify E.164 telephone numbers [39], for example, `tel:+12125551234`.

The list of supported audio and video codecs and the transport addresses to receive media, described using Session Description Protocol (SDP [11]), carried in SIP requests and responses.

Another point to note is that the media path for audio and video can be different from the signaling path for SIP because media can be transferred directly between the endpoints using Real-time Transport Protocol (RTP [31]) over UDP.

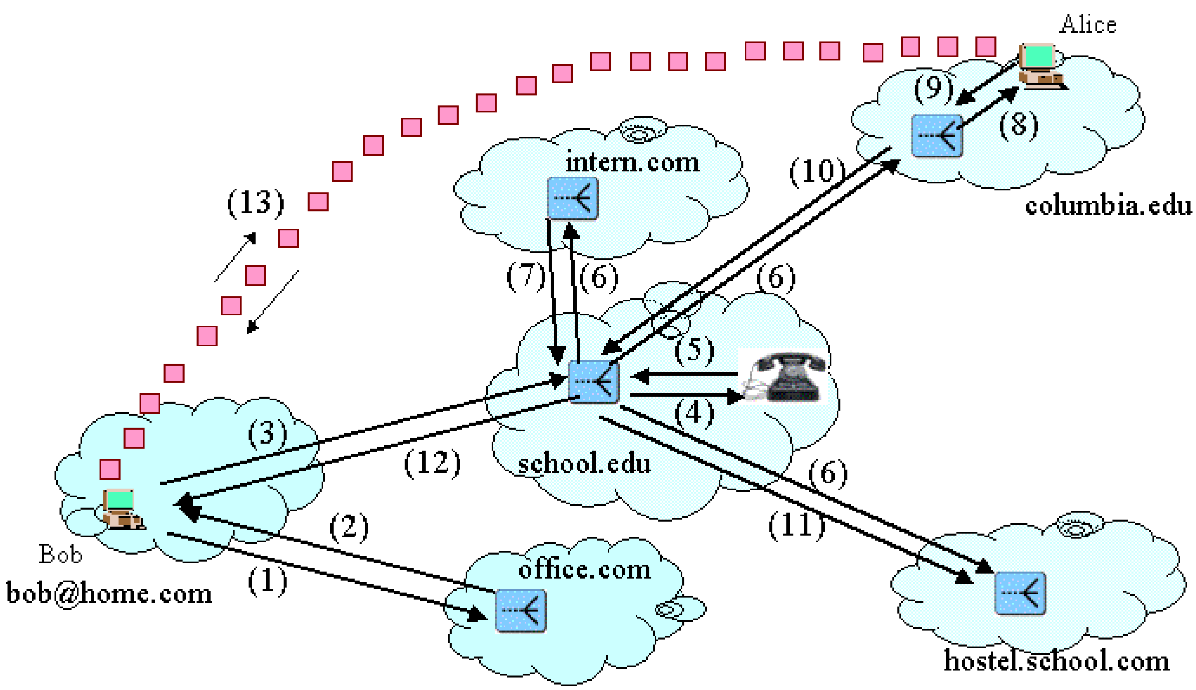


Figure 2: Example SIP call routing

Fig. 2 shows a more complex call routing scenario in SIP.

1. Bob (*bob@home.com*) tries to reach Alice (*alice@office.com*).

2. The server at *office.com* redirects Bob indicating that Alice can be reached at *alice@school.edu*.

3. Bob's user agent tries the new location.

4. Alice has registered four contacts, with one of them (her desk phone) as her preferred location. Thus, the server at *school.edu* tries the more preferred location for Alice at her desk phone.

5. The phone is idle, and sends a "ringing" response. However, since it is not picked up, the server times out.

6. The server then forks the call request to all the remaining three locations simultaneously. The locations are *Alice.Cueba@intern.com*, *alice@columbia.edu* and *ac114@hostel.school.com*.

7. The phone at *intern.com* responds back saying that the user is not available.

8. The server at *columbia.edu* forwards the call to Alice's desktop computer.

9. A popup window appears on Alice's machine indicating an incoming call from Bob. She accepts the call by clicking on the "Accept" button of the user interface.

10. The server at *columbia.edu* forwards the response to the upstream server at *school.edu*.

11. The server at *school.edu* on receiving the successful response, cancels out all the other pending call requests. In this example it cancels the call request branch sent to *hostel.school.com*. The phone at *hostel.school.com* will stop ringing at this time.

12. The server then forwards the successful response to the upstream host (Bob's user agent).

13. The call is successful. Now media (audio and/or video) can be exchanged between the two endpoints.

In the above example we have assumed a wide-area network composed of a variety of environments (campus/corporate/enterprise) running SIP servers. The rest of the paper explores the architecture and components needed for to enable Internet telephony services in such environments.

3.1 Components

The Columbia InterNet Extensible Multimedia Architecture (CINEMA) consists of a set of SIP-based servers that provide a pathway to a post-PBX era of communications. It provides a comprehensive environment for creating and deploying rich Internet multimedia services including programmable Internet telephony services, audio/video conferencing, IP-based voice mail, and unified messaging. Fig. 3 shows the architecture and the interaction among the components of our test bed.

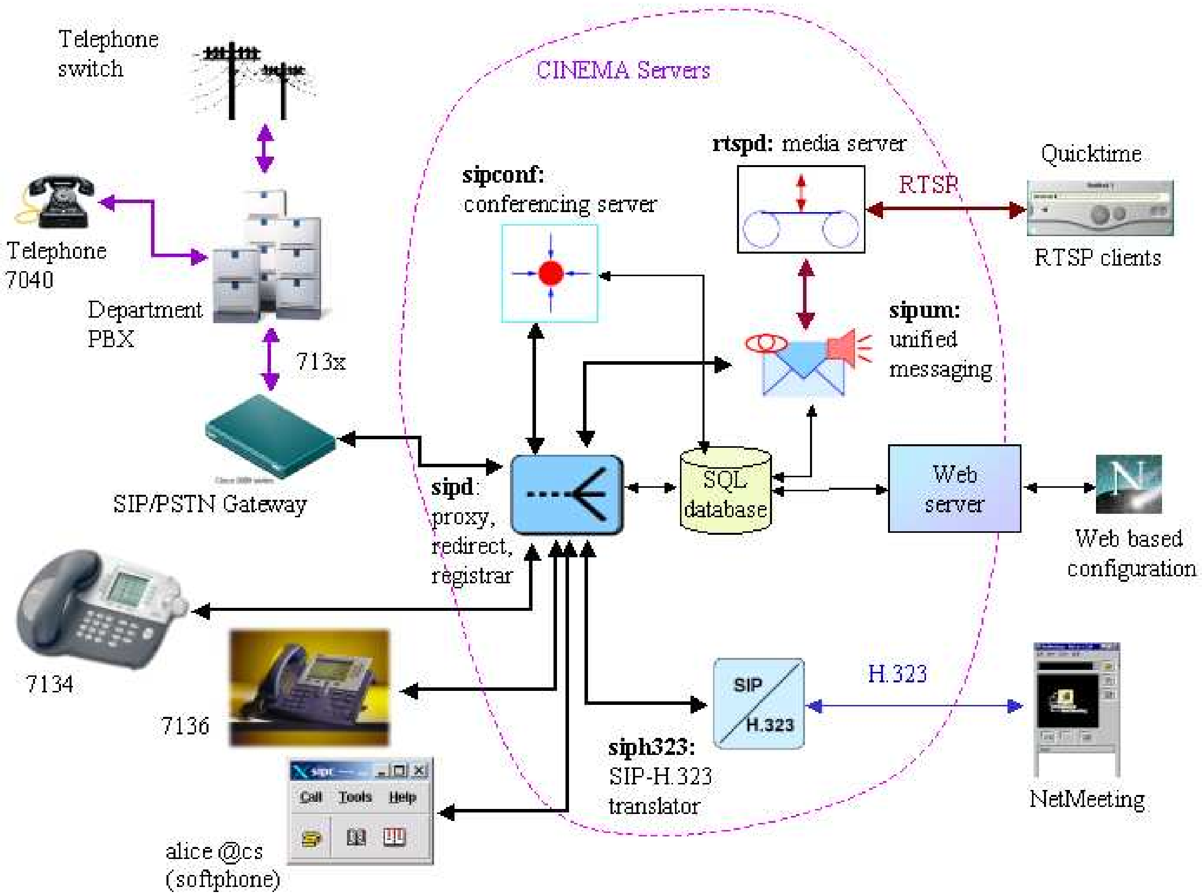


Figure 3: Architecture

SIP server: sipd is a SIP proxy, redirect and registration server. It receives user location information in SIP REGISTER messages from user agents. It also proxies/redirects the incoming calls for registered users thus acting as a call router.

SQL database: sipd uses the MySQL [20] database for storing the current network addresses and phone numbers where a user can be reached. Other per-user information related to voice mail and conferences is also stored in the database.

PSTN gateway: A Cisco 2600 router with SIP/PSTN capability is connected to the departmental telephone switch (PBX) with a T1 trunk and to the department LAN. This could be any SIP-speaking gateway.

User agents: SIP user agents (SIP UAs) allow users to interact with the system over IP. They can be either hardware (Ethernet phone) or software based. Our e*phone [14] is an example of an Ethernet phone, whereas sipc [15] is a software that runs on workstations and PCs. We also use Ethernet phones from Cisco, Pingtel and 3Com in our test bed.

Media server: rtspd is our general-purpose streaming media server, which we use for the storage and delivery of announcements and voice mail messages [38].

Unified messaging: sipum is a centralized answering machine and voice mail system [38] that uses rtspd for storing announcements and messages.

Conference server: sipconf is a centralized audio/video conference server [36].

SIP-H.323 translator: sip323 is a signaling gateway [37] between SIP and H.323. H.323 [13] is ITU-T's standard for multimedia conferencing over any packet based network. sip323 integrates popular H.323 clients such as Microsoft NetMeeting into a SIP infrastructure.

3.2 User database

The SIP server and the SQL database form the core of the CINEMA infrastructure, while the other components can be selectively enabled or disabled. For example, if an installation does not intend to use NetMeeting, it does not need sip323.

Every user of the system is given a unique identifier of the form *user@domain*, also called a canonical user identifier. Although all the local identifiers have the same domain, the domain portion in the identifier allows for unique identification and authentication. Non-local entries are also needed for handling conference members. Generally, users are assigned their email addresses as SIP identifiers. However, our system can also operate in "portal" mode, described in Section 4.3, where a new identity is created specifically for SIP calls.

User information is stored in the SQL database as the Primary User Table (PUT), indexed by user identifier. The system distinguishes between regular users and administrators, in terms of access privileges. Table 1 and 2 describes the PUT fields. Parts of the table deal with unified messaging, which we describe in Section 9.1.

The database also stores the personal address book information in the person table. This includes the full name of the user, organization, department, email address, biography and so on. Every PUT record references an entry in the person table but not every address book entry has a PUT entry.

Fig. 4 shows the realationship among various user profile tables. Fig. 5 shows interaction among different conferences and event tables. System configuration tables are shown in Fig. 6. All these database tables are summarized in tables 4, 5, 6 and 7. Among the important tables, the contacts table stores the current locations of the registered users (Table 3), which can be updated from the web page or by the SIP phones using SIP registration. It also contains the expiration time until which the location information needs to be refreshed, the preference value to sort multiple registered locations for the same user, and the action parameter to redirect or proxy an incoming call for this user. The aliases table stores aliases of all users and is used during canonicalization (see Section 5).

Field (type)	**description**
user (string)	Canonical user identifier. It could be a user identifier of the form *user@domain* (e.g., hgs@cs.columbia.edu).
hash_value (string)	MD5 hash of "user:realm:password" (the encrypted password).
realm (string)	Realm to be used for authentication. This is used as a prompt while challenging the user for authorization. We use the domain name (e.g., cs.columbia.edu).
sip_groups (string)	List of space-separated group identifiers for defining the different services to be made available for this user. Possible values are `admin`, `cgi` and `voicemail`. `admin` has administrative privileges and can access other users' accounts. `cgi` group indicates that the user can have SIP-CGI [18] scripts. `voicemail` group indicates that the user can have a voice-mail box. Since SIP-CGI should be allowed only for trusted users, it is available only to selected users.
auth (enumeration)	Type of authentication needed. Possible values are: `never`, `requires` and `request`. `never` means that the server never asks for authentication. `requires` indicates that the server should always ask for authentication and if the authentication fails the call request should be terminated. `request` indicates that the server should request authentication, but should complete the call even if the authentication fails. This feature can be used to restrict the services offerred to un-authorized users.
algorithm (string)	Algorithm to be used for authentication. Currently only possible value is `MD5`.
sip_methods (string)	List of space-separated SIP methods allowed for the user. Possible values are INVITE, REGISTER and any. For example if the user has set this to "INVITE REGISTER" then only these methods will be allowed. All others (e.g., SUBSCRIBE) will be disallowed.
remote_user (text)	List of remote user identifiers allowed to register for this user. This is needed for *third party registration*, where for instance, a secretary wants to register for her supervisor and receive calls on his behalf.
last_modified (timestamp)	Time when this record was last modified.
gwclass (string)	Gateway class the user belongs to for the purpose of making telephone calls via the gateway. For example we support gateway classes such as `faculty`, `staff`, `phd` and `student`, with different privileges for different classes in our setup. Members of the `phd` and `student` are not allowed to make long distance calls whereas `faculty` and `staff` are.

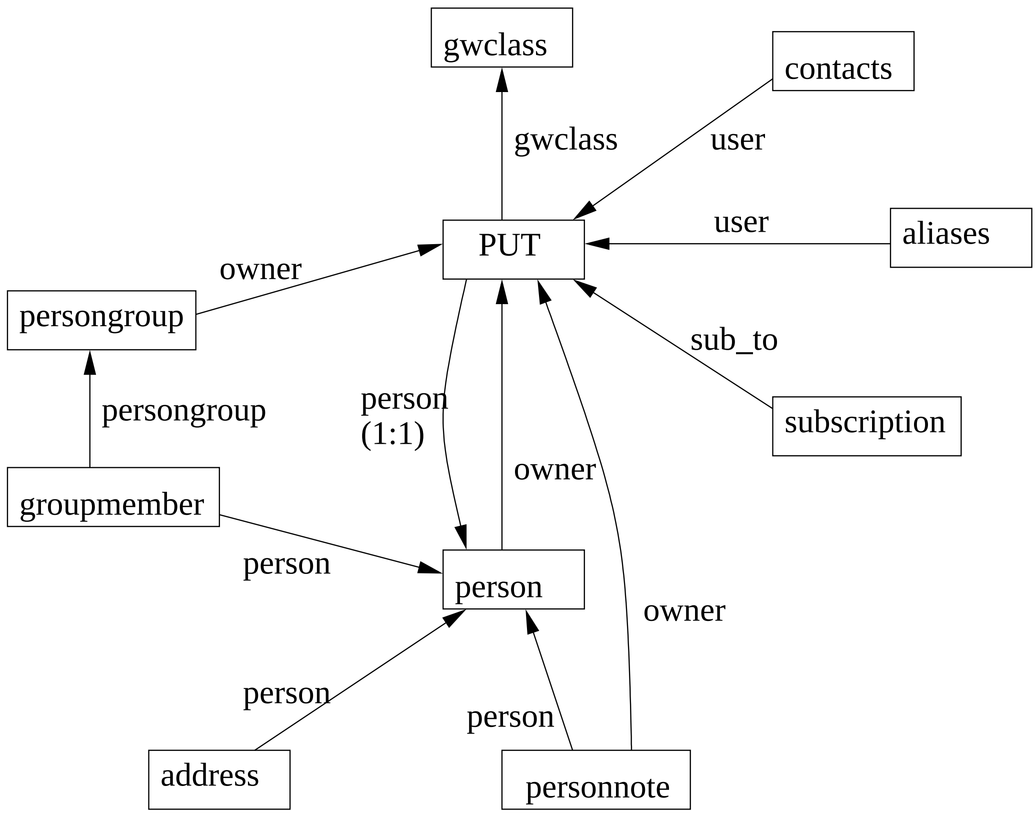
Table 1: Primary User Table (general attributes)

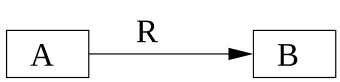
Field (type)	description
busy (string)	RTSP URL for the outgoing message prompt when the user's phone is busy, e.g., rtsp://SERVER/audio/welcome.au. The keyword SERVER is substituted by the unified messaging system based on the current media server in use.
noresponse (string)	RTSP URL for the outgoing message prompt when there was no response from the user's phone, e.g., rtsp://SERVER/audio/welcome.au. The keyword SERVER is substituted by the unified messaging system based on the current media server in use.
message_ template (text)	Message template for the email notification message to be sent when a new message arrives.
um_timeout (int)	Number of seconds to wait before forwarding the call to the voice mail.
max_msgsize_kb (int)	Maximum voice message size in kilobytes. A value of 200 indicates that the system will allow voice messages of upto maximum 200KB and beyond that the message is truncated. This is to avoid accidental recording of very large messages.

Table 2: Primary User Table (unified messaging attributes)

Field (type)	description
user (text)	Canonical user identifier. Same as Primary User Table's user field.
contact (text)	Contact URI for this user. Multiple contacts for the same user are stored in different records. Both the `user` and the `contact` fields together form the database "key" for this table. This could be any URI, e.g., "sip:kns10@muni.cs.columbia.edu" or "mailto:kundan@columbia.edu". However, our SIP server recognizes only "sip" and "tel" URI's.
expires (datetime)	Expiration time for this contact. This is stored in GMT format. Default is set as always active. This is done by setting the expires to the maximum possible value of 9999-12-31 23:59:59. After this time the contact location is expired and removed from the database on next access.
q (float)	Preference value for this contact. This is needed when there are multiple contact locations for this user. Possible value ranges from 0.0 (least preferred) to 1.0 (most preferred).
action (enumeration)	Preferred action to be taken by the SIP server for this registered contact. Possible values are `Proxy` and `Redirect`.
last_modified (timestamp)	Time when this record was last modified.
display_name (string)	Optional display name for the contact URI.
sip_methods (string)	List of space-separated SIP methods possible for this contact location. For example, some contact locations (e.g., "mailto:kns10@columbia.edu") may not allow INVITE.

Table 3: Contacts table





For every A there is one unique associated B by relation R.
There could but multiple B by a different relation.
Relation R indicates a field name in table A.
Multiple A can map to same B unless noted by 1:1 relation.

Figure 4: User profile tables

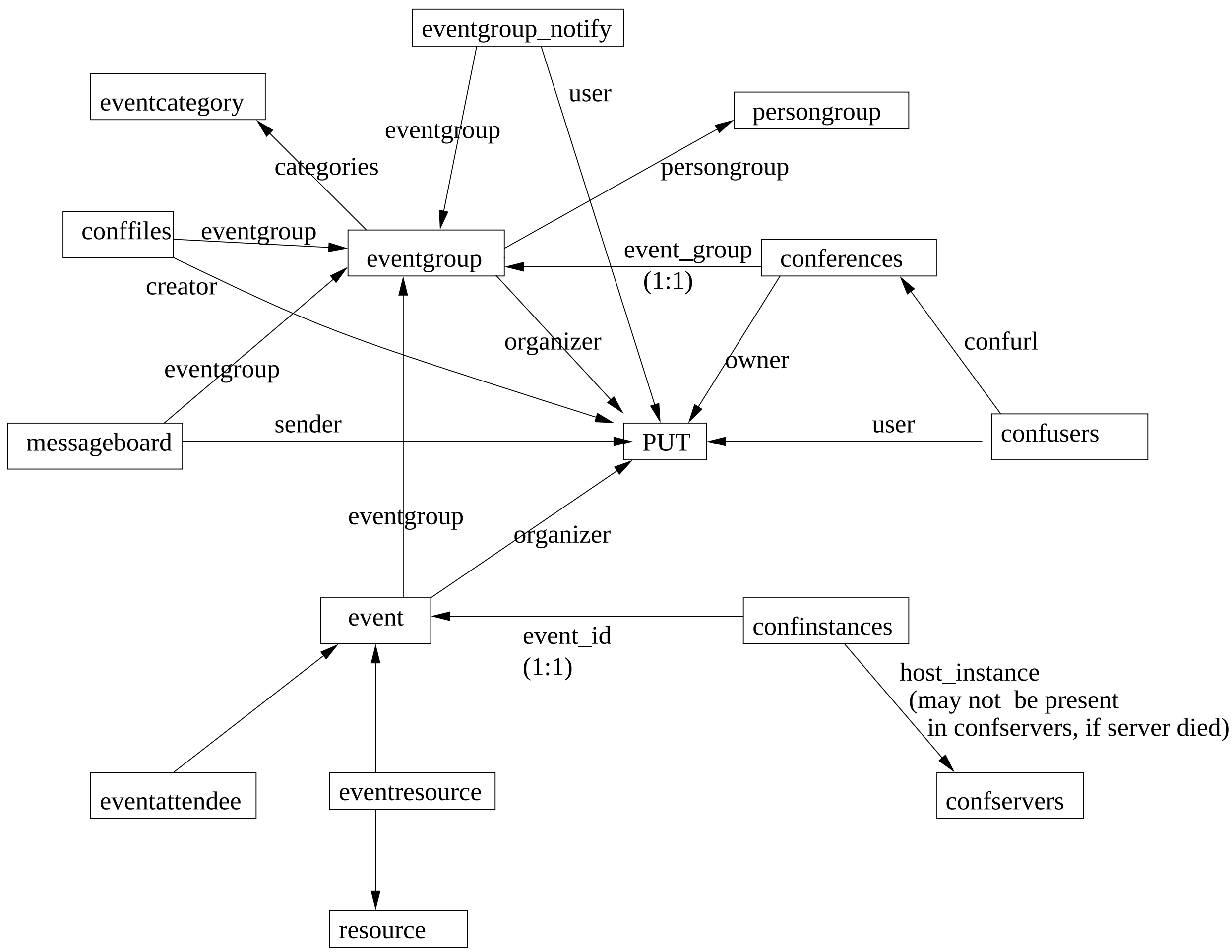


Figure 5: Conferences and event tables

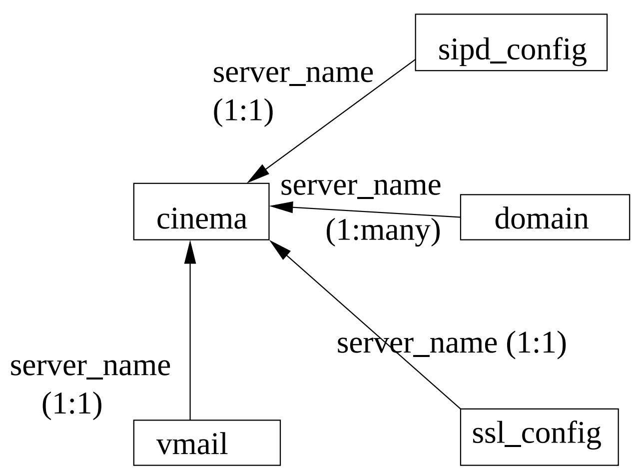


Figure 6: Configuration tables

Table name	Description
cinema	Global system configuration information, e.g., administrator, web server host, default realm, portal or local mode.
sipd_config	System configuration information for SIP proxy/registrar server (sipd), e.g., type of authentication, default mode a proxy or redirect, default expiry for registrations..
domain	Acceptable domains for this system. sipd will accept registrations for these domains. Other domains are considered foreign domain and hence could be proxied or redirected. In addition, calls whose host part of the URI matches one of the entries in the domain table, will be handled locally while others will be proxied to the particular domain's SIP server.
ssl_config	TLS [8] configuration for establishing secure connections (described in Section 7.2).
vmail	System Configuration information for the unified messaging system and media server.
license	Software licensing information for various components.
requestlog	Call logging configuration (described in Section 8.1).

Table 4: System configuration tables

Table name	Description
put	User profile for SIP call routing, e.g., authentication, voicemail. (Table 1 and 2)
aliases	User aliases for call routing. These are alternate identifiers for the user. Canonicalization using aliases is described in Section 5
contacts	Contact locations where a user can be reached, e.g., the current SIP URL of the phone the user is using.
subscription	Subscription information maintained by the presence agent (part of sipd). Presence agent is described in Section 9.3
person	User profile information, e.g., name, organization, email, web page address, etc. Every entry in put has an entry in this table. However, system users can create multiple person entries in their address book.
personnote	Note or comment about persons maintained on a per user basis.
address	Mailing addresses and phone numbers for a person.
persongroup	Different groups for the purpose of access control.
groupmember	Associates a person to a persongroup.

Table 5: User profile and locations

Table name	Description
gwclass	User classes for different call privileges, e.g., full access, guest access; or student, faculty and staff in an university environment.
tariff	Billing related details for various call prefixes (local, long distance) for various user classes (faculty, students, staff). We describe accounting and billing in Section 8.1.3.
dialplan	Dialplan mapping. This is used for canonicalizing telephone numbers to a globally routable number, described in Section 5.
gateway_map	Locations of the telephony gateways and permissions for various telephone number prefixes, described in Section 6.

Table 6: PSTN interworking tables

Table name	Description
conferences	List of all scheduled conferences. A conference is identified using a name, e.g., sip-forum. This is used by the conferencing server.
eventgroup	A group of recurring events. For instance, a weekly seminar or twice a week class. Every conference belongs to an eventgroup. But an eventgroup may not have an associated conference.
event	An instance of an event, e.g., a seminar, a talk, or a conference instance.
eventattendee	Participants in an event.
eventcategory	Type of event: meeting, class, etc.
eventgroup _notify	Notification and announcement related subscription for an event or an eventgroup. Users can register to get notified about an event.
resource	Various resources like room, projector.
eventresource	This associates an event with the resources it needs.
confinstances	A conference instance is an instantiation of a conference. For example a conference that regularly repeats every week has one record in confinstances table every week. The separation of confinstances from conferences allow storing per instance information, e.g., conference recording. A conference instance is associated with an event.
confservers	Available servers used for conference load balancing. Multiple conference servers work together to provide high quality conferencing by balancing load among themselves. A given conference is always hosted by one server. But a new conference can be redirected to a less loaded server.
confusers	Details regarding participants in a conference such as names, media access privileges etc. For example, a conference can allow all users *@cs.columbia.edu* to send and receive audio where as all other users to only receive audio.
messageboard	Message board for sharing offline messages in an eventgroup.
conffiles	Shared files in an eventgroup for offline viewing.

Table 7: Conferences and events tables

4 User Interface

We have implemented a web-based user interface to configure and manage the system. Various user and system profiles can be configured from a web browser. Both the per-user information and system configuration tables can be manipulated from the web interface. There are two classes of users: normal users (referred to as the user class) and administrators (referred to as the administrator class). Administrators have additional privileges compared to regular users. The first user created during installation becomes an administrator. This user can then add additional users as administrators later. A user cannot access profiles of other users or change the status of himself or other users.

Users can login from the web page by providing their `userid` and `password`. The `userid` is a unique identifier of the form *u*ser@domain and identifies a given user.

We use HTTP CGI [10] for implementing the web interface. The CGI scripts are written in Tool Command Language (Tcl).

4.1 New user

A new user is created from the web interface by specifying a valid email address. In "portal mode", this requirement is relaxed (see section 4.3). Only those users with name in the list of local domain can receive SIP calls; other users can use other services like conferencing. Email notification is sent to the `userid` indicating the initial password. The initial password is randomly choosen by concatenating three random words from a dictionary. A new record for this user is created in the SQL database. The other user profile parameters (e.g., `gwclass`, `realm`) are taken from the default user for the particular domain. For example if the userid is kns10@cs.columbia.edu then the user profile parameters are borrowed from that of default@cs.columbia.edu. This default user profile can be altered by the administrator. Users can later change their profile information including the password.

4.2 User login

Users can log in by specifying the `userid` and `password`. The record from the SQL database is fetched, then the `hash_value` is computed using the `userid`, `realm` and `password` and is compared against the database's `hash_value`. Authenticated users are provided with various options: editing user profile, viewing voice messages, updating contacts and aliases information and so on. An example web page is shown in Fig. 7. The `userid` and `password` is carried in HTTP cookies. This means a user does not have to enter these values every time he visits the web page, unless he explicitly logs out.

4.3 Portal mode

Our system can also be installed in a "portal mode" for providing services to users. This is done by using an `userid` which need not be a real email address. The user has to provide an email address for notification purposes. For example if the system is installed in "portal" mode at domain "example.com" then all the registered users will have their `userid`as "user@example.com" but can use their regular email addresses for notification purposes. In non-portal mode the `userid` must be same as that of the email address. The non-portal mode is intended to be used within an organization whereas the portal mode can be used by an Internet telephony service provider.

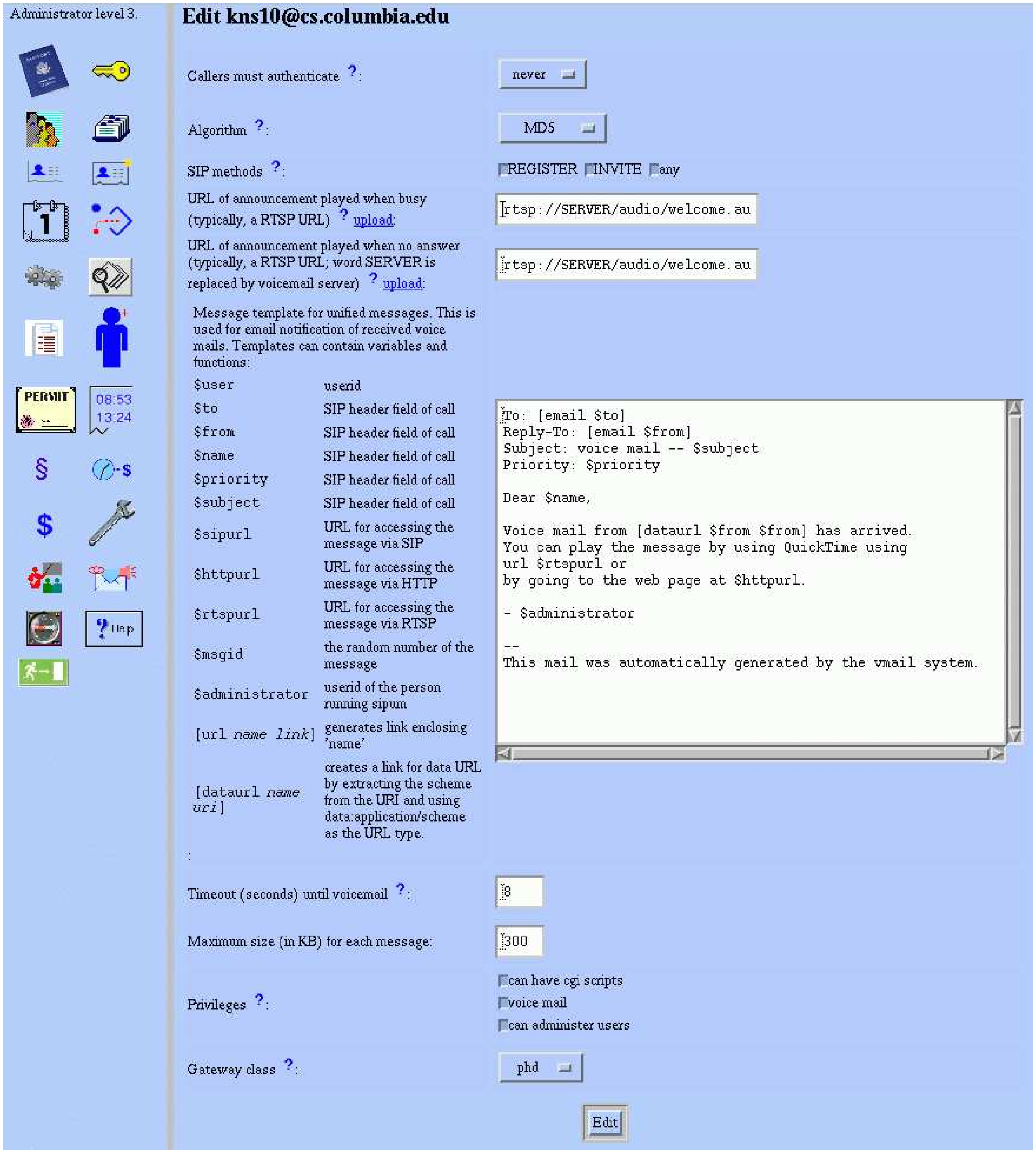


Figure 7: Example web page

4.4 Address book and calendar

The web interface allows users to maintain personal address books and event calendars. This is useful for scheduling meetings and keeping track of reminders. The system can be configured to send notifications, using SIP NOFITY mechanism [21], a regular email, or a SIP call when an event is about to happen.

4.5 Billing and accounting

The web interface allows an administrator to see the log of all the calls made. It also permits him to set tariffs for different types of calls. This is described in detail in Section 8.1.3.

This section describes how sipd handles an incoming call.

5.1 Canonicalization

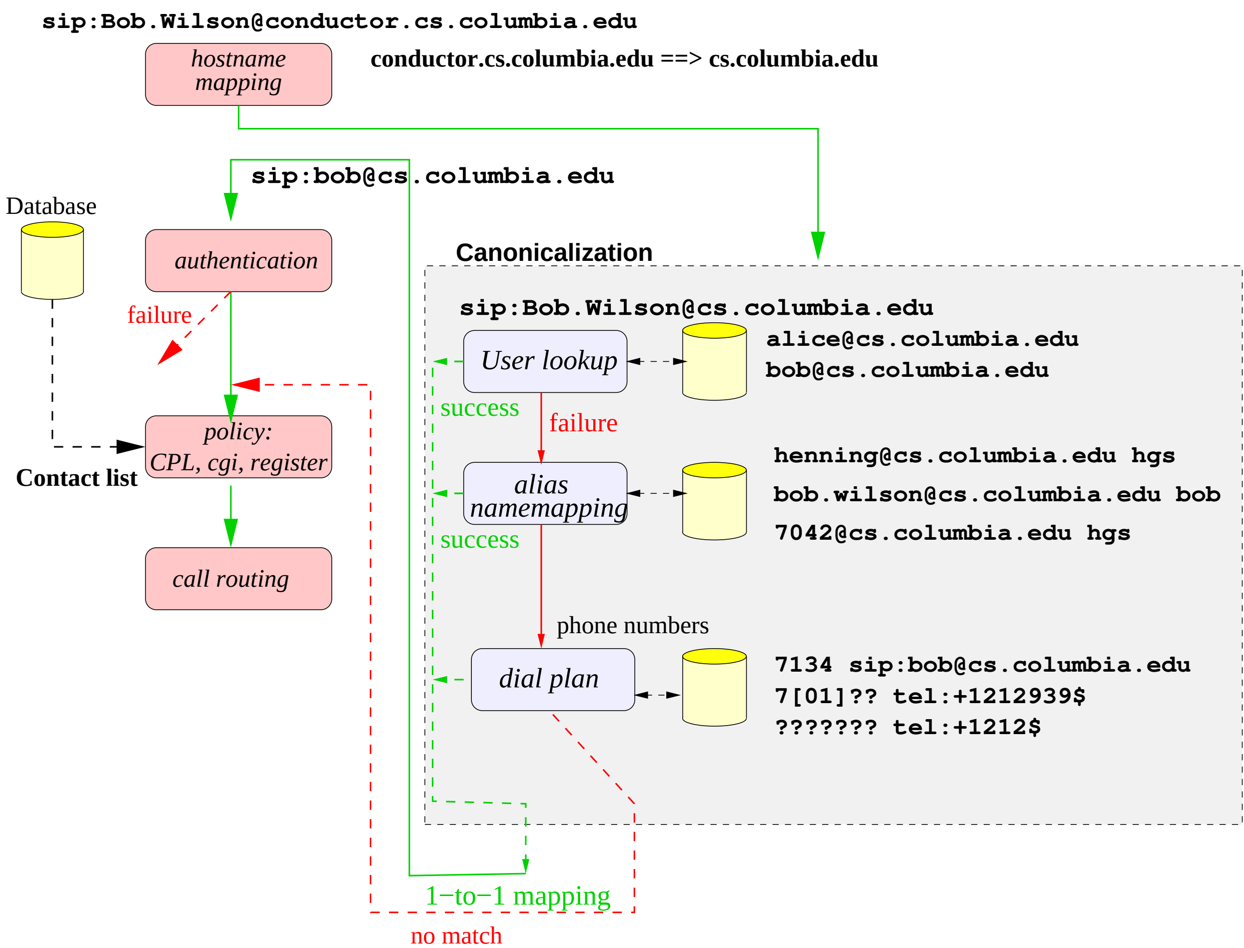


Figure 8: Canonicalization, authentication and routing for a call

An incoming call is processed as shown in Fig. 8. Here, Alice, sip:alice@cs.columbia.edu calls Bob, sip:Bob.Wilson@cs.columbia .edu. Through DNS SRV records, Alice's user agent finds out that the host `conductor.cs.columbia.edu` serves SIP requests for the domain `cs.columbia.edu`. We assume that Bob can be reached in many different ways, for example, as `bob`, `Bob.Wilson`, `bob_wilson`, `Bob.V.Wilson` and `webmaster`.

After validating the syntax of the call request, the server transforms the callee address to a canonical user identifier for database lookup, by first transforming the host portion and then the user name portion. For example, the domain portion, `conductor.cs.columbia.edu` is canonicalized to `cs.columbia.edu`. This is done by matching the domain portion of the request URI against a list of possible domain names and IP addresses for SIP requests to this proxy server. In our case, this includes the domain name `cs.columbia.edu` and the host name and IP address on which sipd is running. If the canonicalized host name does not match, the server is being used as an "outbound proxy server" and just routes the request to the SIP server for the domain, without any processing. Outbound proxy servers are useful for logging and firewall control, for example. Outbound proxies are not needed for "sip" URLs, but SIP requests with "tel" URLs need to designate such a proxy to translate the telephone number into a routable SIP identifier. This SIP identifier can either point to a PSTN gateway or be a regular *sip:user@host* URL.

The server first checks whether the SIP identifier is present in SQL put table. If it is present, then the `username` is used unchanged and is the canonical user identifier. If it is not present, then the server tries to translate the username into a canonical form by two transformations. In the first, the SQL aliases table (described in Section 3.2) is checked to see whether an alias entry is present for the user. If an alias is present, it is resolved to its canonical identifier user. In the second step, the name mapper function searches the SQL person table to see if it can deduce a username, by comparing the user part of request URI to various combinations of the first, last, and middle names recorded in that table. (In the example, the name mapper determines from the person table that the name "Bob Wilson" corresponds to a non-NULL PUT entry for the user bob.)

Finally, the server checks whether the user identifier is a telephone number or not. A request URI for telephone numbers can be of the form:

1. tel:*number*
2. sip:*number*@*domain*;user=phone
3. sip:*number*@*domain*

Note that for the "tel" URL, the domain portion does not exist hence there is no need to canonicalize the domain part. The *number* can have an optional prefix of "+" to indicate a globally routable number, e.g., `+1-212-9397000`. The first and second cases specifically tell the server that the address is a telephone subscriber. A heuristic is used to determine if the address matches the third case. A database lookup is done to compare the address against the available user names and aliases to find a match. This allows to create telephone number as user identifier or to create telephone number aliases for *user@domain*. If the resulting address is still a telephone number, it is canonicalized using a dial plan, described in Section 6.1. If none of the rules match, the user identifier is returned unchanged to the server.

The SIP server then retrieves contact and policy information for the user bob@cs.columbia.edu. The policy information describes how the call is handled, for example whether it is to be proxied or redirected. Bob's preferences and policy are then executed. These may, for example, demand that a calling user be authenticated, refuse or redirect calls, or apply preferences about where Bob wants to be reached. If the server determines that Bob's current policy allows Alice's call to reach him, it contacts Bob's list of registered locations. Bob's current SIP phones ring, he picks up the handset and starts talking to Alice. When they are done, either of them can terminate the call.

If the callee's contact location is a telephone number, then the dialplan matching is done on the contact location as described in Section 6.1. The dialplan leads to a gateway to reach the PSTN destination.

If there are multiple contacts found for the user, then all of the contact locations are used. The preference values (q-value) of the contacts are used to order the contact locations. The more preferred value is tried first, and if it fails or times out, the next preferred location is used. If multiple contacts have the same or similar q values, then the server forks the call request to all those locations in proxy mode. In redirect mode, it returns all those contact information back to the caller. For example, if user *sales@company.com*, has locations *rep1@pc1.company.com* (preference 1.0), *rep2@pc2.company.com* (preference 1.0), *rep3@pc3.company.com* (preference 0.8), *senior-rep@company.com* (preference 0.3) and *manager@company.com* (preference 0.3) then a call to *sip:sales@company.com* is first forwarded to both rep1 and rep2. If they do not pick up the phone or the call fails, then rep3 is tried. If rep3 also does not answer the call, then it is forwarded to senior-rep and manager simultaneouly. The forking behavior with the configurable priorities for different contact locations can achieve enhanced automatic call distribution (ACD).

Unlike PSTN switches, SIP servers normally do not store any call state, that is, state that needs to be maintained until the call completes. They are responsible only for forwarding signaling requests and responses such as call initiation and termination messages, and other messages like instant messages. While the call initiation message goes through the SIP server, the call termination message may be directly exchanged by the two user agents without any SIP server. However, the server can insist on being in the call path for subsequent messages using the SIP Record-Route header [12]. This is useful for call logging and accounting.

5.2 Programmable call handling

When receiving an incoming call request, the SIP server finds the current user location and either proxies, redirects or rejects the call initiation message. Although this simple model satisfies most user's needs, some advanced users may want a more complex scenario. For example, "reach me at my office phone during office hours and call me at my home after office hours, or don't disturb me when a tele-marketer calls." This can be implemented by uploading a piece of software on the server, which governs its behavior based on the time-of-day or caller identification. SIP allows many different ways to achieve this, for example, via the XML-based Call Processing Language (CPL [26, 17]) and the SIP Common Gateway Interface [18]. Our SIP server supports both CPL and SIP-CGI. SIP Servlet [16, 1] implementation is in progress. The piece of software which alters the server behavior, either a SIP-CGI or a CPL script, can be uploaded to the server using a SIP UA such as sipc, or from the web interface.

5.2.1 CGI: Common Gateway Interface

SIP-CGI is similar to HTTP-CGI. SIP-CGI scripts can be written in any language. It has the same potential security problem as HTTP-CGI, and it should be allowed only in a trusted environment since users are allowed to execute arbitrary code. The SIP server runs the script as an external process, passes all the parameters needed by the script (e.g., caller URI, subject headers, etc.) and gets back the response from the standard output of the process. The response indicates how to handle the call, for example, proxy it, redirect or reject.

Fig. 9 shows an example of a SIP-CGI script written in Perl that increases the priority of the call made by userid hgs.

5.2.2 CPL: Call Processing Language

The Call Processing Language (CPL) is a language that can describe and control Internet telephony services. It is implementable on either network servers or user agent servers. It is simple, extensible, easily editable by graphical clients, and independent of operating system or signalling protocol. It is suitable for running on a server where users may not be allowed to execute arbitrary programs, as it has no variables, loops, or ability to run external programs.

An example of a service is a calendar-based call routing system. Calendaring and scheduling information formatted as iCal [7] is combined with a policy file and then converted into a CPL script, which is uploaded to the server. The policy expresses rules such as "if Joe calls and I'm busy (according to my calendar), forward to secretary". Services can also be driven by caller preferences [35], where the caller indicates desired call routing and handling behavior. For example, a caller may request that calls are not forked or that calls are not routed to voicemail or attendants.

Fig. 10 shows an example CPL script for time-of-day based call routing. The idea is to proxy the call to the registered location during office hours and forward the call to voicemail otherwise.

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

# Prioritize messages whose 'From:' matches 'sip:hgs@'
# by proxying them with 'Priority: urgent'.

# Translate the REGISTRATIONS env variable into a list
# of registration addresses, without name-addr forms
# or parameters.

sub get_regs {
# my($reg_str, @regs);

    if (!defined($ENV{REGISTRATIONS})) { return (); }

    $reg_str = $ENV{REGISTRATIONS};

    # Kill quoted strings.
    # A quoted string consists of a pair of quotes, surrounding the following:
    # 1. any character other than a backslash or a quote
    # 2. a backslash preceeding any character
    # This handles all "some number of backslashes before the trailing quote"
    # problems.
    $reg_str =~ s/\"([^\"\\]|\\.)*\"//g;

    # Now, all commas are syntaxtically significant, so it is safe to split
    # on them.
    @regs = split(",", $reg_str);

    grep {
        # Eliminate parameters, then strip <> forms.
        s/;.*//;
        if (/\<(.*)\>/) { $_ = $1; }
    } @regs;

    return @regs;
}


if (defined $ENV{SIP_FROM} && $ENV{SIP_FROM} =~ /sip:hgs@/) {
    foreach $reg (get_regs()) {
        print "CGI-PROXY-REQUEST $reg SIP/2.0\n";
        print "Priority: urgent\n\n";
    }
}
```

Figure 9: SIP-CGI example: call priority

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

<cpl>
  <incoming>
    <time-switch tzid="America/New_York"
        tzurl="http://zones.example.com/tz/America/New_York">
      <time dtstart="20000703T090000" duration="PT8H"
            freq="weekly" byday="MO,TU,WE,TH,FR">
        <lookup source="registration">
          <success>
            <proxy />
          </success>
        </lookup>
      </time>
      <otherwise>
        <location url="sip:jones@voicemail.example.com">
          <proxy />
        </location>
      </otherwise>
    </time-switch>
  </incoming>
</cpl>
```

Figure 10: Example CPL script: time-of-day routing

6 PSTN Inter-operation

6.1 PSTN-to-IP call

PSTN subscribers are identified by telephone numbers rather than SIP URLs, email or IP addresses. A PSTN user can reach the gateway by dialing any of the extensions assigned to the gateway's T1 line. For example, our PBX has assigned extensions in the range 7130-7139 to the gateway. So anybody who dials (212) 939-7134 reaches the gateway at extension 7134. An example of a PSTN-to-IP call is shown in Fig. 11.

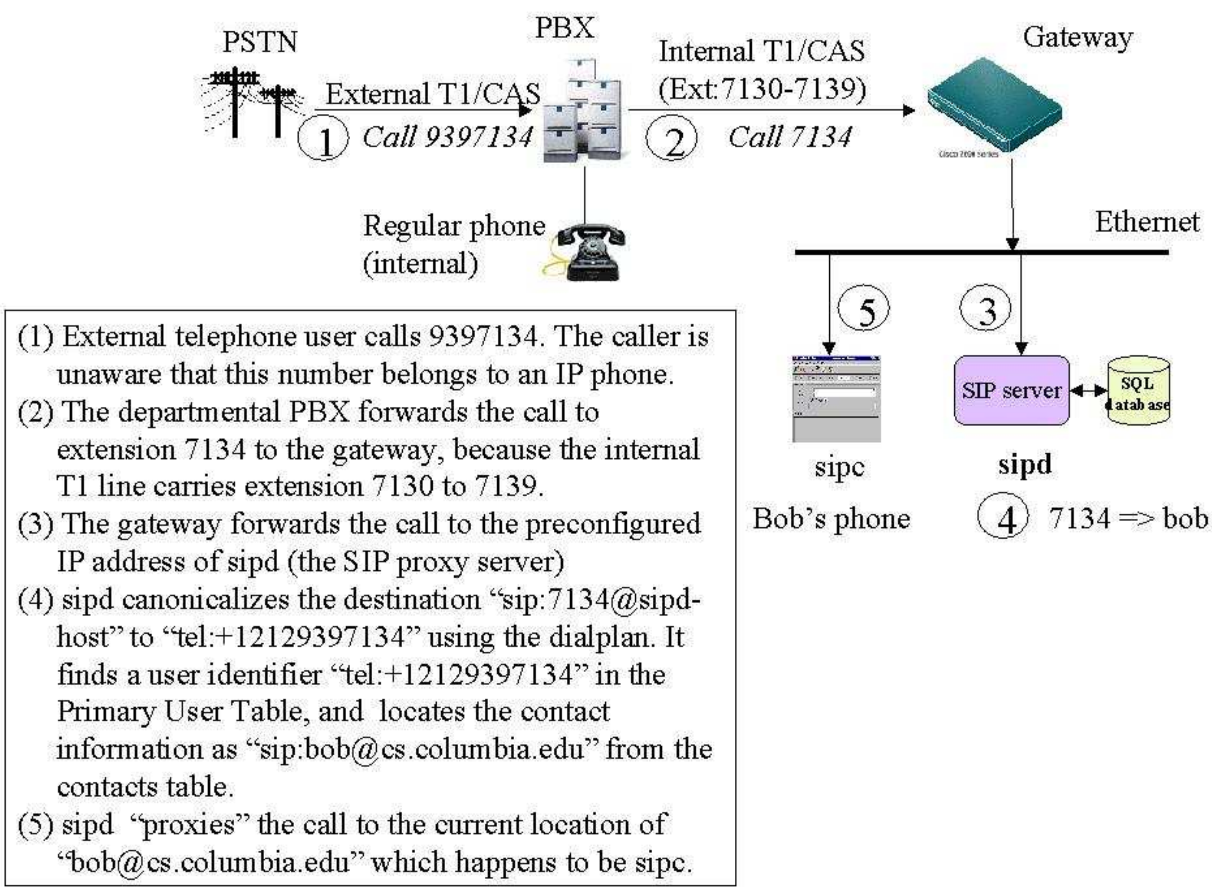


Figure 11: A PSTN-to-IP call

On the gateway, we need to define a voice-over-IP call-leg specifier (called a dial peer). An example Cisco configuration where the SIP server's IP address is 128.59.19.62[2], is as follows:

```
dial-peer voice 1 voip
 preference 1
```

[2]The IP addresses and net masks are not necessarily real.

```
destination-pattern 713[0-9]
voice-class codec 1
session protocol sipv2
session target ipv4:128.59.19.62
```

The following example dial peer specifies 7-digit POTS (Plain Old Telephone Service) local calls from SIP to PSTN:

```
dial-peer voice 1005 pots
 preference 6
 destination-pattern 8.......
 no digit-strip
 port 1/0:1
```

A "." is a wildcard for any digit, and "8" is the prefix the user must dial to reach a number outside our PBX. The "preference" parameter is used to match dial peers in a certain order: Lower value means higher preference. If not listed, the default preference is highest with value 0.

When a call comes in from the PSTN, the gateway can react in one of the two modes, direct-inward-dialing (DID) or no-DID. In DID mode, the incoming trunk delivers the destination extension to the PBX or gateway. So a call to 7134 is forwarded to the SIP server as sip:7134@128.59.19.62. The SIP server maintains a mapping between the telephone number and the user identifier. The mapping is called a dialplan. For example, 7134 can be mapped to sip:bob@cs.columbia.edu so that the above call reaches Bob at his SIP phone (see Section 5). For the 713x range, the DID mode can support only up to 10 users. In the no-DID mode, the gateway will prompt the caller with a second dial tone. After the caller dials a new extension, it is captured and forwarded to the SIP server. The differences between the two modes are summarized below.

Mode	usage	advantages
DID	dial directly	simpler dialing from PSTN
No-DID	dial extension	supports more users

The SIP server also supports ENUM [9] for translating a telephone number to an URI. So if the call is made to 7134, as in the previous example, and ENUM is used then the server tries to do DNS lookup for the hostname 4.3.1.7.9.3.9.2.1.2.1.enumdomain. This lookup can yield a "tel" or "sip" URI. The enumdomain is configurable.

6.2 IP-to-PSTN call

In the reverse direction, when a SIP user dials a telephone number, e.g., sip:9397040@cs.columbia.edu, the SIP server transforms the telephone number to the telephone subscriber tel:+12129397040. Also, as is typical for PBXs, the same number can be dialed in a number of different ways, for example, as a four-digit extension (7042), as a local phone number (939-7042) or as a global number (1-212-939-7042), with country code. In addition, PBXs often designate a digit such as 8, 9 or 0 to reach an outside line. Thus, for IP telephones, which often follow the mobile phone model of requiring an explicit indication of the end of a phone number, a large number of variations need to be unified into a single global number which can then be used to determine the appropriate gateway. An example of an IP-to-PSTN call is shown in Fig. 12.

This model is reflected in the following sample dialplan used for our server, where both a 4-digit extension and a 7-digit local number are mapped to a canonical format with country code, area code, and local number. The symbol "$" is substituted by the matched string on the left column, while "?" matches a single digit and "*" matches any digit string. Visual separators like ".", "–", "(" and ")" in the telephone number are ignored on the left hand side of the pattern. When the

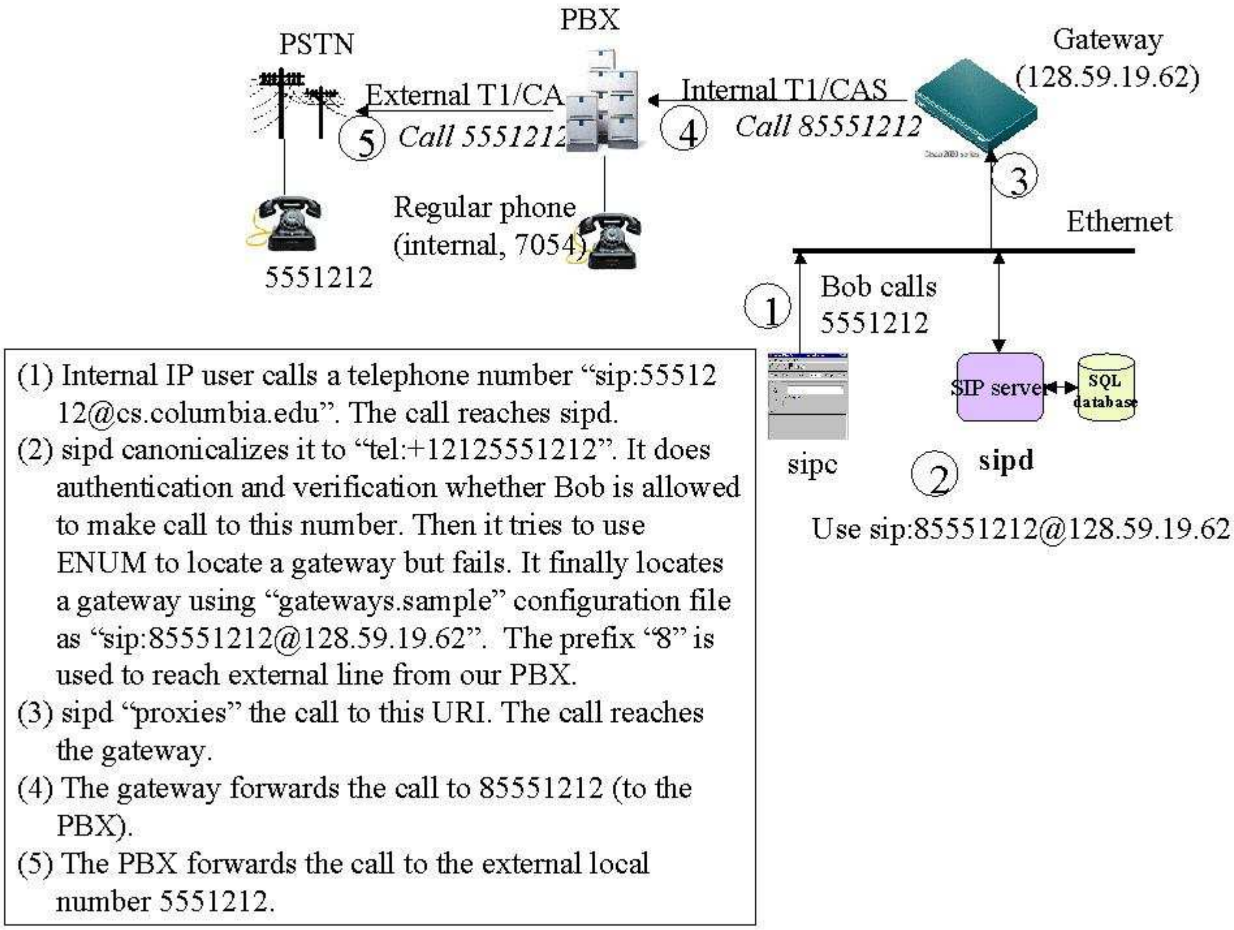


Figure 12: An IP-to-PSTN call

first match is found, everything which matched outside the parenthesis is used for substituting "$" on the right column. In example dialplan of Fig. 13 both "9397040" and "89397040" are translated to "tel:+12129397040". If more than one rows match, then the row with higher priority value is used.

Local numbers	Mapped to	Priority
7[01]??	tel:+1212939$	50000
???	tel:$	49000
[134]????	tel:+121285$	48000
???????	tel:+1212$	47000
(8)???????	tel:+1212$	46000
(81){800,888,877,866,900,700}???????	tel:$	45000
(1)??????????	tel:+1$	44000
(81)??????????	tel:+1$	43000
(011)*	tel:+$	42000
(8011)*	tel:+$	41000
/^81010288(.*)$/	tel:$2;tsp=mci.com	40000
/^81010233(.*)$/	tel:$2;tsp=sprint.com	39000
/^[0-9]*$/	tel:$0	38355

Figure 13: An example dialplan

The server then locates the appropriate gateway to route the call to the PSTN. For an orga-

Class	Pattern	Priority	Gateway
faculty	(+1212939)7[01]??	50000	sip:$@itgw1.cs.columbia.edu
faculty	(+121285)4????	45000	sip:$@itgw1.cs.columbia.edu
faculty	(+1212)???????	40000	sip:8$@itgw1.cs.columbia.edu
faculty	(+1){800,888,877,866}???????	35434	sip:81$@itgw1.cs.columbia.edu
faculty	(+1){347,646,718,917}???????	30000	sip:81$@itgw1.cs.columbia.edu
faculty	(+1)??????????	25000	sip:81$@itgw1.cs.columbia.edu
faculty	(+)*	20000	sip:8011$@itgw1.cs.columbia.edu
full	(+1212939)7[01]??	50000	sip:$@itgw1.cs.columbia.edu
guest	(+1212939)7[01]??	50000	sip:$@itgw1.cs.columbia.edu
phd	(+1212939)7[01]??	50000	sip:$@itgw1.cs.columbia.edu
phd	(+121285)4????	45000	sip:$@itgw1.cs.columbia.edu
phd	(+1212)???????	40000	sip:8$@itgw1.cs.columbia.edu
phd	(+1){800,888,877,866}???????	35000	sip:81$@itgw1.cs.columbia.edu
phd	(+1){347,646,718,917}???????	30000	sip:81$@itgw1.cs.columbia.edu
student	(+1212939)7[01]??	50000	sip:$@itgw1.cs.columbia.edu
student	(+121285)4????	45000	sip:$@itgw1.cs.columbia.edu
student	(+1){800,888,877,866}???????	35000	sip:81$@itgw1.cs.columbia.edu

Figure 14: An example gateway-map

nization with a small number of gateways, a static table, as currently used in sipd, is sufficient. If networks of IP telephony gateways are deployed, more complex routing protocols such as TRIP [27] may become essential. TRIP allows to route the call to the optimal gateway, e.g., the one closest to the destination. Here the meaning of closeness may be defined in terms of geographical proximity, QoS parameters, or per-call cost.

In our system, each local user is assigned a "gateway class", such as faculty, phd or student as shown in Fig. 14. In this example, faculty can make calls to any long distance number but student is allowed only intra-department or toll-free numbers through the gateway `itgw1.cs.columbia.edu`. No gateway is choosen if the user class and destination pattern do not match any row in the gateway map. In which case, the server may terminate the call as the caller does not have sufficient privileges. If more than one rows match, then the row with higher priority value is used.

Note that we need special provisioning in the SIP server so that the telephone number as dialed by the caller is billed to the caller, whereas the telephone number present as a contact location for a user (callee) is billed to the callee. This means the *gwclass* authorization is done for the caller in the former case and for the callee in the latter case.

6.3 Connecting to the PBX

PBX (Private Branch eXchange) is used in many corporations and universities. It centralizes telephone management, consoliates external trunk lines and voice mail. Our PBX is a Nortel Meridian Option 11C. It has an external T1 line to the public telephone network, capable of 24 incoming/outgoing calls. It also has an internal T1 line to connect with the PSTN/IP gateway. With this topology, a user can make IP telephone calls from either an analog phone (whether inside or outside the department) or a SIP UA. Fig. **??** shows our installation with two SIP-PSTN gateways, one to the department PBX and the other to the University telephone switch.

During deployment, we encountered quite a few problems that are worth describing. Many of

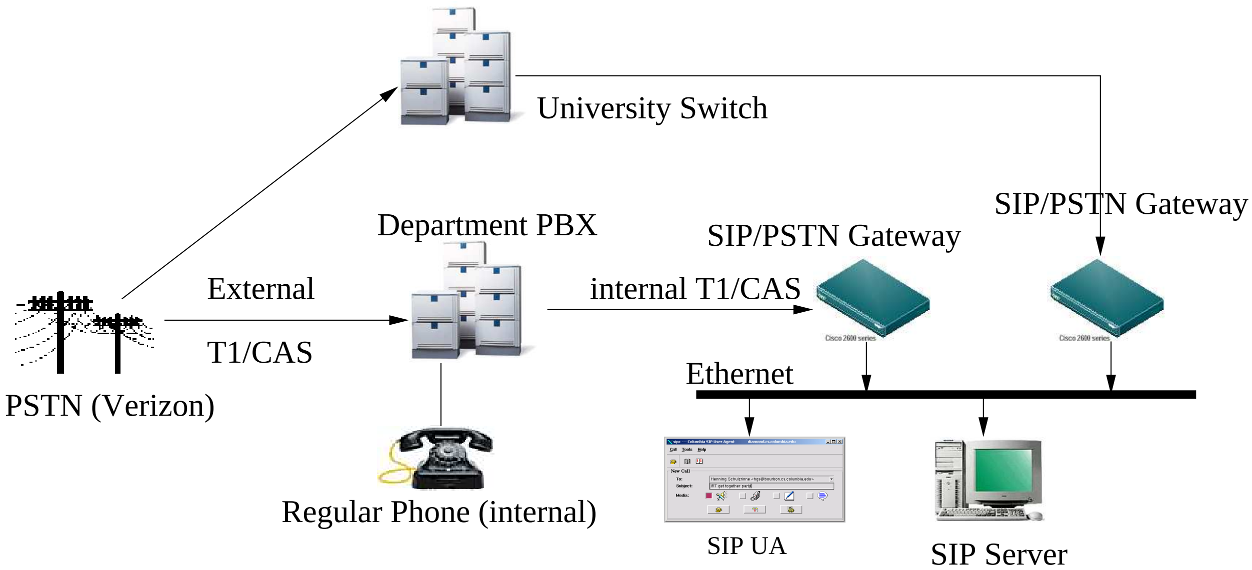


Figure 15: PBX set-up; an incoming call flow

them have to do with the proprietary and arcane nature of PBX systems.

T1 line type: A T1 line can be either channelized or PRI [5, pages 446-447]. The former supports 24 DS-0 (64 kb/s PCM) voice channels, and uses Channel Associated Signaling (CAS). CAS is a form of in-band signaling, where some bits in each voice channel are "robbed" for signaling, hence the nickname robbed-bit signaling. In comparison, PRI supports 23 DS-0 B (voice) channels plus 1 DS-0 D (signaling) channel, and uses an out-of-band signaling method known as Common Channel Signaling (CCS). PRI is a form of business grade narrow-band ISDN. Channelized T1 has more voice channels, but each channel is not full 64 kb/s, and it is not guaranteed to provide advanced features such as Caller-ID. We use channelized T1 in our PBX for both T1 lines. The PRI service may require additional hardware in the PBX's T1 line-card.

Line type	voice channels	signaling	caller-ID
Channelized	24, robbed-bit	CAS	maybe
PRI (ISDN)	23B + 1D	CCS	yes

T1 line characteristics: T1 lines can use several different line codings, including Alternate Mark Inversion (AMI) or Bipolar 8 Zero Substitution (B8ZS) [5, pages 175-182]. We recommend B8ZS because it provides a full 64 kb/s for each DS-0 channel, whereas AMI steals one out of every eight bits (leading to a 56 kb/s channel), thus degrading the voice quality. The line coding is *not* always independent of the line type. For instance, AMI cannot be chosen with PRI, because PRI requires a full 64 kb/s channel. Second, one needs to select a framing type, usually either Super Frame (SF), also known as D4, or ESF (Extended SF) [5, pages 210-216]. We choose ESF, which is more advanced and should be supported on most PBX systems.

Trunk type: The most popular trunk types are DID (Direct Inward Dial) and TIE. A TIE line is a bi-directional trunk line. The name TIE comes from the fact that the trunk line "ties" two nodes together. We recommend configuring the T1 line as a TIE trunk, because it allows both DID (incoming) and outgoing calls.

Channel type: The channel type can be data, voice-only, or data/voice. This is a crucial parameter. If a channelized T1 line is used on a Meridian system, the channel type must be set to voice-only, otherwise, IP-to-PSTN calls may fail as the PBX could treat a call as data transmission. In a Nortel Meridian PBX system, this parameter is named DSEL (Data SELector).

Access permissions: Nortel Meridian systems use a concept called Network Class Of Service (NCOS). Typically, a low NCOS means low access permission. For example, 1 may indicate internal or local call only, and 7 may indicate all long-distance allowed. To restrict SIP phones to local PSTN (outgoing[3]) calls only, the administrator should specify a NCOS of 1 for the internal T1 line.

For incoming calls, however, the scheme is less obvious: when a call arrives at the PBX, whether incoming or outgoing, the calling entity's NCOS is compared against the callee's NCOS in the PBX call routing table. Note that the callee in this case is *not* the internal T1 line, but the 713x range - a virtual entity. The calling entity for an incoming call is the external T1 trunk, and it usually has a NCOS of 0. The call goes through only if the caller's NCOS is high enough. In our test-bed, the 713x range is a virtual phone number range, being a part of what is called Coordinated Dialing Plan (CDP). One must ensure that the routing entry for this CDP (713x) range has an NCOS value less than or equal to the caller's NCOS, so that incoming calls can be accepted. Therefore we use an NCOS of 0 for the CDP entry.

These issues are summarized in Table 8.

T1 attributes	common choices	recommended
Line Type	Channelized, PRI (ISDN)	PRI, used channelized
Line Coding	AMI, B8ZS	B8ZS
Framing	D4(SF), ESF	ESF
Trunk Type	DID, TIE	TIE
Channel Type	data, voice-only, voice-Data	voice-only

Table 8: Summary of key attributes

[3]Here outgoing and incoming are viewed from the perspective of the PBX (or the department).

7 Security Issues

We need to deal with three security-related issues, namely user registrations, remote callers and access to the PSTN. First, user registrations need to be authenticated to prevent unauthorized users from redirecting calls to themselves or elsewhere. We use digest authentication, where a shared secret between the server and the client is verified via challenge-response. The use of digest authentication is a system-wide parameter that can not be over-ridden by per-user configuration. However, users can configure their profiles to do a more fine grained authentication on incoming calls. A local user may choose to force remote callers to be authenticated or may allow incoming calls from any user. Fig. 16 shows the message flow for typical REGISTER and INVITE messages.

We cannot rely on a public key infrastructure, so we chose a more pragmatic, albeit less secure, approach. Our authentication goal is to establish a consistent mapping between a caller's SIP identity and her email identity. If a caller is unknown, a mail message is sent to the same identifier, treated as an email address. The mail message contains a randomly generated password and a link to the original called SIP URL. The caller simply retries the call after receiving the email message and stores the secret for future use. This ensures that the SIP caller is indeed identical to the corresponding email address. (One approach that does not work is to simply have the callee issue an INVITE in the reverse direction. This could be easily abused to cause somebody to make nuisance calls to a third party.)

It should be noted that it is much harder to use call filtering to prevent VoIP crank calls than their PSTN equivalent since Internet identifiers are abundant and cheap. However, it is possible to at least restrict unknown callers to, say, daytime hours or leaving voicemail.

Finally, we need to restrict access to the PSTN gateway so that not everyone can make unauthorized calls through our gateway. The next subsection elaborates on this issue.

7.1 PSTN security

In most cases, only an outgoing call incurs a toll charge. The last line of security in such case is the PBX, but the PSTN/IP gateway and the SIP proxy server are in general much more flexible and programmable. Currently, our gateway does not have user authentication and authorization capability, so we delegate this functionality to the SIP proxy server, so that only authorized users can make calls. However, if a user discovers the gateway's IP address, he can still bypass the proxy and make free calls. To enforce security without adding security code to the gateway, we can make the gateway reject direct-dialed calls. Since our gateway is a fully functional Cisco 2600 router with IOS (Internetwork Operating System), we can use the IOS Access Control Lists (ACL) to accept SIP requests only from the proxy, but accept UDP media streams from all potential users. The following example IOS ACL blocks certain inbound traffic on the gateway 128.59.19.61. Its subnet has a net mask of 255.255.255.0, hence its ACL's reverse mask is 0.0.0.255. The gateway would allow UDP media packets from machines on the same subnet with port number range 512 to 65535. However, since SIP requests are typically carried on UDP port 5060 (inside 512-65535), to reject "direct-dialed" SIP calls, our ACL allows a UDP packet only if its destination is not the gateway (128.59.19.61)'s SIP port (5060). Note that the SIP proxy server (128.59.19.62)'s request will still be honored because ACL rules are evaluated in the same order they are defined.

```
interface FastEthernet0/0
 ip address 128.59.19.61 255.255.255.0
 ip access-group 101 in
```

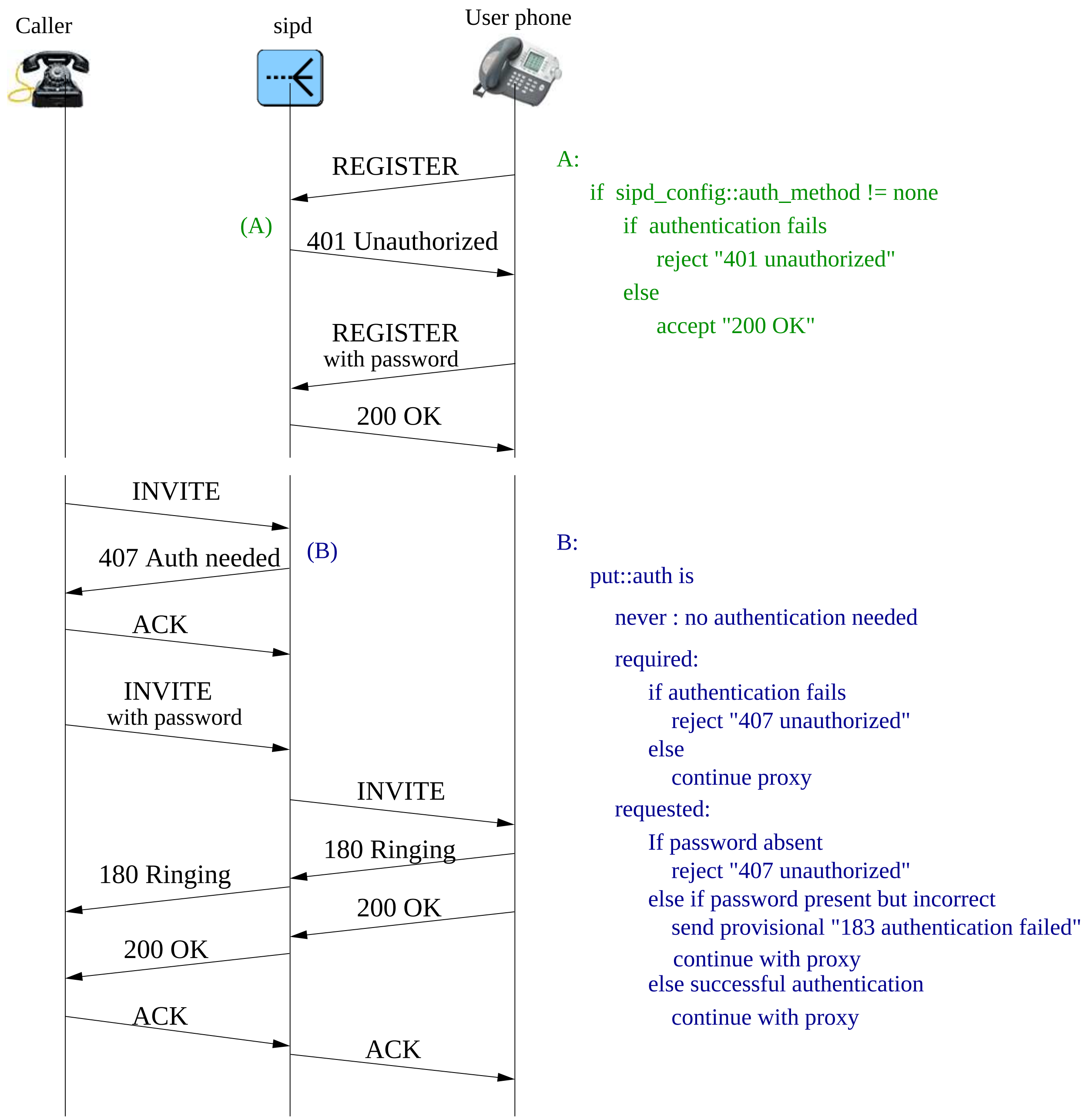


Figure 16: Message flow for REGISTER and INVITE with authentication

```
...
access-list 101 permit ip host 128.59.19.62 any
access-list 101 permit udp 128.59.19.0 0.0.0.255 \
   range 512 65535 host 128.59.19.61 neq 5060
```

There is a potential security problem of spoofed IP address. For example, if a malicious caller spoofs its IP address as one of the allowed IP addresses for the INVITE request to a long-distance destination, the telephone call is initiated and billing starts as soon as the remote destination picks up the phone. However, the "200 OK" response will get routed to the SIP proxy server (128.59.19.61) instead of the malicious caller. Since the call setup response (200 OK) can not be ACKed by the spoofed address, the gateway will timeout and terminate the call in approximately one minute. However, the system will get billed for the first minute of the call. To prevent such attacks, we can use challenge-response kind of authentication (e.g., digest). A simple null authentication scheme [30] can be used by the gateway to challenge the SIP proxy server so that the gateway knows that the source IP address is not spoofed as it can be reached by the initial "401 Unauthorized" response.

All the calls from the gateway are forwarded to the SIP server. This configuration, along with the SIP Record-Route mechanism that forces subsequent requests within a call to traverse a designated set of proxies, allows call logging and billing services to be part of the SIP server.

7.2 TLS: Transport Layer Security

Confidentiality, integrity protection, and server-client authentication for SIP call signaling messages can be provided using TLS [8], an IETF standard security protocol that resides logically on top of TCP. TLS is commonly used in web (using the https URL scheme) for authentication and encrypted communications between clients and servers. TLS allows the client to authenticate the server and vice-versa, select mutually supported cryptographic algorithms (or ciphers), generate shared secrets using public-key encryption, and establish an encrypted connection for data exchange by higher level protocols.

We use the OpenSSL TLS library [2] for providing TLS services in our SIP server, sipd. sipd can accept TLS connections, and proxy SIP requests and responses using TLS as transport. sipd currently allows server authentication. This means that UAC's (user agent client) can authenticate the server. We will add client authentication by which a UAS (user agent server) can authenticate the server instead of otherwise. An important thing to note that, if a user has registered contacts only for UDP or TCP, sipd will proxy an incoming call through UDP or TCP by normal SIP forking mechanisms, which could potentially limit the security-advantage gained on the first hop of the call path. Support for "sips:" URI scheme is in progress.

7.3 Anonymous access

The expanding popularity of the Internet presents new possibilities for communication, but also new implications for the users' privacy. There are many reasons why a user on the Internet might wish to remain anonymous. However, even to a casual investigator, it is often possible to determine the identity of an Internet user, e.g., using IP addresses. For this reason, many Internet developers and entrepreneurs have set up anonymizing services designed to hide the identity of their users for web or email applications.

A SIP call anonymizer can be implemented as a back-to-back user agent (B2BUA). The anonymizer receives a SIP INVITE request, changes the From header and sends out the request to the actual destination. Anonymizers thus maintain a mapping between real SIP addresses and anonymous ones. Thus, whenever a request comes in destined for an anonymous address, the translation is the reverse of the normal case and the From address is maintained, while the To address is set to the corresponding real address. Whenever a response is received to a previously forwarded request, it is forwarded in the proper direction. The anonymizer also sometimes must generate responses of its own; for example, a remote address might be unresolvable, in which case the user will be notified. The anonymizer re-writes the IP addresses in the SDP of the request and responses and then acts as a simple UDP relay for RTP/RTCP media packets.

Our implementation provides supports both in-memory (hash table) or external SQL/JDBC-based databases for maintaining mappings. It should be noted that, when using a database backend for the anonymizer, anyone who can read the database has immediately compromised all users' identities. When setup as a service for multiple users, the software automatically generates a random SIP address of the form `anon-nnnn@hostname` where `nnnn` are four random letters. For the outbound proxy case it takes care to prevent two different users accidently ending up with the same randomly-generated address. It also supports sending and receiving of instant messages using the SIP MESSAGE method.

Sometimes, a user will wish to hide his IP address but still use a known SIP address to communicate with another party. If a user agent sends a REGISTER message to our implementation, the address in the REGISTER message will become the client's new anonymous ID, replacing the default randomly-generated address. In addition, it can be configured to forward notice of these registrations to some central directory server. So, a user who wishes to use a publicly-known SIP address without revealing his IP address can do so.

It is important to look into possible attacks that might be used to compromise a user's anonymity, and how they might be defeated in the future. Without the use of other security measures, anyone who has access to the network segment containing the anonymizer can spy on the network traffic and guess the identity of any anonymous users. The easiest way to do so would be to compare messages coming into the anonymizer with those coming out, and note any similarities. TLS/SSL can be used to resolve this.

7.4 Other issues

One issue can be raised: what happens if the user allows both secure and non-secure connections. For example, if the user's contact locations list both TLS and non-TLS contacts, should the proxy-server try to use the secure location before attempting the non-secure contacts. Secondly, how should we allow services to both authenticated and un-authenticated users. For instance, allow leaving a voicemail for non-authenticated caller, but forward the call to the cell-phone for authenticated callers (because the callee will be billed for the telephone location in the contact). We can use the q value in the contact locations to specify the preference for various contacts. So the user can give higher preference for the TLS contact. SIP servers can allow anonymous access with restricted privileges.

Security of the whole system is defined by the security of the weakest link in the system. So to make the system secure all components must use high degree of secure protocols. Security loop holes are exploited usually in the weaker parts of the system. For instance, users should be careful in trusting the certificate authority otherwise an intruder and generate spoofed certificate and get access to the secure SIP services.

8 Monitoring and Accounting

8.1 Logging and accounting

Next to call establishment and teardown, call logging is probably the most important feature in any telephony system. Call logging is needed for generating accurate and verifiable billing records. In addition, accurate call logging information can help understand usage behavior of the system and can help in system dimensioning.

Our SIP server sipd allows administrators to configure a variety of loggers including text (flat) file, SQL, RADIUS, and Unix syslog facility. In addition, log output can be piped to another running program. Use of the pipe mechanism allows us to easily extend the system behavior. For example, piped output could be fed into an SNMP monitor, which will generate traps for every "407 Not authenticated" error, and could inform an administrator if call failures exceed a certain thereshold.

8.1.1 SQL

The primary logging mechanism that helps us to generate billing records is the SQL database. For every transaction served by sipd several parameters are logged in the requestlog SQL table (described in Section 3.2). In particular, the SIP method (INVITE, ACK, BYE, REGISTER), the status code (which indicates whether the transaction was successful or failure) and the canonical identities of the caller and callee are logged. The SIP server also logs the Call-ID which uniquely identifies the various messages exchanged as part of the same call. The Call-ID is useful for matching the call setup and teardown requests and hence helps us to calculate the duration of the call. It is to be noted that SIP allows call teardown (BYE) messages to be exchanged directly between endpoints. The use of SIP Record-Route mechanism forces the endpoints to send the teardown message through to the SIP server.

8.1.2 RADIUS

In order to integrate CINEMA into existing telephony infrastructures, it is necessary that the accounting information generated by the SIP server is communicated to and usable by external AAA accounting [4] servers. One protocol that helps achieve such interoperability is RADIUS (Remote Authentication in Dial-In User Service) [23], which is a protocol that can be used for carrying accounting information between network entities. RADIUS has traditionally been used by ISP's to provide authentication and accounting services to their dialup customers. Its extension mechanism [22] allows us to easily extend these messages to carry SIP transaction records. Our SIP server sipd can be used as client of one or more RADIUS servers. Addresses of RADIUS servers are configured in the SQL database. For each transaction sipd sends a RADIUS message to the RADIUS server with the details of the transaction as defined in [32].

8.1.3 Billing

In this subsection we describe how the SQL call logs generated by the sipd server are used to generate accounting information.

Telephony pricing structures usually follow a differential model instead of a fixed flat-rate model. For instance, they could depend on time-of-day, user type and destination number. Charge per call could be determined based on the type of the call, type of the user and time-of-day. For example, international calls made during night time could be charged less than those made during the day. Another scenario could be in a campus environment where telephone calls are free for faculty whereas students may have to pay for their calls. We use a flexible configuration scheme where a particular telephone number prefix could be associated with user classes and time periods. Fig. 17 is an illustration of how tariffs are configured. `$/unit` is the call `rate` per `increment` time unit. The total charge per call is computed as `[ceil(call duration/increment)] * rate`. For example, if the rate is $0.15 with increment 6 seconds, then a 40 second call costs `ceil(40/6)*0.15= 7*0.15 = $1.05`.

List of Tariffs

Prefix	User class	Valid from	to	$/minute	Daily from	to	increment (sec)
+1*		1999-01-01	2002-12-31	0.100	00:00	23:59	60
+382*		1999-12-01	2002-01-31	0.500	00:00	23:59	60
+44*		2000-01-01	2001-12-31	0.050	00:00	23:59	60
+44*		1999-12-01	1999-12-31	0.096	21:00	06:59	10
+49*		2000-01-01	2002-01-31	0.150	00:00	23:59	6
1*		1999-12-01	2002-01-31	0.100	00:00	23:59	60
1212*	faculty	1999-12-01	2002-01-31	0.010	00:00	23:59	60
1800*		1999-01-01	2002-12-31	0.000	00:00	23:59	60
1877*		1999-01-01	2002-12-31	0.000	00:00	23:59	60
1888*		1999-01-01	2002-12-31	0.000	00:00	23:59	60
70??*		1999-12-01	2002-12-31	0.010	00:00	23:59	60
71??*		1999-12-01	2002-01-31	0.000	00:00	23:59	60

Figure 17: Sample tariff configuration

Telephone number prefixes are standard regular expressions, hence a "*" matches zero or more digits. A leading "+" is used to indicate a globally routable (international) number. Rules are matched in a first to last order, and based on the caller's user class so that the telephone number 12129397028 will match the rule 1212* if it is made by a user belonging to the "faculty" user class, but will match the 1* rule if made by other classes of users.

8.2 Monitoring with SNMP

Similar to classical PSTN systems, an Internet telephony system should be highly available with negligible downtimes. Call routing depends on the proper functioning of the SIP server and hence it is necessary to have a mechanism by which the state of the SIP server can be monitored, with the administrator being alerted when necessary. SNMP [6] is a protocol that can be used for this purpose. We have implemented a SIP MIB [19] on our SIP server using the Net-SNMP [3] framework. Any SNMP station can be used to query and monitor our SIP server. An example screen shot of one such station, MG-SOFT MIB browser[4], is shown in Fig. 18. Our SIP server can also be configured to send SNMP traps when events of interest occur. As an example, the server can send trap notifications to the SNMP station when the call arrival rate exceeds a configured threshold.

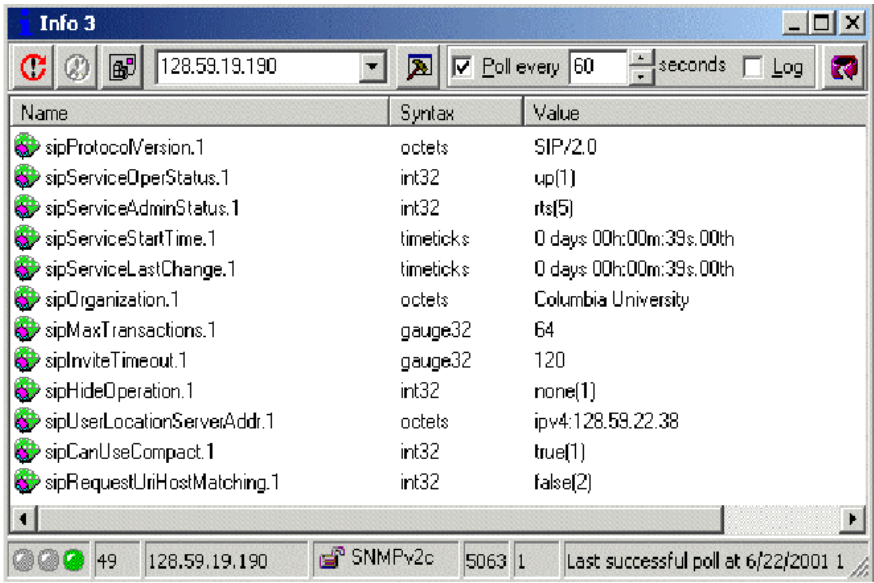


Figure 18: SNMP agent for SIP MIB

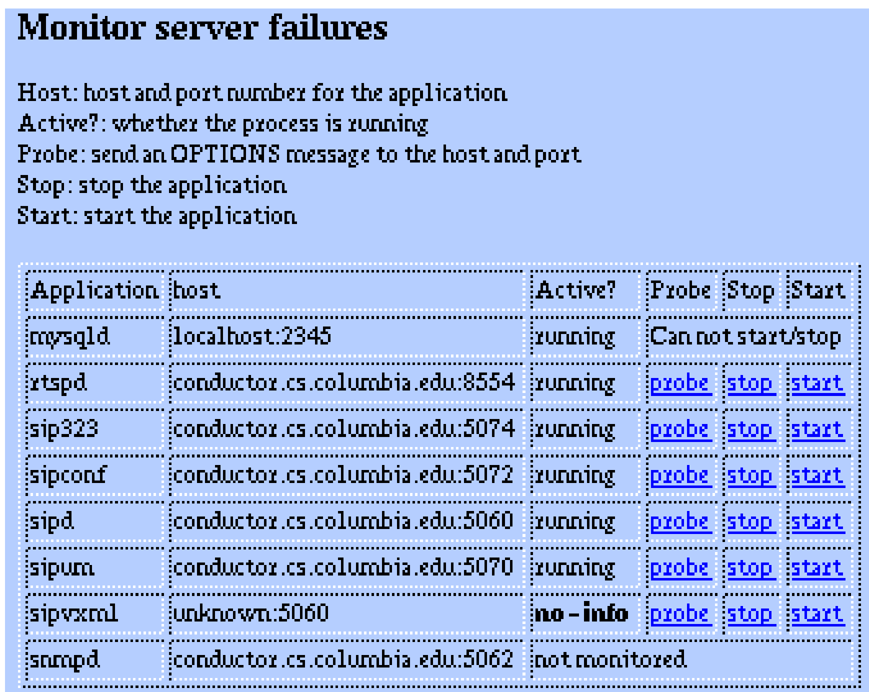


Figure 19: CINEMA monitor

8.3 Server monitoring

In addition to SNMP-based trap notifications and monitoring, we also provide a web-based user interface by which administrators can check the liveness of various CINEMA servers (probing), start and stop them. This is shown in Fig. 19. The "Activity" column indicates whether the

[4]http://www.mg-soft.org

server process is running on the host. The "probe" option can be used to determine the activity by sending a SIP OPTIONS message to the server host and port. We use this to find out if the server process is actually handling requests or has gone into some non-processing (e.g., deadlock) mode due to some program error.

9 Other Services

This section describes other services provided by the system, including unified messaging (Section 9.1), multi-party conferencing (Section 9.2), instant messaging and presence (Section 9.3), interactive voice response (Section 9.4) and IPv6 support (Section 9.5).

9.1 Unified messaging

Answering machines and voice mail systems are crucial PSTN components. They are equally important in an Internet telephony environment. Installing the voice mail service on every SIP phone is inefficient and inconvenient if the user has many phones. Secondly, it may not work if the user has calls forwarded to many different devices. Also, end-system-based answering machines place a high premium on the reliability of those end systems. Centralized voice mail systems have an advantage in the centralized management of user accounts and configuration. An Internet-based voice mail system can be integrated easily with other Internet services like email, web, video mails and fax, giving an unified messaging environment. Moreover, it can use the existing protocols and tools, like SIP and RTSP (Real Time Streaming Protocol [33]).

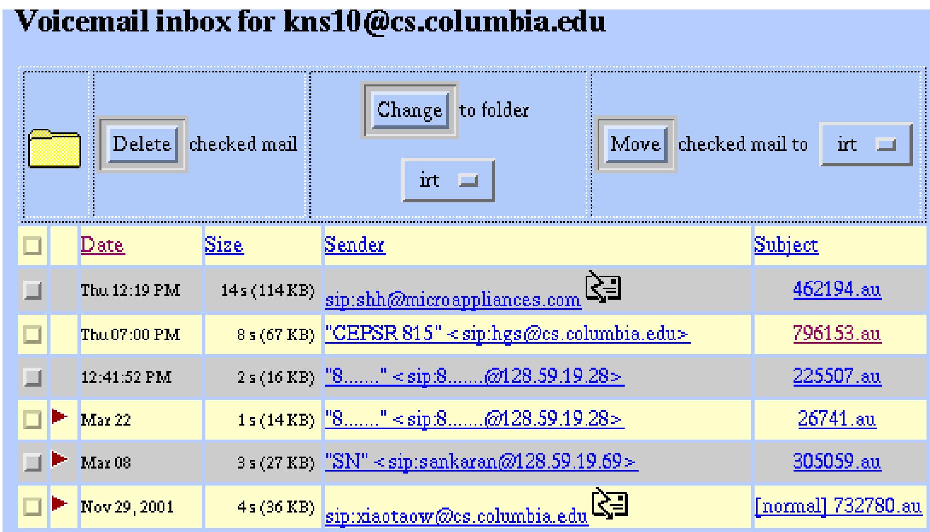


Figure 20: User voicemail web interface

Our system uses SIP for signaling and RTSP for storage and retrieval of voice messages as described in [38]. The user gets an email notification when a new message arrives. The user messages are also listed on a web page as shown in Fig. 20, where they can be played by just a mouse-click. Alternatively, an RTSP client such as Apple's QuickTime can be used to play back the message. Using streaming media to deliver voicemail avoids having to download the whole message while traveling, for example.

9.2 Multi-party conferencing

Multi-party conferencing is also an important telephony service, provided in the PSTN by conference bridges. Our Internet telephony environment employs a SIP conference server with audio and video capabilities. The conferences can be set up via a web interface.

Every conference is identified by an address similar to the canonical user identifier, e.g., staff-meet@cs.columbia.edu. Users join the conference by dialing that conference address. A telephone number alias can be created for the conference so that regular PSTN users can also take part in conferences.

We are planning to extend the system to provide dial-out conferences instead of the traditional dial-in conferences. In this mode, the conference server itself invites the participants at the start of a pre-configured conference.

The SQL database stores various conference attributes and can be updated from a web page. These include the conference identifier, duration and schedule, authentication mechanism for restricted conferences, limit on the number of participants, types of media allowed. The participant list can be further restricted by defining different capabilities for different set of participants. For instance, one may want a conference where anybody can listen but only users located on the *cs.columbia.edu* domain can send media. Authentication of PSTN phones requires some form of voice interface (Section 9.4).

9.3 Instant messaging and presence

Presence and instant messaging are popular Internet services that need to be supported both by servers and clients. Combining presence and Internet telephony offers improved services, reducing, for example, the number of failed call attempts or automatic redirects to voice mail. Users subscribe to each other and are notified of changes in state (such as online, offline, busy, idle) and can send each other short instant messages. SIP-based instant messaging and presence provides a standard way to send instant messages and form buddy lists. Presence can be coupled to phone status, marking the user as busy when in a phone call.

Our SIP server, sipd, has a built-in centralized presence server that is used to store and convey presence status. In this model, user Alice who is interested in the presence status of Bob subscribes to a server conveying her interest. The server keeps track of subscriptions for each user. When Bob comes online, his user agent registers with the SIP server, providing Bob's contact address. The server then uses this registration information to asynchronously notify Alice about Bob's status.

Our user agent, sipc, can retrieve presence information about buddies from a SIP presence server such as sipd. The status of buddies is displayed as part of the user's address book. Users can exchange instant messages with their buddies using the SIP MESSAGE [29] method that can be sent either directly, or through a proxy server.

9.4 Interactive voice response

An Internet telephony system should also be capable of providing services such as listening and deleting voicemails, or authentication when joining conferences to a PSTN user who is limited to using the telephone keypad and spoken audio for supplying input.

VoiceXML [40] is an XML-based markup language for voice dialogs, is designed to ease creation of audio dialogs. These audio dialogs feature synthesized speech, digitized audio, input speech and DTMF recognition, and audio recording for telephony applications. The separation of user interaction code (such as DTMF and speech recognition) from the service logic (email by phone, providing weather reports) facilitates development of a variety of interactive voice-response services.

We have built a SIP-based VoiceXML browser that allows telephone users to interact with CINEMA components. This component resides between the SIP/PSTN gateway and the voice mail or conferencing service in our system. A dialog specification for a service such as voice mail access is specified in VoiceXML. Our browser fetches this dialog from a web server (using HTTP) and interprets it in a form suitable for PSTN users.

9.5 IPv6 support

This section describes our experience in implementing IPv6 in CINEMA components. The main advantage of IPv6 over IPv4 is that it has more address bits and thus solves the address space problem at least in the forseeable future.

On startup, CINEMA applications determine whether the host system has usable IPv6. Some hosts can have different IPv4 and IPv6 host names (e.g, thalys.cs.columbia.edu for IPv4, and thalys.cs.ip6.columbia.edu for IPv6). In such cases, the IPv6 hostname needs to be explicitly passed either through configuration file or via the command line. The hostname is then resolved into an IPv6 address. If the application is unable to get an IPv6 address for the supplied host name, the IPv6 module is disabled even if the operating system has support for IPv6. This can happen, for instance, when no IPv6 addresses are configured for the system's network interfaces.

All name-address mappings are handled first by using OS support (`getaddrinfo`, `getnameinfo`, `gethostbyname`, `gethostbyaddr`, the latter two are used when former are unavailable), and then by querying a name server. SRV records are always resolved by contacting the name server. Currently, CINEMA can contact only IPv4 name servers. However, it can recognize and use AAAA address records. Name server configuration is specified in `/etc/resolv.conf` file for Unix systems, and `resolv.conf` file in Windows systems.

All SIP URI's with IPv6 address have the numeric address enclosed within brackets, e.g., `[1080::8:800:200C:417A]`. Our proxy server can proxy between IPv4 and IPv6 hosts. However it is up to the user agents to negotiate a suitable network address for media transmission. In order to facilitate SIP requests such as BYE between hosts using heterogenous network addresses, the record route mechanism should be used. This will cause all subsequent requests to take the same signalling path (through the proxy server, which can handle the heterogeneity).

If both A and AAAA records are returned by the name server during a SRV lookup, our implementation will always try the A record first. There are three ways to implement the Contact header generation: use IPv4 address, use IPv6 address or put two Contact headers containing both. The last option is needed if the IPv4 and IPv6 hostnames are not the same. Our implementation prefers the IPv6 host name if the registrar or outbound proxy is a IPv6 host. For the user agent library, when the stack initiates a call it adds both IPv4 and IPv6 addresses to the SDP media description, if the host is IPv6 capable.

10 Implementation

Many of the architectural components described in section 3 are implemented in C/C++ (e.g., sipd, rtspd, sipconf, siph323 and sipum). All these pieces of software share the common code base wherever possible. The common part is identified and abstracted as a set of libraries. Then the applications are built on top of these libraries. The libraries and the applications consititute the **Columbia InterNet Extensible Multimedia Architecture** (CINEMA). This section describes the various modules used for implementing the system and discusses the design details of the SIP library.

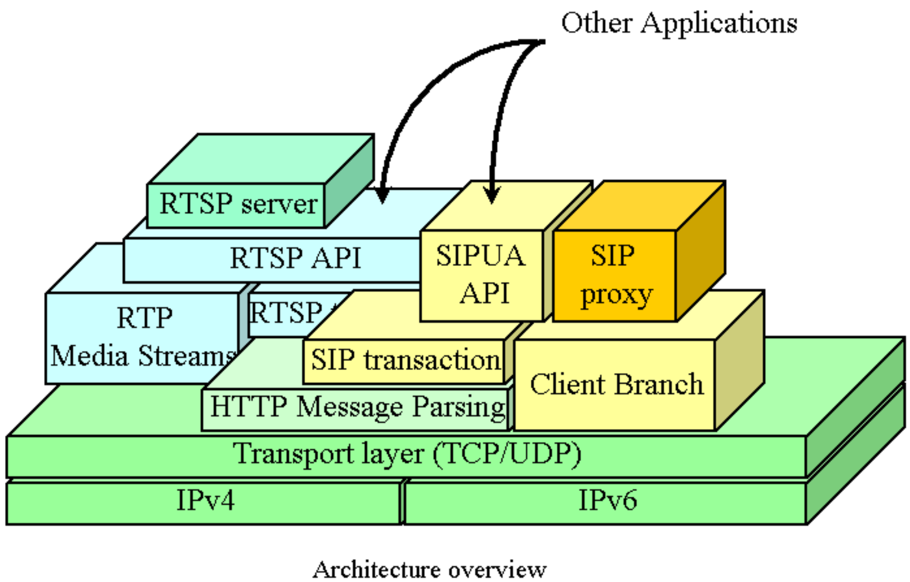


Figure 21: Software Design Modules

The layered hierarchy of various sub-modules is shown in Fig. 21. The lowest transport layer is assumed to be TCP or UDP. We use the standard `socket` interface for this layer. A generic HTTP message parsing layer is used for parsing various HTTP-like messages, SIP and RTSP. RTSP and SIP-specific routines are added above this layer. In particular, the RTSP transaction layer maintains the state for a media session, while the SIP transaction and client branch layers maintain the state for a SIP transaction. The SIP transaction layer is used in implementing the SIP proxy server. SIP user agent library uses the transaction layer, and implements the call control state machine above that. Both internal and external libraries are used to build various applications as shown in Fig. 22.

The CINEMA libraries are briefly described below:

libcine: libcine is a generic library with general-purpose utility functions for parsing HTTP messages, manipulating URIs, logging requests, MD5 functions, database access, software license

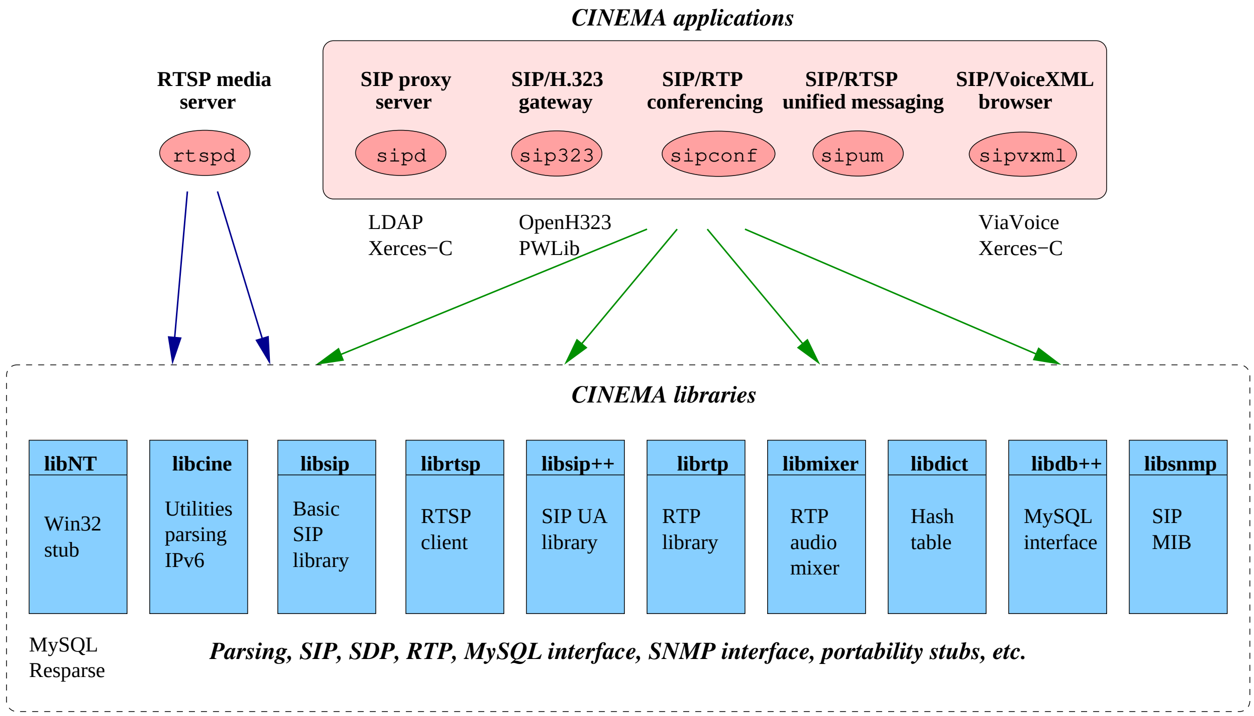


Figure 22: Software library and applications

check, TCP/UDP wrapper, dynamic string, resolving host names, logging debug information. This library is shared by both SIP and RTSP implementations.

libdict: libdict is a general-purpose library for dictionary or hash-tables in C.

libdb++: We use the MySQL database in our environment to store various user and system configuration. This module is a high-level C++ interface for accessing the database tables built as a wrapper over the libmysqlclient library. It also provides an in-memory database mechanism to speedup database access. It also implements a file based authentication to allow non-database type simple applications like user agent libraries.

libsip: libsip is a SIP library in C that implements the SIP transaction and client branch layers. It allows different authentication mechanisms used by SIP. It also contains the SNMP interface to libsipsnmp and the database interface to libdb++.

libsip++: libsip++ is a SIP user agent library that implements the call control for establishing, maintaining and terminating a SIP call. It also has SDP parsing routines. It uses libsip for implementation of transaction and client branch layers.

libmixer: libmixer is a RTP audio mixing library. It is used in the conferencing server implementation.

NT: NT library implements the basic portability stubs on the Microsoft Windows platform for the commonly used Unix functions. In particular, it contains routines for `aliases`, `crypt`, `hashtable`, `inet`, `regex`, `getopt`, and `pthread`. These stubs allow us to use the same code base for both the Unix and Windows platforms.

10.1 SIP library overview

A *SIP transaction* is identified by the Call-ID, To, From and CSeq SIP headers and the SIP request URI. A transaction roughly corresponds to a request and all its responses plus their retransmissions.

A transaction can be of two types: proxy transaction and user agent transaction. A proxy transaction is associated with a set of client branches. When the proxy *receives* a request from an upstream client it creates the transaction object then forwards the request to the downstream server(s) using client branches. The responses received by the client branches from the downstream server(s) are forwarded to the upstream client. A user agent transaction can either receive a request and terminate it or can originate a request and wait for responses. Thus, the user agent transaction can be further classified into incoming transaction (without any client branch) and outgoing transaction (with only one client branch). Contrast this with the proxy transaction which can have one or more client branches. More than one client branches signify the forking proxy behavior. A forking proxy forwards a call to several possible locations simultaneously and completes the call setup by connecting the caller to the first location answering the call. A client branch represents a possible location where the destination can be reached.

Once a request is received it can be processed in a variety of different ways: proxy it (proxy transaction), send a redirect response, inform the user (user agent transaction), or reject it. The decision to choose appropriate behaviour can be governed by different *policies* as shown in Fig. 23.

The rest of this section assumes familiarity with the SIP specification.

10.2 SIP transaction and client branches

A SIP transaction is implemented as a `request_t` structure (Fig. 24. It contains the following pieces of information.

- `time_received` stores the time when the first request (not the retransmissions) in this transaction was received.
- The actual incoming message is parsed and stored in `parser_t p`. The message body is part of the `request_body` field. `source_sin` stores the IP address and the port number of the remote upstream client that sent the request.
- `socket` is the socket identifier to be used for sending responses upstream for this transaction. `type` represents whether TCP, UDP or TLS is being used for this transaction for communication with the upstream client. The IP address and port number of the upstream client to which the response has to be sent are stored in `sin`.
- The last response sent to the upstream client is stored in the `status` and `reason` fields. The actual response, i.e., the `headers` and the `response_body` are also stored in this transaction object.
- Authentication information, challenge and credentials, are stored in the `authorization` and `authenticate` fields, respectively.
- The `server` field is an opaque object pointing to the higher-level construct to which this transaction object belongs. For example, it could point to the user agent object which started the listening UDP and TCP threads. All transactions created as a result of messages received from these instances of the listening threads correspond to this user agent object. This allows for implementing multiple user agent objects, say one on port 5060 and another on 5070.

  A `call` pointer is also an opaque pointer used only for a user agent transaction, to associate it with the higher level call control objects.

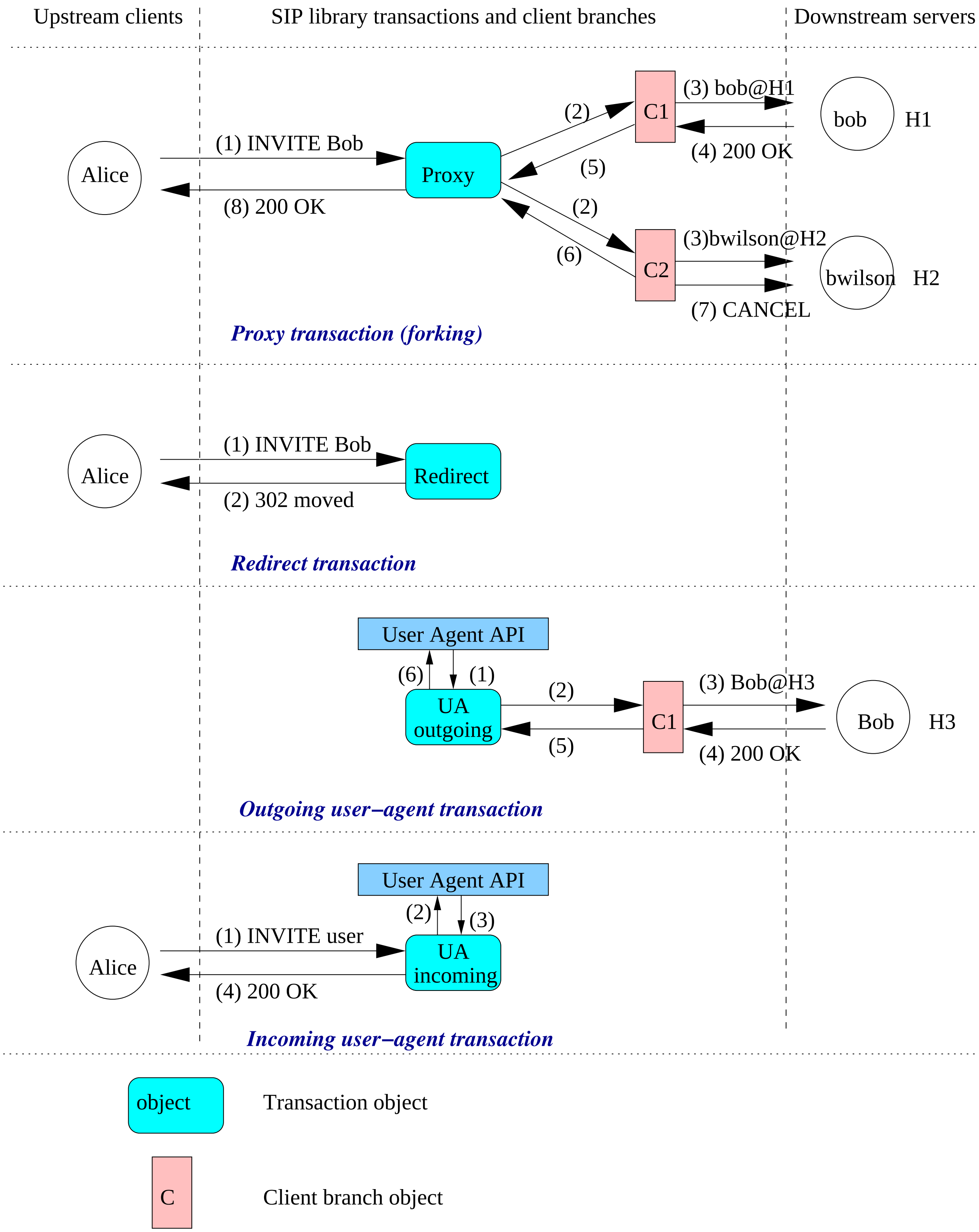


Figure 23: SIP transaction and client branches

- Some of the more frequently used SIP message headers are extracted and stored as `version`, `cseq`, `callID`, `to`, `from`, `expires`, `contact` and `user_agent`. `via` stores the top-most Via header in the SIP request.

```
struct request_t {
  /* Basic request information */
  double time_received;             /* time request was received */
  /* Where is the response going to? */
  int type;                         /* SOCK_DGRAM, SOCK_STREAM, SOCK_TLS */
  int socket;                       /* socket id for response */
  struct sockaddr_storage source_sin; /* socket address request came from */
  struct sockaddr_storage sin;      /* socket address for response */
  /* we are not using type == SOCK_TLS because most of the code is written as if (type == SOCK_DGRAM) else ... making the
   * type multiple valued, may cause subtle breaks somewhere. */
  void *ssl_data;                   /* a pointer to SSL specific data or NULL, identifies a connection specific structure for ssl*/
  /* Information about the request */
  parser_t p;                       /* header for this request */
  body_t request_body;              /* The body of the request; */
  http_authorization_t authorization; /* Authorization header */

  /* Information about the response */
  int status;
  char *reason;
  headers_t headers;                /* Headers of response */
  body_t response_body;             /* Body to send with respnse */

  void *server;    /* server object associated with this request */

  char *version;                    /* request version */
  /* SIP request headers */
  sip_cseq_t cseq;                  /* sequence number of request */
  char *callID;                     /* the call ID of the request */
  sip_addr_t to, from;              /* addresses */
  sip_via_t via;                    /* top-most Via header */
  time_t expires;                   /* Expires header */
  sip_contact_t *contact;           /* SIP Contact headers */
  char *user_agent;                 /* SIP User-Agent */

  unsigned long hash;               /* Hash of identifying headers */
  int in_request_list;              /* Is this added to request list? */
  int acked;                        /* request has been ACKed */

  /* SIP-specific Information about the response .*/
  http_authenticate_t authenticate;   /* SIP WWW-Authenticate (response) */

  /* Transaction state: */
  /* Locking. */
  pthread_cond_t cond;             /* condition variable */
  pthread_mutex_t mutex;           /* mutex for access */
  /* User information.  Used when necessary. */
  put_table_entry *source_user;      /* The originator of the transaction */
  put_table_entry *destination_user; /* The (logical) destination of the transaction */
  /* the canonicalized request uri. check the status flag before accessing the unique pointer for the canonicalized uri. */
  canon_result canonicalized_uri;
  char *method;                     /* For requests, the SIP method */
  /* Communication with subsequent requests and responses */
  message_t *mqueue;                /* Message queue for this transaction */
  int sent_final;                   /* Whether we've sent a final response to the original request */
  int sent_200;                     /* Whether we've sent a 200 response to to the original request. */
  /* State for the policy core */
  int done;                         /* Whether the policy handler is done with this transaction. */
  struct timespec final_timeout;    /* Final timeout for the transaction */
  /* Transaction & user policies */
  struct transaction_policy_t *transaction_policy; /* Low-level policy to use for this transaction */
  struct timespec transaction_policy_timeout; /* Transaction policy timeout, or 0 */
  void *transaction_policy_info;    /* Info specific to a transaction policy */
  int transaction_policy_done;      /* Whether the transaction policy is done with this transaction. */
  struct user_policy_t *user_policy;  /* High-level policy to use for this transaction.  May be NULL. */
  struct timespec user_policy_timeout;/* User policy timeout, or 0 */
  void *user_policy_info;           /* Info specific to a user policy */
  int user_policy_done;             /* Whether the user policy is done with this transaction. */

  /* clients (proxy branches) */
  branch_t *branches;
  int branch_count;                 /* no of parallel searches (branches) */
  int branch_space;                 /* number of branch_t's allocated */
  int branches_outstanding;         /* current branches awaiting responses */

  void *call;                       /* call object associated with request*/
  struct sockaddr_storage remote_sin; /* needed for user agent in libsip++  */
  uri_t   outbound_proxy;           /* Address of the outbound proxy if any */
  int     preferred_type;           /* SOCK_DGRAM, SOCK_STREAM, SOCK_TLS or 0 as a preferred socket type
                                       for outgoing request, typically used by a user agent */

  struct request_t *next;           /* next request in queue */
};
```

Figure 24: request_t: transaction object

- Every time a message is received, the important fields (i.e., Call-ID, From, To, RequestURI and CSeq) are compared against all the existing transactions. To speedup the search, a hash of identifying headers is defined and kept in the transaction object. The search is further optimized by storing a hash-table of existing transactions instead of a linked-list.

- Whether an ACK is received from the upstream client or not (for a proxy transaction) is defined the boolean field, acked. For a user agent transaction it represents the status of whether ACK has been sent (outgoing transaction) or received (incoming transaction).

- Various inter-thread synchronization constructs, like mutexes and conditions are also part of the transaction object. Any thread which is manipulating this object should acquire the object's mutex before doing so. Also, a message queue, mqueue, is defined as part of request_t to allow inter-thread communication.

- sent_final indicates whether the state machine has sent a final response to the original request or not. Similarly, sent_200 tells whether it has sent a 2xx-class final response to the original request or not.

- The boolean variable done indicates whether the policy handler is done with this transaction or not. A policy defines the behavior of the transaction object on receipt of a new request or response or on timeout. The policy specific information such as state, is stored in policy_info.

- A transaction can have zero or more client branches. A client branch represents an outgoing message to the downstream server. Multiple client branches signify multiple parallel searches. The branch_count gives the number of client branches actually present whereas branch_space stores the number of branches allocated. Usually branch_space is more that branch_count to reduce frequent memory allocation overhead. branches_outstanding represents the current number of branches awaiting response from the downstream servers.

- outbound_proxy is the address of the outbound proxy server if any for the user agent transaction. The outbound proxy is the one to which all the requests are sent regardless of the actual request URI.

- preferred_type describes the preferred transport type: UDP, TCP or TLS for an outgoing user agent transaction. Note that the preferred_type value may be different from the actual type of the transport protocol used.

- Finally, the next element allows a linked-list of transaction objects. Currently a list of active transactions is maintained in the library.

A client branch (branch_t structure shown in Fig. 25) has its own state machine to send the request to the downstream server and collect all the responses. It also handles the timeouts and other error conditions (e.g., socket send error).

- Every branch is given an integer identifier, branch, which is an index in the branch array of the associated transaction. This id is also used in the Via header of the SIP requests sent to the downstream server.

- The parent field points to the associated transaction object of type request_t.

- The inter-thread communication is done using a message queue, mlist. The client branch uses the mutex lock of the associate transaction object for inter-thread synchronization. When a response is received from the downstream server, the response thread signals the

```
typedef struct branch_t {
  int branch;               /* branch index; -1 if none */
  struct request_t *parent;  /* parent request */
  pthread_t tid;            /* thread id for branch */
  uri_t uri;                /* URI to try */

  struct message_t *orig;    /* Original request for this branch */
  struct message_t *mlist;   /* Message queue for this branch */
  int sent_200;             /* Have we sent a 200 response to parent? */
  int sent_final;           /* Have we sent a final response to parent? */

  pthread_cond_t response_received; /* alert to messages on the queue */

  int marked_done;          /* has the parent noticed that we finished? */
  int shutdown;             /* Have we shut down our state machine? */

  int sent_something;       /* do we have something interesting to
                             * wake the parent up about? */

  void *user_policy_info;    /* User-policy-specific info for the branch */
  void *transaction_policy_info; /* Transaction-policy-specific info for the
                             * branch */

} branch_t;
```

Figure 25: branch_t: client branch

`response_received` condition variable of the client branch. The client branch can then process the response message from its message queue. `marked_done` is a boolean variable that is used to tell the client branch state machine whether the parent transaction object's state machine has noticed the completion of the client branch state machine. If the client branch terminates and releases memory before the parent transaction object is notified, then it may lead to illegal memory access problems. This is why we need the `marked_done` variable. The `shutdown` boolean variable asks the client branch state machine to terminate itself. This is signalled by the transaction state machine.

- `sent_200` and `sent_final` have the same meaning as that in the transaction object. `sent_final` indicates whether the state machine has sent a final response to the parent transaction object. Similarly, `sent_200` tells whether it has sent a 2xx-class final response to parent transaction object.
- The original SIP request for this branch is stored in `orig`.
- The URI of the downstream server (which was obtained either from the database lookup in a proxy transaction or supplied by the application in an outgoing user agent transaction) is stored in the `uri` field.

10.3 Receiving messages

Fig. 26 shows what happens when a message is received. The `tcp` and `udp` sub-modules (as part of the `libcine` library) take care of starting the threads to listen for incoming messages as shown in

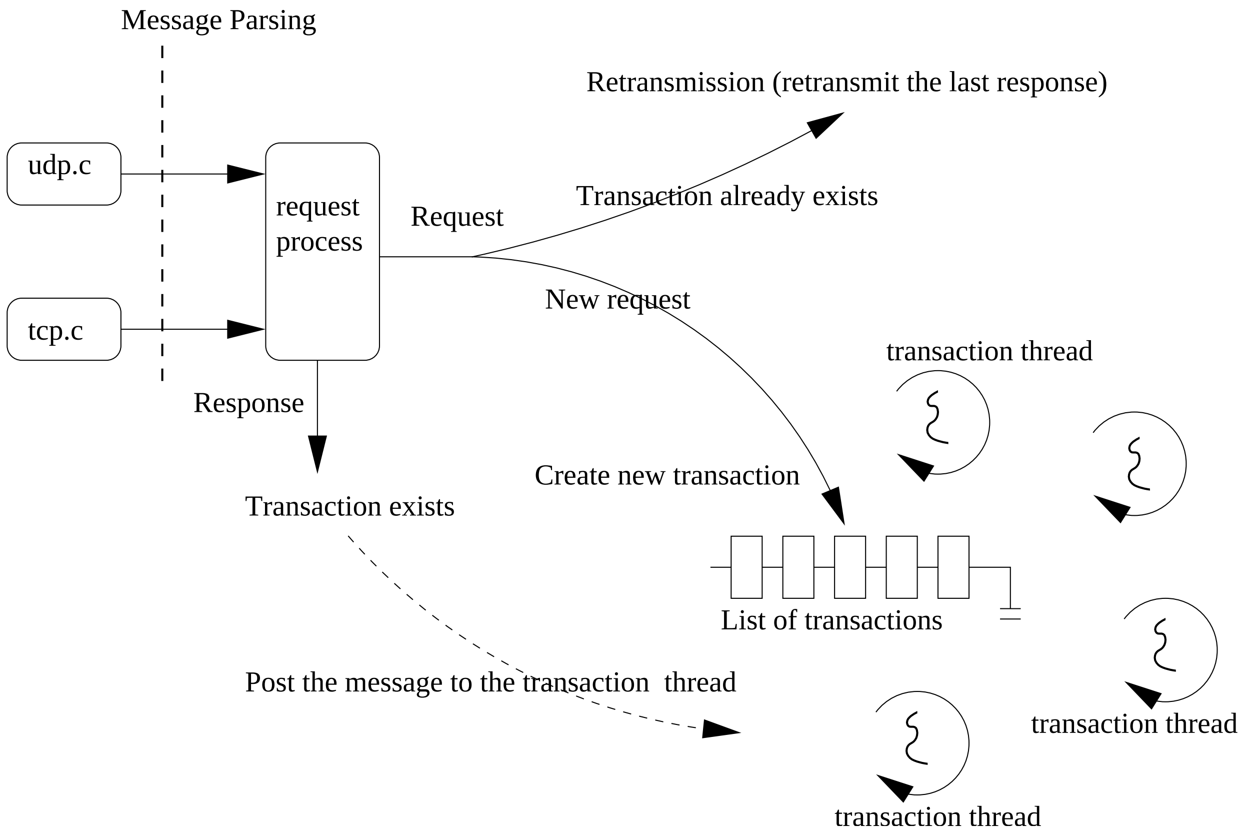


Figure 26: Handling an incoming message

Fig. 27 and 28. When a message is received it is parsed into a `parser_t` structure and a transaction object is created out of it.

```
ReceiveTCP   -- a thread
{
   Create and bind a TCP socket at port 5060
   While true  -- infinite loop
   {
      Wait for incoming connection.
      Invoke ReceiveTCPRequests thread
      on incoming connection.
   }
}
ReceiveTCPRequests -- a thread
{
  Receive and parse the incoming message using HTTP_Parse().
  Create a transaction object, request_t.
  Invoke  RequestProcess() on this object.
}
```

Figure 27: tcp.c: TCP receive thread

The transaction object is different for different protocols, SIP and RTSP. The multiplexing is done in the header files. Basically, the fields needed by both SIP and RTSP are put in the common

```
ReceiveUDP -- a thread
{
   Create and bind an UDP socket at port 5060
   While true -- infinite loop
   {
      Wait for incoming UDP message.
      Parse the incoming message using HTTP_Parse().
      Create a transaction object, request_t.
      Invoke RequestProcess() on this object.
   }
}
```

Figure 28: udp.c: UDP receive thread

file (`request-common.h`) that is shared by the transaction objects for both SIP (`request-sip.h`) and RTSP (`request-rtsp.h`).

Once the transaction object is created, the `RequestProcess` function is invoked to process the appropriate message. It invokes either `RequestProcessThread` or `ResponseProcessThread` based on whether the message is a request or response.

The library maintains a list of currently active transactions. If the received message is a response, then `ResponseProcessThread` searches for the corresponding transaction objects from the list and forwards the response to that transaction object. To be more precise, it forwards the response to the transaction's client branch associated with that request. Since the client branch id is also sent in the Via header to the downstream servers and the response from the downstream servers contains the same Via header and the branch id, any response can be readily mapped to the appropriate client branch. It is up to the *client branch state machine* to handle the response as appropriate. If no corresponding transaction object is found in the list, then the response is ignored.

If the received message is a request, then the `RequestProcessThread` function does the following things.

ValidateIncomingRequest: check the validity of the request. This includes checking for the presence of the SIP method, protocol, version, supported URI ("sip" and "tel"), and mandatory headers like From, To, Call-ID and CSeq. If the validation fails, then an appropriate failure response is sent to the upstream client. The library support only "sip" and "tel" URIs. Also if the Require or Proxy-Require (depending on whether the request has to be handled locally or not; for example for a proxy and registrar server REGISTER is handled locally but INVITE is not) header is present and is not supported, then the appropriate failure response is sent.

RequestIsNotLooped: Check if this request has been looped. If our proxy name appears in its Via headers, and the hash part of that Via header's branch-id corresponds to this transaction's hash then it means that the request was sent by this proxy and came back to itself. If a loop is detected then the appropriate failure response is sent back to the upstream client.

RequestSearch: The library maintains a list of currently available transaction objects. Whenever a new request is received, it looks up into the available transactions list for a match. If it belongs to the existing transaction the request is forwarded to the state machine of that transaction, otherwise a new transaction state machine is entered. The search is done by comparing the request URI, From, To, CallID and CSeq fields. The request hash is used to speed up the search.

HandleDuplicateRequest: If the request belongs to an existing transaction, then `HandleDuplicateRequest` explores various possibilities as shown below:

```
If top Via of this request does match the original request
  We have request merging.
  Send 482 unless the request is ACK, since there is no response for ACK.

If the request is ACK
  If we have already sent 2xx response
    This request is probably being handled by the proxy server's policy
    and has already sent 200. So this ACK needs to be handled end-to-end.
    Hand it off to the proxy server's policy or state machine.
  Otherwise
    Our state machine is currently retransmitting the final responses.
    Set the appropriate flag (acked) so the retransmission stops.

If the request is CANCEL
  Forward the message to the appropriate transaction state machine.
  The 200 response to CANCEL is sent no matter what happens in the
  transaction state machine.

If the request is something else say
BYE, INVITE, OPTIONS, REGISTER.
  This is probably a retransmission.
  Send back the previous final response and clean up this request.
```

HandleNewRequest: If no matching transaction is found, then `HandleNewRequest` does the following things:

```
If the request is ACK
  ACK is for unknown transaction
  Ignore it for proxy. (Not for user agent)

If the request is CANCEL
  Respond with 481 Call leg does not exist

For all other requests
  Invoke the application (proxy, user agent) specific processing
           SIP_OnIncomingRequest().
```

Note that a proxy transaction can ignore the ACK if it does not match an existing transaction but for an user agent transaction it is not always possible. The ACK might have been sent end-to-end, in which case the request URI is different from the original request. So one has to compare again the incoming message with the available requests using only Call-ID, CSeq, From and To, and not using the request URI.

The application specific processing of any new request is done in the `SIP_OnIncomingRequest` function. For example, a SIP proxy and registrar server will handle REGISTER requests differently from all the other requests. The REGISTER request requires updating the local database to update the contacts whereas other requests are proxied (or redirected) based on the contact locations in the database for the user. Our proxy server, sipd, has a built-in presence server that handles REGISTER and SUBSCRIBE differently from all the other requests as shown in Fig. 29.

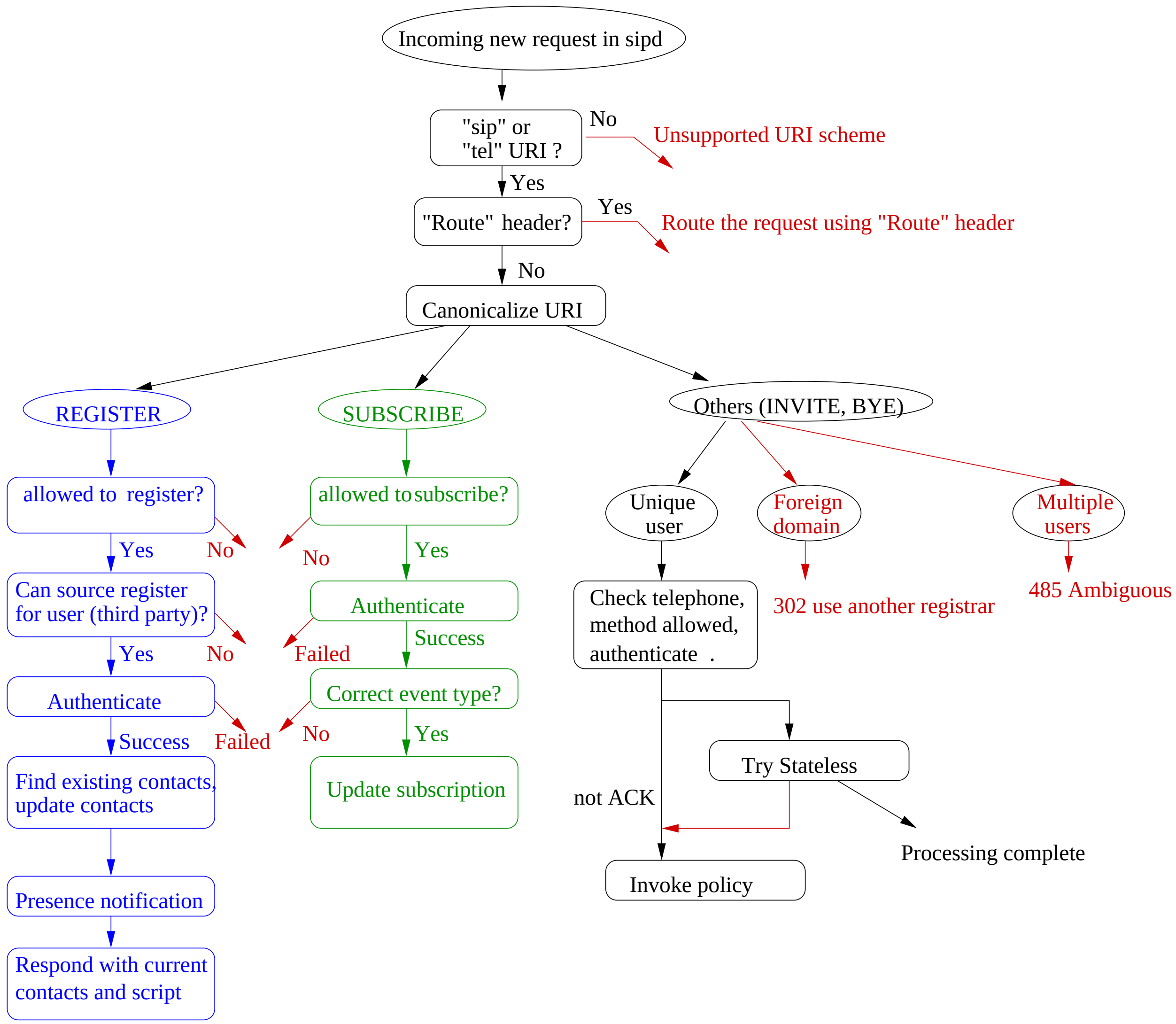


Figure 29: Incoming request in sipd

Functions like `SetEarlyRequestData` and `SetLateRequestData` are used to populate the transaction object fields with the appropriate values from the parsed incoming message.

10.4 Incoming registration

Fig. 29 shows processing of an incoming REGISTER method in sipd. If the user is not allowed to register, then the request is rejected. If the source address and the destination address are different, then it is a third-party registration scenario. If the source is not allowed to register for the destination, then the request is rejected. The request is then authenticated using the digest authentication. The server updates the contact locations in the database based on the new request. If a contact is expired, it is removed. If the REGISTER message explicitly sets the Expires header as 0, then also the contact(s) are removed. If any user has subscribed to receive the presence event for the registering user, then corresponding NOTIFY request is sent to that subscribed user. The server then responds back with a successful response containing the existing contacts and programmable script (SIP-CGI), if any.

10.5 Policy architecture

The SIP library provides a very easy-to-use architecture to plug in various independent components in a generic architecture. This includes the proxy server, user agent, redirect server and presence server. Note that the policy is used to define the behavior of the transaction state machine. It may not always be used. For example, a SIP server may not assign a policy to incoming REGISTER requests, but may assign proxy or redirect policy for all other requests. The decision as to which policy to use for a particular request depends on the type of the application (proxy server or user agent), and/or user preference (user may have configured the server to use redirect mode instead of proxy mode).

A policy can be started using `execute_policy` function call.

```
void execute_policy(request_t *r, policy_info_t *reg, int reg_status)
```

The transaction object, `request_t`, is passed as the first parameter. The second parameter is specific to the particular policy. It typically stores the input needed for taking further action, e.g., the available registration contacts for the user in case of proxy policy so that the state machine can proxy the request to those contact locations. The third parameter is the SIP status code to start the policy with. A status code of 200 means everything is fine. In some cases, the policy is started with a non-200 status. For example, a failure policy may be started with status "404 Not Found".

The life cycle of a transaction or a policy is shown in Fig. 30. The processing handlers, init,

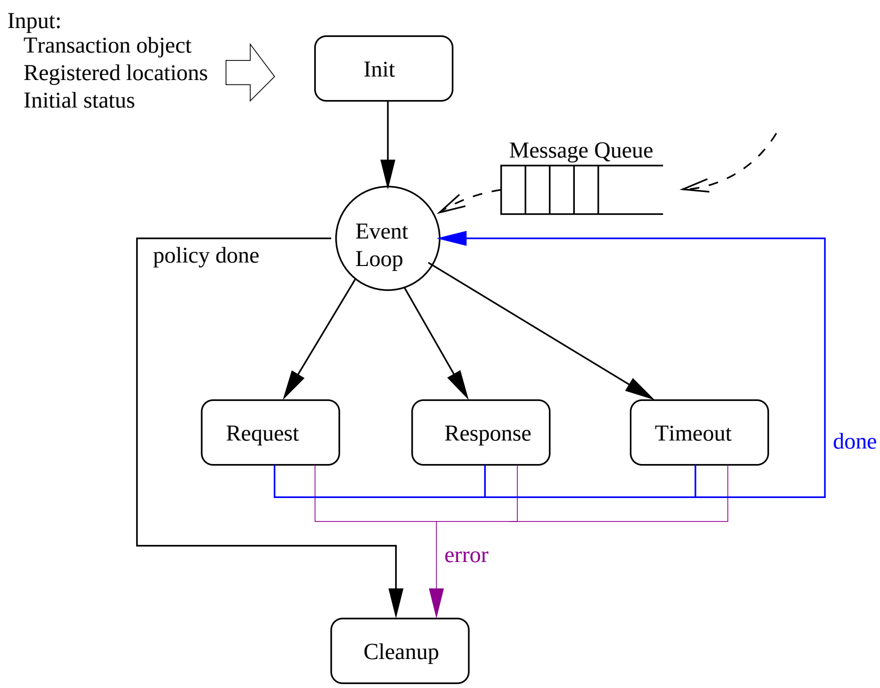


Figure 30: Life cycle of a policy

request, response, timeout and cleanup, are defined by the specific implementation of a policy.

The `execute_policy` function implements the overall state machine for the transaction. It does not terminate until the transaction object is destroyed. After the function exits, the transaction object is no longer valid and must not be referenced. Since the calling thread may remain inside this function for long time, it is recommended to use a separate thread to invoke this function if one does not want to wait or block in the caller's thread.

As mentioned earlier, the transaction object has a message queue for inter-thread synchronization. Other modules (e.g., client branch state machine, or call control state machine) can send their messages to this message queue. The `execute_policy` thread is also called as the *policy thread* or the *transaction thread*.

When a policy is created, the init function is called first. Then the thread waits on an event in a loop. This event is signalled when something "interesting" happens like a message is sent to the thread's message queue or a timer expires. Depending on the type of the event, the appropriate function, request, response or timeout, is called. These handlers should not block or wait while processing the event or message. Typically, an handler function does some minimal processing, updates state, sends message(s) to other modules or to remote and returns. After the function completes, the thread again waits on the next event. In case of an error in processing of these functions, or when the policy thread is done its work (signaled by setting a boolean variable done), the cleanup function is called and the policy terminates.

We have implemented the following policies. Some of these are shown in Fig. 23.

Redirect (redirect) This is implemented by sipd. It returns a 300-class response to the upstream client in the init stage and terminates immediately.

Proxy (proxy) This is implemented by sipd. It creates new client branches based on the registered locations in the init stage. The subsequent requests are processed in the request stage whereas the subsequent responses received from the client branches are processed in the response stage. timeout stage is needed to indicate any timer expiry in proxying. For example, if the more preferred contact (with higher q value) did not respond, then the lower priority contact location needs to be used after a timeout.

Failure (failure) A failure policy simply responds back with the appropriate failure response 4xx-6xx in the init stage.

User Agent (user) This is implemented in the libsip++ library. There are two possible types of user transaction: outgoing and incoming. The outgoing transaction has client branches similar to the proxy policy. The incoming transaction does not have a client branch.

10.6 Client branch - state machine

A client branch is similar to a UAC. Client branches are created when the transaction needs to send a request to the remote UAS. The client branch logic takes care of receiving the responses, performing retransmissions and handling timeouts. A single transaction object can have one or more client branches (e.g., in the forking proxy behavior) or it can have no client branch. This subsection describes the state machine for a single client branch.

Fig. 31 shows the state machine for the client branch. The state machine is based on the client transaction state machine specified in SIP [12]. When a new client branch is created (in the Initial state) it tries to find out the IP address to send the request to. If the location is not found, it immediately sends back a "404 Not found" to the parent transaction object. If it is found, then it sends the SIP request to that address and moves to the Retransmitting request state. In this state it retransmits the request as per the SIP specification. Awaiting response state is reached when it has received a provisional response in Retransmitting request state. The retransmission stops only for INVITE request. For other messages the timeout of the retransmission is changed but the retransmission continues. Various timers in different states are shown in Table 9. Events can be responses (1xx-6xx) received from the downstream servers, timer expiry, cancellation or shutdown of the branch by the transaction layer, or some other error. The client branch thread is always

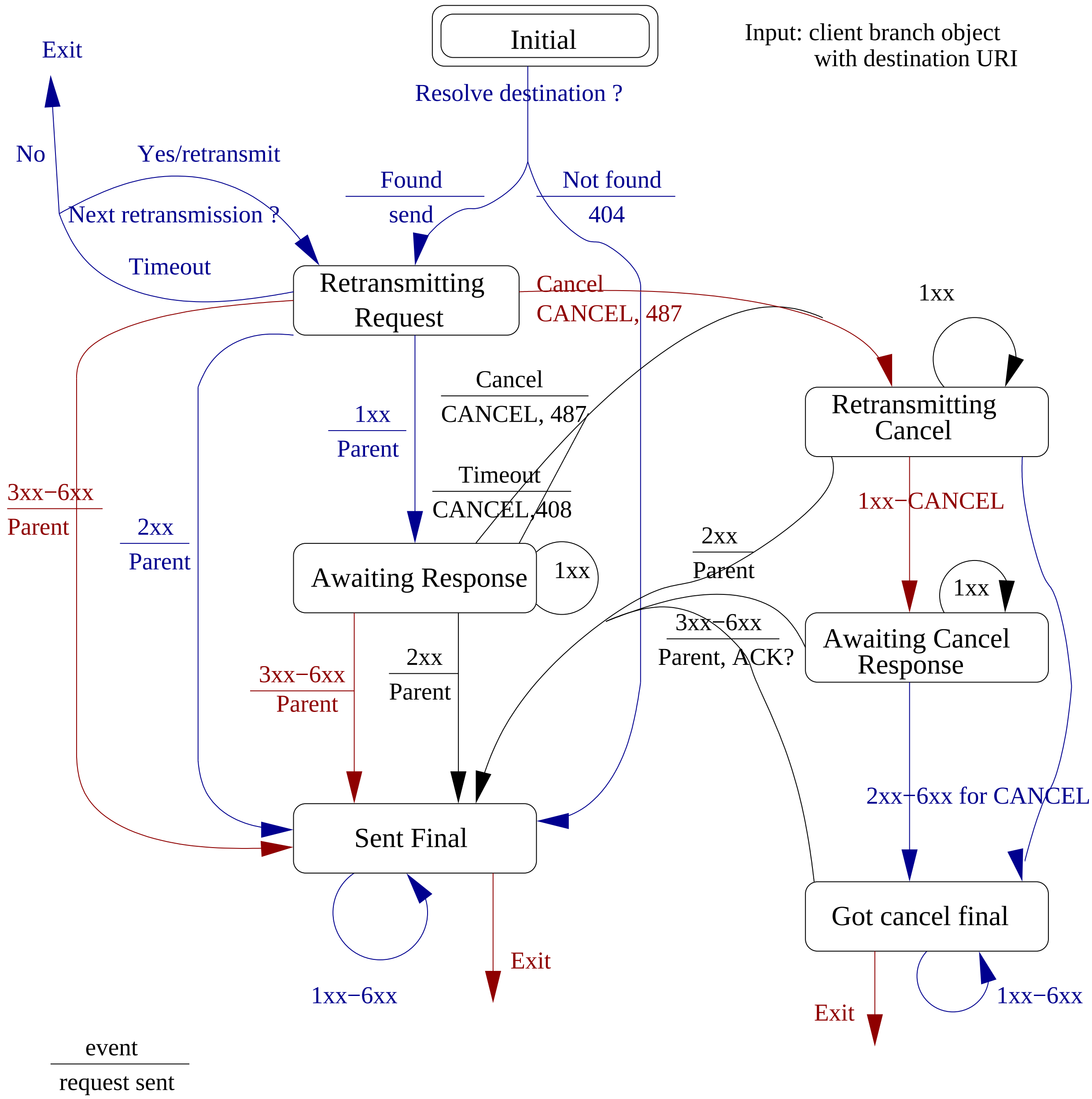


Figure 31: SIP client branch state machine

immediately terminated when a shutdown is sent by the transaction layer. On the other hand, the cancellation of the branch calls for sending of the SIP CANCEL request to the downstream servers. Most of the downstream responses are forwarded to the *parent* transaction thread, which inturn may forward it to the upstream client. The client branch can send either a real response (e.g., when the actual SIP response message is received from the downstream server) or a pseudo response (e.g., in case of a socket call failure to send the message, the client branch indicates a pseudo response of "404" to the transaction state machine, which in turn formats the real response message and sends it to the upstream UAC).

10.7 Stateful proxy

A stateful proxy is implemented using the proxy policy. This section describes the implementation of this policy. Fig. 32 shows the state machine for the proxy transaction.

State	description	timer
Initial	Initial state. As soon as the client branch is started it transitions to retransmitting request or awaiting responses state.	None.
Retransmitting request	The system is retransmitting the original request to the downstream server.	Time to the next retransmission.
Awaiting responses	It is awaiting a final response frm the downstream server.	If the request was an INVITE then the timer is set to Expires value or infinite if it is absent. For any other request it is set to Expires.
Sent final	It has sent a final response to the upstream server. It is probably waiting to be terminated by the transaction layer.	Infinite timeout.
Retransmitting cancel	The client branch was cancelled by the transaction layer. We are retransmitting the CANCEL request to the downstream server.	Until next cancel retransmission.
Awaiting cancel response	We are waiting for a final response for the CANCEL from the downstream server.	T2 (slow retransmit) or infinite afterwords.
Got cancel final	We got the final response for the CANCEL request. but the final response for the original request was not received.	Infinite timeout.

Table 9: SIP Client Branch States

proxy_init When the proxy policy is created, it starts one or more client branches based on the preferences (or q value) of the different contact locations. Client branches for most preferred location are created first. If there are some more locations (with lower preferences) that are not yet tried, then the system goes into the sending request state. If there are no more locations to try then it goes into the awaiting responses state. A timer is started here with a default timeout of 30 seconds. Once the client branches are created, it is the reponsibility of the client branches to send the request and handle all the retransmissions and timeouts.

proxy_handle_request The state machine can get either a CANCEL or an ACK request from the upstream client for this transaction. If a CANCEL is received, then all the unfinished client branches are cancelled and the system goes into the final wait state. An ACK is forwarded to all the finished branches.

proxy_handle_response The transaction state machine can receive the responses from the client branches. These responses can be the real SIP responses or pseudo responses (e.g., on socket error while sending the message). Choosing the next state is tricky here. If the response was a successful 2xx class response, then the system goes to the final wait state immediately and all the other branches are cancelled. However if the response was some other final response then the system needs to wait for the other client branches to finish. If no further client branch response is expected, but there are some more locations that have not yet been tried (i.e. the system is in sending request state), then the next preferred location(s) are tried by creating one or more client branch(es). If there are no more pending responses from any client branch and there is no more location to be tried, then the system selects the best response from all the received final responses. In case of a single location proxy, there will be only one response. But in case of a forking proxy there could be multiple responses. A 3xx-class

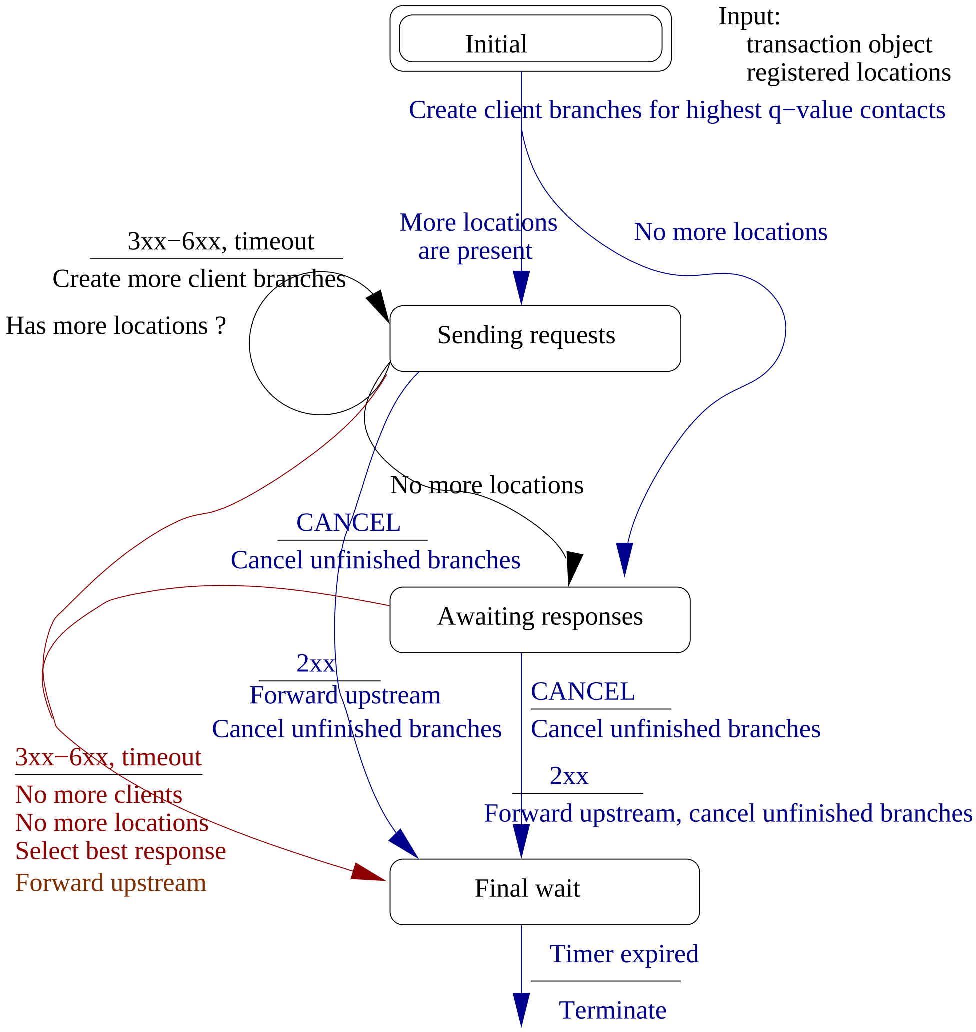


Figure 32: Stateful proxy policy

response is better than 4xx, which in turn is better than 5xx. On the other hand a 6xx response should cancel all the other pending branches, terminate the search and return that failure response immediately to the upstream client.

In the final wait state the system waits for some time before exiting the policy thread (default is 30 seconds). This waiting time is also explained in the SIP specification and is used so that the state machine can respond to any retransmissions.

proxy_timeout A timeout is handled differently in different states. In the `sending request` state, a timeout causes creation of more client branches, if possible, or the termination of the transaction due to the timeout. In the `awaiting responses` state, a timeout indicates that the location could not be contacted, the transaction fails and goes to the `final wait` state. If the system is already in the `final wait` state when the timer expires, the transaction is considered to be terminated by signaling the boolean variable `done`, which in turn terminates the event wait loop of the policy thread and cleans up the transaction object.

proxy_cleanup All internal policy related structures are freed during cleanup.

Note that the transaction state machine waits in the `final wait` state for some time before the transaction object is terminated or deleted. This feature is undesirable in the case of a stateless proxy that is described next.

10.8 Stateless proxy

A stateless proxy requires that there is no state maintained per transaction by the system. A stateless proxy is possible only for an UDP based signaling because the TCP by definition is stateful. Also, a stateless proxy must not have any retransmission or request forking logic.

We have implemented a stateless proxy in sipd. How an incoming message is handled differently in the stateless proxy is shown in Fig. 33. Compare this with Fig. 26. The decision whether to

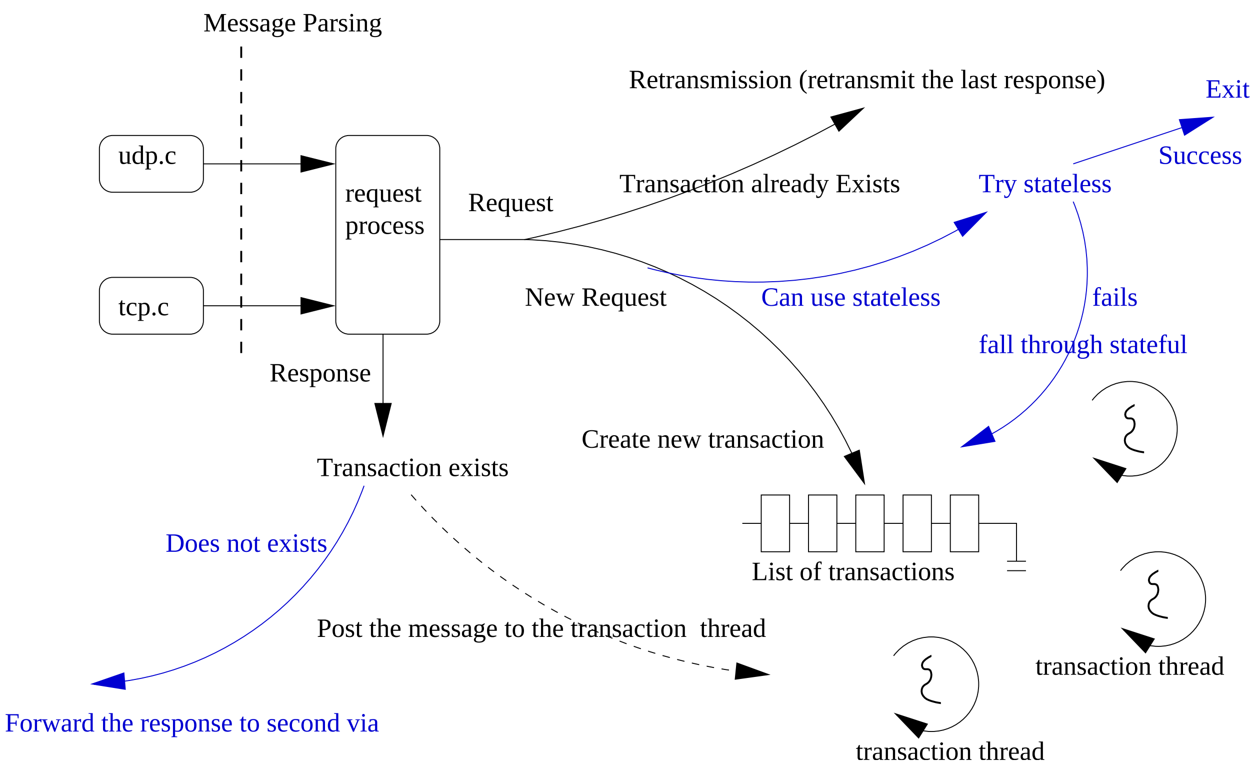


Figure 33: Stateless vs stateful proxy policy

proxy the request statelessly is done dynamically on a per-request basis. The stateless proxy mode is choosen only when both the upstream and the downstream signaling is over UDP, and there is only one contact location for the request URI (i.e., no request forking). If the stateless proxy procedure fails, then the system uses the default stateful proxy procedure.

Since there must not be any state and the policy architecture is stateful, we cannot use the existing policy architecture. The stateless proxy is layered parallel to the policy architecture (and not above it). When a new request is received and the server decides to use the stateless proxy, it forwards the request to the selected contact location and terminates the transaction object and other associated thread(s). Similarly, when a response is received and no associated transaction is found for the response then it is forwarded statelessly to the second Via header. The top Via is this server's address whereas the second Via is the address of the upstream client.

10.9 User agent library

A user agent library, libsip++, is built on top of the transaction and client branch architecture. It is implemented as a policy called user. As mentioned earlier, there are two types of user agent transactions: outgoing and incoming. The state machine for the outgoing transaction is similar to the proxy policy except that it handles only one client branch. Incoming transaction's state machine is more simple as there is no client branch. The user agent library primarily focusses on the following two things:

Call control: A SIP/SDP multimedia call signaling is implemented using the state machine shown in Fig. 34. The library maintains a list of the active call objects (`sipcall_t`). Every call object

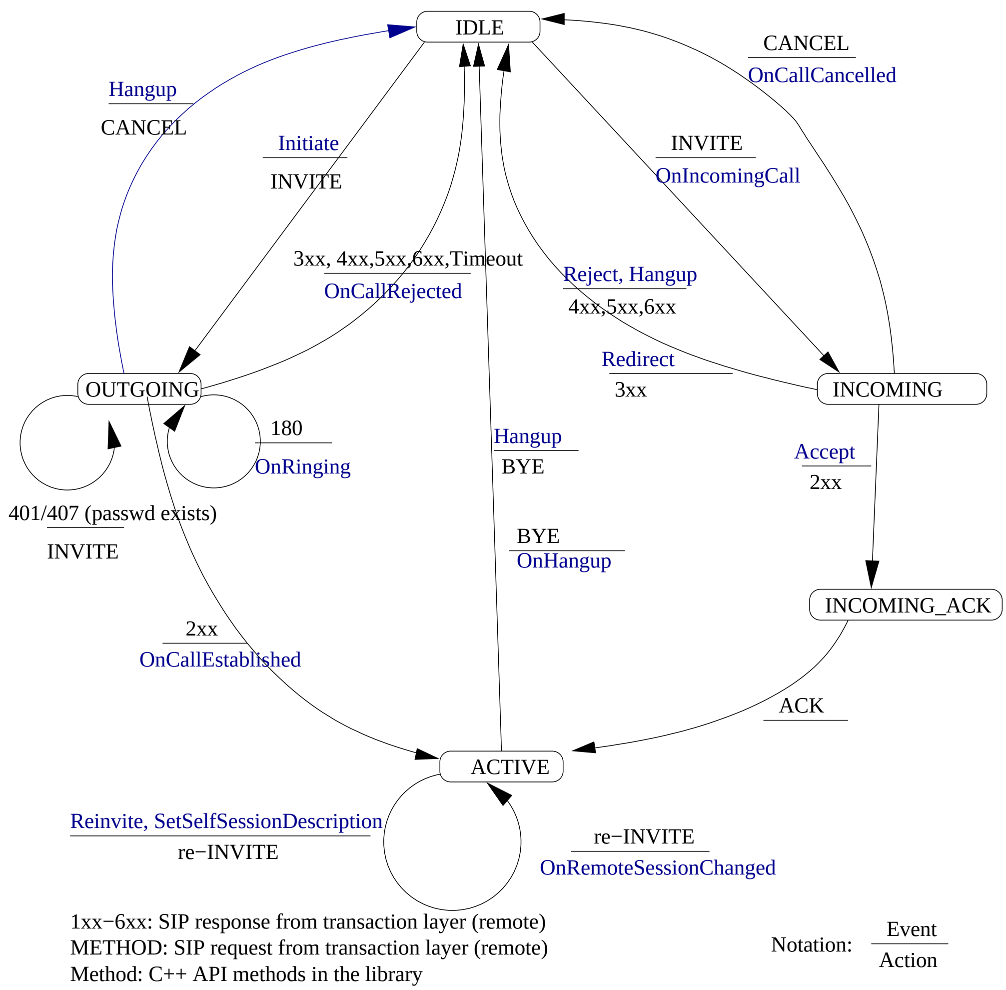


Figure 34: call control state machine

is associated with a higher level C++ object (`SIPCall`) defined by the libsip++ API. The state machine processing is implemented in C, but the API is in C++. The call thread has a message queue, similar to the transaction thread's message queue. Communication with the C++ API functions or the transaction layer happens through this message queue. Whenever an API function is called to alter call state (e.g., hangup or initiate a call), a message is posted to the message queue. Similarly when the transaction layer has something interesting (e.g., transaction cancelled by remote) to inform to the call control layer, it posts a message to the message queue. The call thread invokes the appropriate C++ API call-back methods to indicate something interesting to the application, e.g., a new incoming call. The state machine handles the SIP INVITE, ACK and BYE methods.

Outgoing registration refresh: The user agent library can perform outgoing registrations to the remote SIP registration servers like sipd. A single one time registration does not need any state machine, but if one wants to implement an automatic registration refresh mechanism,

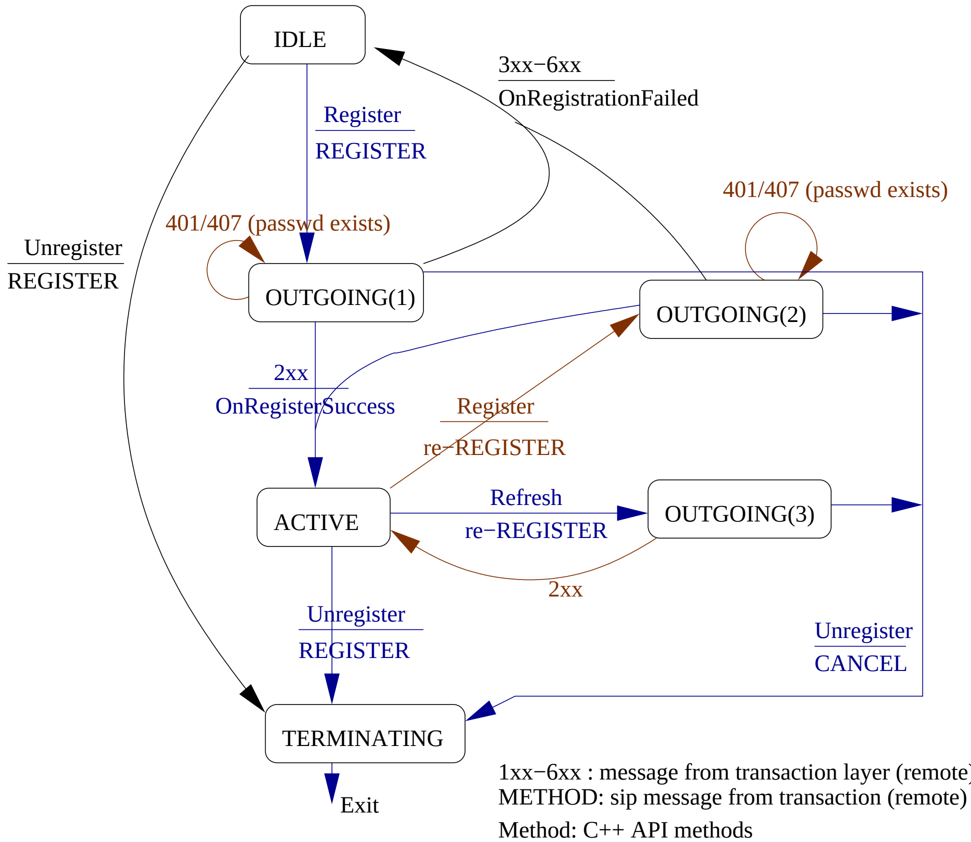


Figure 35: Outgoing registration state machine

the state machine shown in Fig. 35 is useful. The state machine handles the outgoing SIP REGISTER method for an user agent.

The user agent library also has functions for parsing the SDP message body. However the media transport or encoding and decoding of the multimedia data is outside the scope of this library.

A similar state machine can be implemented for other SIP methods like SUBSCRIBE and NOTIFY. Most of the other methods (e.g., DO, MESSAGE and REFER) do not require any state machine. We are working on abstracting the addition of a new method in the library so that it is easier for a third party to plug-in a new implementation for any SIP message.

The user agent library is meant for implementation of user agent type of applications. This may include, besides a traditional user agent, a conferencing server, an unified messaging system and a signaling gateway. sipconf, sipum and sip-h323 are the examples of these applications in our test-bed.

10.10 Thread synchronization

Multi-threading helps modularity. It is easier to implement a state machine as an independent thread. For every transaction a new thread is created. Moreover every client branch has its own thread. As an example, a simple proxy request to one location will generate two threads, one for the transaction state machine and the other for the client branch state machine.

We use the POSIX thread (`pthread`) API. The mutex and condition variables are used for synchronization. Typical pseudocode use of these constructs is shown in Fig. 36. Typically, every transaction object has a mutex and a condition variable. If some thread wants to access the object, then it must acquire the mutex lock before doing so. All the client branches associated with a

```
...
//event loop
pthread_mutex_lock(&r->mutex); // get the lock
while ( 1 ) {
  //set timeout if applicable
  e = pthread_cond_timedwait(&r->cond, &r->mutex, &timeout);
  if ( e = TIMEOUT ) {
    //process timeout
  }
  else {
    //get message from message queue and process
  }
}
pthread_mutex_unlock(&r->mutex); //release the lock

...
//singnaling code
pthread_mutex_lock(&r->mutex); //get the lock
//send some message to the message queue
pthread_cond_broadcast(&r->mutex); //signal the event.
pthread_mutex_unlock(&r->mutex); //release the lock
...
```

Figure 36: Example on mutex and condition variable

transaction use the mutex of that transaction object. The transaction event wait loop waits on the condition variable of the transaction object. Any other thread that wants to signal the transaction logic should signal this condition variable. For example, when a new message is received from an upstream client, the message is forwarded to the transaction thread's message queue and the condition variable is signaled. Signaling the condition variable of the thread causes it to come out of the event wait loop, and process the messages from its message queue. However, this happens only when the thread acquires the transaction's mutex lock.

A client branch also has a condition variable. This is used by the response thread to trigger the event that a new response has been received from the downstream server. The client branch signals the condition event of the associated transaction object when it wants to send some event or message to the transaction thread.

The user agent library is implemented above the transaction layer. In particular, the call control or registration refresh threads need to communicate with the transaction thread. Every call control or registration refresh thread has a mutex for exclusive access, and a condition variable for signaling events.

The list of currently active transaction object is maintained as a global hash-table, protected by a global mutex, R_lock. Similarly, the list of active calls and list of registrations are maintained as global linked-lists.

A single thread may need more than one mutex lock. This might result in a deadlock. We prevent any possible dead-lock by avoiding the circular wait condition. The various mutexes are ordered in a partial-order as shown in Fig. 37. Any thread which needs to acquire the mutex with a lower order should release all the higher ordered mutexes. For example, any thread which wants to acquire the R_lock mutex should first release any transaction object's mutex because R_lock is ordered before any transaction object's mutex. Similarly, a transaction object's mutex is ordered before the associated call control object's mutex, so one must release any call mutex in the user agent library, before trying to acquire the transaction object's mutex. All other mutexes that do

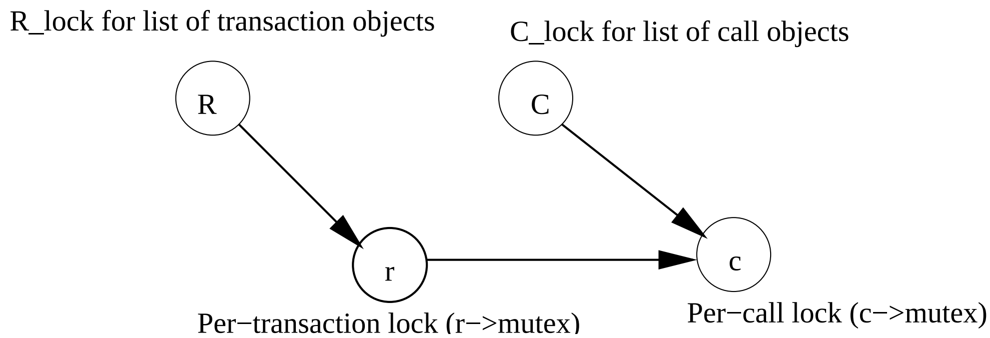


Figure 37: Ordering of the mutexes to avoid dead-lock

not appear in Fig. 37 are independent of other mutexes.

10.11 Database lookup

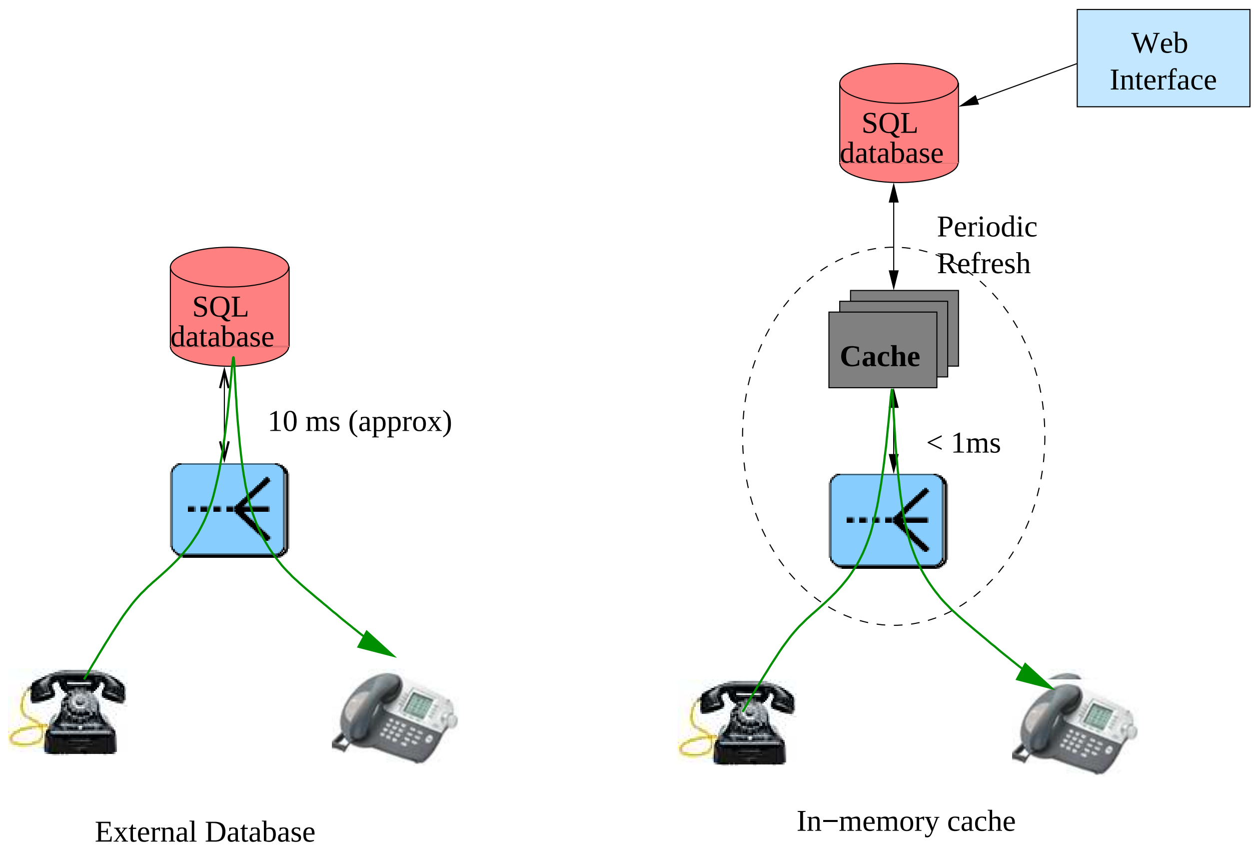


Figure 38: SQL vs FastSQL

Database lookup for locating the contacts of the users constitues a substantial fraction of the processing power in a SIP proxy server. Higher delay in database lookup (approximately 10 ms per query) increases the response time/delay of the transaction, hence the performance and the scalability. We have implemented an in-memory database scheme to speed-up the database access time in our SIP server (sipd) as shown in Fig. 38. This involves loading the various database tables (e.g., user information, contact locations, aliases) into the main memory, instead of doing lookup into the database for every transaction. Since each table entry takes less than few hundreds of bytes it is perfectly reasonable to use it in an enterprise environment with only a few thousands of users for improved performance. However this optimization causes another problem related to synchronization of the in-memory and external database. In particular, care must be taken to updated the in-memory database when a new contact is added from the user interface. We define a periodic refresh interval (About two minutes for contacts table and half an hours for user information and aliases tables) to refresh the in-memory database. The contacts table is written out to the external database from in-memory database periodically. We read only those entries that are modified since last read and write only modified entries back to the database during refresh.

11 Related Work

Several companies, such as Net2Phone and MediaRing, provide PC-to-PC and PC-to-phone calls. Their objective is mainly to provide low-cost call service to the PSTN from the public Internet. We initially intended CINEMA to minimize telephone infrastructure and service costs for an organization, but out architecture is well suited as an Internet telephony infrastructure within an organization. We can configure CINEMA to carry calls between campuses or branch offices over IP with virtually no added cost.

Several multimedia conferencing products use SIP or H.323 for signaling. These include MeetingPoint from CUseeMe Networks [5], Sametime from Lotus[6], and GnomeMeeting [7] from the Linux community. Our system can provide services beyong standard video conferencing and can actually incorporate these tools as long as they are standard-compliant.

There is a fair amount of early voice-messaging work, in particular, Xeroz PARC's Etherphone, but none of it addresses the integration of Internet telephony with voice messaging systems. Several vendors offer SIP proxy servers and user agents that we can use in the CINEMA infrastructure as well. The SIP standard simplifies integration among different vendors' products. The CINEMA architecture facilitates such integration with readily available features; new components can be added as needed.

[5] www.cuseeme.com

[6] www.lotus.com/home.nsf/welcome/sametime

[7] www.gnomemeeting.org

12 Conclusions and Future Work

We have described the architecture of our Internet telephony installation consisting of the SIP server, SIP-PSTN gateway, RTSP media server, unified messaging server, conferencing server and SIP-H.323 translator.

The test-bed is initially intended for small scale experiments within the department and later to be extended to a campus-wide Internet telephony environment. A similar architecture can be deployed at other campus and organization networks who want to benefit from the services provided by Internet telephony, in particular SIP.

We will continue with integration of additional services. For example, SIP-based instant messaging and presence will allow a standard way to send instant messages and form buddy lists [24, 25]. Combining presence and Internet telephony offers improved services, reducing, for example, the number of failed call attempts or involuntary redirects to voice mail.

We have implemented device control using the SIP method DO in our user agent, sipc.

We are implemementing a VoiceXML [40] browser that allows us to easily implement services such as retrieving email, including voicemail, via traditional phones, but also simplifies the task of building voice menus, making such services available to small organizations. (VoiceXML is an XML DTD that mimics HTML forms input via DTMF or speech recognition.)

We are currently instrumenting our proxy and conference server to better understand how to build highly scalable systems. The performance evaluation of SIP servers is more difficult than that of, say, web servers since finding the maximum operating rate is complicated by SIP's use of UDP, causing packet loss and retransmissions under overload. Also, the workload is likely to differ dramatically between registration-bound mobility service and script-processing-bound service engines.

A commercial deployment involves many other issues related to security, billing and quality of service. Interworking with the corporate firewalls and Network Address Translators (NATs) is another challenge. We are also planning developing a Windows CE version of our SIP UA, making it possible to integrate wireless PDAs into the infrastructure.

Finally, a short-term goal is to deploy the system throughout the Computer Science department and then be able to replace our PBX. In the long term, we will provide direct SIP services for integrated access, and address quality of service issues.

13 Acknowledgments

We would like to thank Enlai Chu of Cisco, Keane Chin of SIP Communications Inc., and Sarmistha Dutta of Columbia University for their help with the Cisco gateway and departmental PBX. Xiaotao Wu implemented sipc. Tarun Kapoor installed the MySQL database. Xu Li added the ENUM support to sipd. Huwei Zhang implemented conference load balancing and file sharing. Timo Ohtonen incorporated IPv6 support. Michael Castleman implemented the anonymizer. Li Liao helped in TLS configuration. Anshul Kundaje added RADIUS accounting to sipd. Ali Khwaja worked on G.722 support and quality improvement in sipconf.

This work was supported by equipment and research grants from 3Com, Cisco, Clarent Communications, Lucent, Pingtel, SIPcomm, and Sylantro.

14 Glossary

AAA	Authentication, Authorization and Accounting
ACD	Automatic Call Distribution
ACL	Access Control List
AMI	Alternate Mark Inversion
API	Application Program Interface
B2BUA	Back-to-Back User Agent
B8ZS	Bipolar 8 with Zero Substitution
CAS	Channel Associated Signaling
CCS	Common Channel Signaling
CDP	Coordinated Dialing Plan
CGI	Common Gateway Interface
CINEMA	Columbia InterNet Extensible Multimedia Architecture
CPL	Call Processing Language
DID	Direct Inward Dialing
DNS	Domain Name System (or Service or Server)
DSEL	Data SELector
DTD	(XML) Document Type Definition
DTMF	Dual-Tone Multiple Frequency
ENUM	Telephone Number Mapping
ESF	Extended Super Frame
H.323	ITU-T recommendation for multimedia communication over packet based networks
HTTP	Hyper-Text Transport Protocol
MD5	Message Digest version 5
IETF	Internet Engineering Task Force
IOS	(Cisco) Internetwork Operating System
IP	Internet Protocol
IPv6	IP version 6
ISDN	Integrated Services Digital Network
ISP	Internet Service Provider
ITU-T	International Telecommunications Union - Telecommunication standardization sector
IVR	Interactive Voice Response
MIB	Management Information Base
NAT	Network Address Translator
NCOS	Network Class Of Service
PBX	Private Branch eXchange
PCM	Pulse Code Modulation
POTS	Plain Old Telephone Service (also PSTN)
PRI	Primary Rate Interface
PSTN	Public Switched Telephone Network
PUT	Primary User Table
QoS	Quality of Service
RADIUS	Remote Authentication in Dial-In User Service

RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SDP	Session Description Protocol
SF	Super Frame
SIP	Session Initiation Protocol
SNMP	Simple Network Management Protocol
SRV	(DNS) Service resource record
SQL	Structured Query Language
T1-line	A digital line with 24 channels each of 64 kb/s
TCP	Transport Control Protocol
TLS	Transport Layer Security
TRIP	Telephony Routing over IP
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
URI	Universal Resource Identifier
URL	Universal Resource Locator
VoiceXML	Voice eXtensible Markup Language
XML	eXtensible Markup Language

References

[1] The SIP servlet API. http://www-uk.hpl.hp.com/people/sth/sip/servlet/.

[2] OpenSSL home page. http://www.openssl.org/.

[3] Net-SNMP home page. http://www.net-snmp.org/.

[4] B. Aboba, J. Arkko, and D. Harrington. Introduction to accounting management. RFC 2975, Internet Engineering Task Force, Oct. 2000.

[5] J. Bellamy. *Digital Telephony.* John Wiley & Sons, New York, 1991.

[6] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Introduction to community-based SNMPv2. RFC 1901, Internet Engineering Task Force, Jan. 1996.

[7] F. Dawson and D. Stenerson. Internet calendaring and scheduling core object specification (icalendar). RFC 2445, Internet Engineering Task Force, Nov. 1998.

[8] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999.

[9] P. Faltstrom. E.164 number and DNS. RFC 2916, Internet Engineering Task Force, Sept. 2000.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[11] M. Handley and V. Jacobson. SDP: session description protocol. RFC 2327, Internet Engineering Task Force, Apr. 1998.

[12] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: session initiation protocol. RFC 2543, Internet Engineering Task Force, Mar. 1999.

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

[14] IRT Lab, Columbia University. E*phone home page. http://www.cs.columbia.edu/~hgs/ephone/.

[15] IRT Lab, Columbia University. sipc home page. http://www.cs.columbia.edu/IRT/software/sipc.

[16] A. Kristensen and A. Byttner. The SIP servlet API. Internet Draft, Internet Engineering Task Force, Sept. 1999. Work in progress.

[17] J. Lennox and H. Schulzrinne. CPL: A language for user control of internet telephony services. Internet Draft, Internet Engineering Task Force, Nov. 2001. Work in progress.

[18] J. Lennox, H. Schulzrinne, and J. Rosenberg. Common gateway interface for SIP. RFC 3050, Internet Engineering Task Force, Jan. 2001.

[19] K. Lingle, J. Maeng, J. Mule, and D. Walker. Management information base for session initiation protocol. Internet Draft, Internet Engineering Task Force, Feb. 2002. Work in progress.

[20] MySQL AB Co. MySQL home page. http://www.mysql.com.

[21] S. Petrack and L. Conroy. The PINT service protocol: Extensions to SIP and SDP for IP access to telephone call services. RFC 2848, Internet Engineering Task Force, June 2000.

[22] C. Rigney. RADIUS accounting. RFC 2866, Internet Engineering Task Force, June 2000.

[23] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote authentication dial in user service (RADIUS). RFC 2865, Internet Engineering Task Force, June 2000.

[24] J. Rosenberg et al. SIP extensions for instant messaging. Internet Draft, Internet Engineering Task Force, July 2001. Work in progress.

[25] J. Rosenberg et al. Session initiation protocol (SIP) extensions for presence. Internet Draft, Internet Engineering Task Force, Apr. 2002. Work in progress.

[26] J. Rosenberg, J. Lennox, and H. Schulzrinne. Programming Internet telephony services. *IEEE Network*, 13(3):42–49, May/June 1999.

[27] J. Rosenberg and H. Schulzrinne. A framework for telephony routing over IP. RFC 2871, Internet Engineering Task Force, June 2000.

[28] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. Request for comment, Internet Engineering Task Force, May 2002. RFC 3261.

[29] J. Rosenberg, D. Willis, R. Sparks, B. Campbell, H. Schulzrinne, J. Lennox, C. Huitema, B. Aboba, D. Gurle, and D. Oran. SIP extensions for instant messaging. Internet Draft, Internet Engineering Task Force, Feb. 2001. Work in progress.

[30] H. Schulzrinne. SIP authentication: The null authentication scheme. Internet Draft, Internet Engineering Task Force, Sept. 2000. Work in progress.

[31] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications. RFC 1889, Internet Engineering Task Force, Jan. 1996.

[32] H. Schulzrinne, A. Kundaje, and S. Narayanan. RADIUS accounting for SIP servers. Internet Draft, Internet Engineering Task Force, July 2002. Work in progress.

[33] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP). RFC 2326, Internet Engineering Task Force, Apr. 1998.

[34] H. Schulzrinne and J. Rosenberg. Internet telephony: Architecture and protocols – an IETF perspective. *Computer Networks and ISDN Systems*, 31(3):237–255, Feb. 1999.

[35] H. Schulzrinne and J. Rosenberg. SIP caller preferences and callee capabilities. Internet Draft, Internet Engineering Task Force, Nov. 2001. Work in progress.

[36] K. Singh, G. Nair, and H. Schulzrinne. Centralized conferencing using SIP. In *Internet Telephony Workshop 2001*, New York, Apr. 2001.

[37] K. Singh and H. Schulzrinne. Interworking between SIP/SDP and H.323. In *Proceedings of the 1st IP-Telephony Workshop (IPtel 2000)*, Berlin, Germany, Apr. 2000.

[38] K. Singh and H. Schulzrinne. Unified messaging using SIP and RTSP. In *IP Telecom Services Workshop*, pages 31–37, Atlanta, Georgia, Sept. 2000.

[39] A. Vaha-Sipila. URLs for telephone calls. RFC 2806, Internet Engineering Task Force, Apr. 2000.

[40] VoiceXML Forum. Voicexml home page. http://www.voicexml.org/.

A Our Installation

Fig. 39 shows the system installation in our department. The host names and IP addresses are also shown.

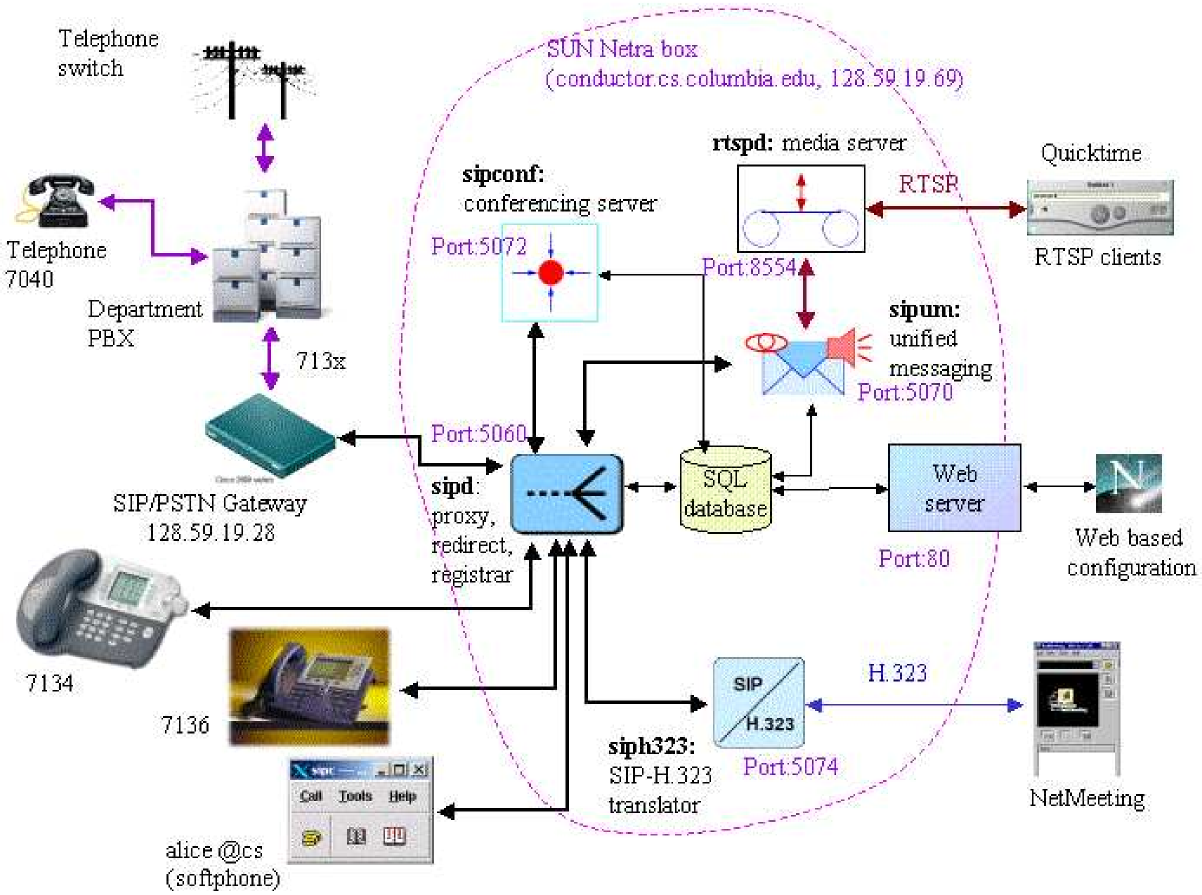


Figure 39: Our installed system

A.1 System configuration

The database tables for cinema, sipd_config and vmail are shown below. The userid and password for SQL database is changed. The SQL URL for accessing the server is of the form `sql://root:NULL@conductor.cs.c` for access from the local host (conductor).

```
# cinema table

server_name     cs.columbia.edu
mode            local
```

```
realm          cs.columbia.edu
administrator  kns10@cs.olumbia.edu
http_server    conductor.cs.columbia.edu
icon           http://conductor.cs.columbia.edu/cinema/images/IRT_logo50.gif
banner         Columbia InterNet Extensible Multimedia Architecture (CINEMA)
mail_relay     ober.cs.columbia.edu
serverroot     NULL
mailrelay      ober.cs.columbia.edu
conffiledir    /opt/cinema/conffile
sipdhost       conductor.cs.columbia.edu
sipconfhost    conductor.cs.columbia.edu
sipconfport    5072
sip323host     conductor.cs.columbia.edu
sip323port     5074

# sipd_config table

server_name      cs.columbia.edu
proxy_name       NULL
primary_ip       NULL
port             5060
domain           ((cs.columbia.edu)|(128.59.19.69))
server_root      /opt/cinema/sipd
error_log        stdout
log_format       %h %u %t "%r" %s To=%{To}i \
                 From=%{From}i Call-ID=%{Call-ID}i %T
log_condition    INVITE,REGISTER,ACK,BYE
group_name       NULL
user             NULL
pid_file         logs/sipd.pid
canonicalize     -d -n -u -D dialplan.sample
gateway_map      gateways.sample
script_base      scripts
auth_method      digest
private_key      1
nonce_life_time  60
clock_skew       300
cgi_timeout      15
expires          3600
threadpool_size  128
foreign_domain   proxy
default_reg      proxy
display_ambigous false
help_resolve     false
proxy_recursion  false
third_party_reg  false
start_snmp       false
try_stateless    false
record_route     true
use_threadpool   true
```

```
enum_root         trial.e164.com
agentx_socket_path 127.0.0.1:5062
max_expires       86400
realm             cs.columbia.edu
timeout           36
numeric_via       true
host_name6        conductor.cs.ip6.columbia.edu
proxy_name6       NULL
primary_ip6       NULL
start_ssl         true
use_namemapper    true
require_userparam false

# vmail table

server_name       cs.columbia.edu
vmailuserdir      /opt/cinema/vmusers
vmailhost         conductor.cs.columbia.edu
vmailport         5070
rtsphost          conductor.cs.columbia.edu
rtspport          8554
randomdigits      6
vmailquota_kb     2000
```

A.2 Cisco 2600 (gateway) configuration

The configuration (`show run`) for the Cisco 2600 gateway is shown below.

```
Current configuration : 2645 bytes
!
version 12.1
service config
no service single-slot-reload-enable
service timestamps debug uptime
service timestamps log uptime
no service password-encryption
!
hostname itgw1
!
no logging buffered
no logging buffered
logging rate-limit console 10 except errors
# Following two lines have been changed.
enable secret 5 $1$N3uKF4uJ.5lXdJahsUf/Ya8sK.
enable password mypass
!
!
!
clock timezone GMT 0
voice-card 1
!
```

```
ip subnet-zero
!
!
no ip finger
ip ftp username wenyu
ip domain-list cs.columbia.edu
no ip domain-lookup
ip domain-name cs.columbia.edu
ip name-server 128.59.16.20
ip name-server 128.59.16.9
!
no mgcp timer receive-rtcp
call rsvp-sync
!
voice class codec 1
 codec preference 2 g711ulaw
!
!
!
!
!
!
!
controller T1 1/0 # T1 parameters
 framing esf
 linecode b8zs
 ds0-group 1 timeslots 1-4 type e&m-wink-start # only 4 out of 24 channels allocated
 description line
!
!
interface Tunnel1
 no ip address
!
interface FastEthernet0/0
 ip address 128.59.19.28 255.255.248.0 # IP addr and netmask
 ip access-group 101 in # usage of IOS ACL number 101
 no ip mroute-cache
 speed auto
 half-duplex
 no cdp enable
!
ip default-gateway 128.59.16.1 # routing purpose
ip classless
ip route 0.0.0.0 0.0.0.0 FastEthernet0/0 128.59.16.1 2 # routing purpose
no ip http server
!
access-list 101 permit ip host 128.59.19.141 any # permit all traffic from/to sipd
access-list 101 permit ip 63.70.87.96 0.0.0.31 any # SIP Comm machines
access-list 101 permit udp host 128.59.16.20 range domain 54 any # DNS traffic
access-list 101 permit udp 128.59.16.0 0.0.7.255 range biff 65535 host 128.59.19.28 neq 5060
```

```
access-list 101 permit ip host 156.111.237.127 any
no cdp run
route-map map permit 10
 set ip default next-hop 128.59.16.1
!
snmp-server engineID local 0000000902000002FD40E640
snmp-server community public RO
snmp-server packetsize 4096
!
voice-port 1/0:1
!
dial-peer cor custom
!
!
!
dial-peer voice 1005 pots
 preference 6
 destination-pattern 8....... # outgoing local call
 no digit-strip
 port 1/0:1
!
dial-peer voice 1010 pots
 preference 7
 destination-pattern 4.... # outgoing campus call
 no digit-strip
 port 1/0:1
!
dial-peer voice 1020 pots
 preference 8
 destination-pattern 81.......... # outgoing long-distance call
 no digit-strip
 port 1/0:1
!
dial-peer voice 1003 pots
 preference 4
 destination-pattern 8... # outgoing service call (such as 411)
 no digit-strip
 port 1/0:1
!
dial-peer voice 3 voip
 destination-pattern 1.. # 3-digit extensions for non-DID call, to be verified
 voice-class codec 1
 session protocol sipv2
 session target ipv4:128.59.19.141
!
dial-peer voice 2 pots
 application session.t.old
 destination-pattern 7138\$ # non-DID incoming call leg, to be verified
 no digit-strip
 port 1/0:1
```

```
!
dial-peer voice 1 voip
 preference 1
 destination-pattern 713[0-79] # DID incoming call range
 voice-class codec 1 # u-law coding, defined earlier
 session protocol sipv2
 session target ipv4:128.59.19.141 # sipd's IP address
!
dial-peer voice 1000 pots
 preference 3
 destination-pattern ((70..)|(71[0-24-9].)) # IP to PBX internal call
 no digit-strip
 port 1/0:1
!
!
line con 0
 exec-timeout 0 0
 transport input none
line aux 0
line vty 0 4
 password remote8
 login
!
end
```

Cisco IOS commands for setting up a PRI T1 connection with the PBX are shown below. Note that we list only the differences with the channelized T1 case.

```
isdn switch-type primary-5ess   # setting up the ISDN PRI switch type, here it is 5ESS

controller T1 1/0
 framing esf
 fdl ansi
 linecode b8zs
 pri-group timeslots 1-24       # this PRI line is a full T1 (24 DS-0
channels)

interface Serial1/0:23          # the last DS-0 channel is the D-channel for
signaling
 isdn switch-type primary-5ess  # ISDN PRI switch type for each individual
circuit

voice-port 1/0:23

dial-peer voice 1 voip          # the dial-peer id is not important
...
 progress_ind setup enable 3    # to force ring-back in PSTN-to-IP calls for
ISDN
 dtmf-relay rtp-nte             # DTMF relay by RTP, available in IOS 12.2
...
```

```
dial-peer voice 10 pots
...
 direct-inward-dial     # this option is needed if the PRI line sends
                        # only DNIS (Dialed Number) digits instead of
                        # actually dialing the digits
 port 1/0:23
...
```

On a freshly booted Cisco router, if first-time setup does not get saved properly to flash memory, check its configure register by "show version" command:

```
Configuration register is 0x2102  # this is the right value; 0xA102 is the wrong one
```

If the register has the wrong value, change it by entering configure mode and type:

```
config-register 0x2102
```

A.3 PBX configurations

A.3.1 Layer 1: T1 Line cabling

First, we describe the cabling of the internal T1 line, because the connector types are not necessarily the same between the PBX and the gateway.

The T1 card on the PBX is a Nortel AS-1074PRI card. It uses a cable with a male DB-15 connector. In contrast, the Cisco voice gateway has a female T1 RJ-48C port. The pin assignments are listed in the following table. In this table, both the "network" and Telco is simply the other (peer) entity with respect to itself.

PBX's DB-15			Gateway's RJ-48C		
pin	signal	comment			
1	T	trasmit tip to network (network is the peer entity)	pin	signal	T1 NIC ↔ Telco
9	R	trasmit ring to network	1	R	←
2	FGND	frame ground	2	T	←
3	T1	receive tip from network	4	R1	→
11	R1	receive ring from network	5	T1	→

Table 10: Comparison of T1 port pin assignments between the PBX and gateway

As a result, we need an adapter cable with the following pin assignments:

DB-15 pins	RJ-48C pins
1	2
9	1
3	5
11	4

Therefore, we choose the BlackBox ETNM03-0005 (cross-pinning) unit. Because ETNM03-0005 has a male DB-15 connector, and the PBX's T1 cable also has a male DB-15 end, a BlackBox FA455 DB-15 female to DB-15 female couple (gender changer) is also used. Because the cables are not long enough, we use a RJ-48C female couple and a category-5 straight through cable to extend the total cable length.

Attribute	value
T1 line type	DTI (Digital Trunk Interface), i.e., channelized T1.
Line coding	B8ZS
T1 framing	ESF (Extended Super Frame)
Trunk signaling type	EM4 (E & M with 4-wire)
Trunk start signaling	Wink signaling
Signaling tone type	DTMF (Dual Tone Multiple-Frequency)

Table 11: Summary of key attributes

A.3.2 Layer 2: Link Layer configuration

Next, we describe the key parameters in the internal T1 line in Table 11, most of which have been discussed in Section 6.3

The following is a summarized printout of our PBX configuration, only the key parameters are displayed. All underlined words are typed in by the PBX administrator.

```
LD 22                   # LoaD the overlay program #22
REQ PRT                 # user REQuest (as the prompt) in PRinT
TYPE CFN                # ConFiguratioN record data for the PBX
...
PCML MU                 # PCM law is μ-law
...
DLOP NUM DCH FRM TRSH   # Digital trunk LOoP (i.e., like configurations)
TRK 003 24 D4 00        # T1 trunk to local phone company
004 24 ESF 00           # internal T1 trunk, using ESF framing
...

LD 16                   # LoaD the overlay program #16
REQ PRT                 # user REQuest (as the prompt) in PRinT
TYPE RDB                # Route Data Block (for routing call to the correct trunk)
CUST 0                  # CUSTomer id (usually 0)
ROUT 2                  # route number 2 for call routing
...
TKTP TIE                # TrunK TyPe. Configure it as a TIE line (bidirectional)
...
DTRK YES                # digital (as opposed to analog) trunk
DGTP DTI                # DiGital trunk TyPe: DTI is Digital Trunking Interface (i.e., channelized T1)
DSEL VCE                # Data SELector is VoiCE (i.e., voice-only trunk)
...
SRCH RRB                # free channel SeaRCH method: Round-RoBin
...
TARG 01                 # Trunk Access Restriction Group
...

LD 21
```

```
REQ LTM                    # List Trunk Members (individual channels in a trunk)
CUST 0
ROUT 2
TYPE TLS
TKTP TIE
ROUT 2
TN 004 01 MBER 1           # 4 out of 24 channels allocated
TN 004 02 MBER 2           # TN means terminal number (like a telephone slot)
TN 004 03 MBER 3
TN 004 04 MBER 4

LD 20
REQ PRT
TYPE TNB
TN 4                       # list all Terminal Numbers with prefix 4
CUST
DATE
PAGE
DES
TN 004 01                  # Terminal Number 4 1 (1st out of 4 allocated channel)
TYPE TIE
CUST 0
NCOS 7                     # NCOS (Network Class Of Service) of 7,
...                        # usually means all calls allowed.
                           # Higher value implies more privileges.
SIGL EM4                   # E&M 4-wire trunk
STRI/STRO WNK WNK          # wink start signaling
...

LD 87
REQ PRT
CUST 0
FEAT CDP                   # FEATure if Coordinated Dialing Plan
TYPE DSC                   # Distant Steering Code, which is the DID prefix in our case
DSC xxx
RLI xxx                    # Route List Index to be accessed for DSC

LD 86
REQ PRT
CUST 0
FEAT RLB                   # FEATure is Route List Information
RLI xxx
ROUT xxx
FRL xxx                    # Facility Restriction Level, usually same as NCOS
...


LD 60                      # what to do if T1 line is on red-alarm after rebooting the gateway
STAT                       # shows current STATus of all trunks
DISL 4                     # DIsable Loop (trunk) number 4 (the internal T1 line)
```

```
ENNL 4                  # enable it again, which reset the T1 alarm
STAT                    # double-check the trunks' status

LD 14                   # Modify Trunk Data Block
REQ NEW
TYPE TIE
...
                        # Just type return by default
TN 4 5                  # Adding a new terminal number
...
CUST 0
...
NCOS 7
RTMB 2 5                # RouTe MemBer (corresponds to an individual DS-0 channel),
...                     # Add 5th member for T1 trunk 2
TGAR 0                  # Trunk Group Access Restriction, restricts this line if TGAR matches with TARG
SIGL EM4                # E&M 4-wire trunk
...
STRI WNK                # Incoming calls: WiNK STart Signaling
STRO WNK                # Outgoing calls: WiNK STart Signaling
CLS DTN                 # T1 needs tone-dialing instead of the default pulse-dialing,
                        # CLS means service CLaSs and DTM stands for Digital ToNe dialing
```

After adding a DS-0 channel on the PBX, make sure to reflect the change in the voice gateway as well using the ds0-group sub-command in the T1 controller command. For example, with 5 channels now, use the following:

```
ds0-group 1 timeslots 1-5 type e&m-wink-start
```

A.4 Database tables

This section describes the database tables we are using with an example of the default values.

```
mysql> desc put;
+------------------+-----------------------------------+------+-----+---------+-------+
| Field            | Type                              | Null | Key | Default | Extra |
+------------------+-----------------------------------+------+-----+---------+-------+
| user             | varchar(255)                      |      | PRI |         |       |
| hash_value       | varchar(255)                      |      |     |         |       |
| realm            | varchar(150)                      |      |     |         |       |
| sip_groups       | varchar(100)                      | YES  |     | NULL    |       |
| auth             | enum('required','request','never') | YES |     | NULL    |       |
| algorithm        | varchar(20)                       |      |     |         |       |
| sip_methods      | varchar(100)                      | YES  |     | NULL    |       |
| remote_user      | text                              | YES  |     | NULL    |       |
| busy             | varchar(255)                      | YES  |     | NULL    |       |
| noresponse       | varchar(255)                      | YES  |     | NULL    |       |
| message_template | text                              | YES  |     | NULL    |       |
| um_timeout       | smallint(5) unsigned              | YES  |     | NULL    |       |
| max_msgsize_kb   | int(10) unsigned                  |      |     | 200     |       |
```

```
| last_modified    | timestamp(14)                        | YES  |     | NULL    |       |
| gwclass          | varchar(100)                         | YES  |     | NULL    |       |
| email            | text                                 | YES  |     | NULL    |       |
+------------------+--------------------------------------+------+-----+---------+-------+
16 rows in set (0.00 sec)

mysql> desc contacts;
+---------------+--------------------------+------+-----+---------------------+-------+
| Field         | Type                     | Null | Key | Default             | Extra |
+---------------+--------------------------+------+-----+---------------------+-------+
| user          | text                     |      | PRI |                     |       |
| contact       | text                     |      | PRI |                     |       |
| expires       | datetime                 | YES  |     | 9999-12-31 23:59:59 |       |
| q             | float(10,2)              | YES  |     | NULL                |       |
| action        | enum('Proxy','Redirect') | YES  |     | NULL                |       |
| last_modified | timestamp(14)            | YES  |     | NULL                |       |
| display_name  | varchar(100)             | YES  |     | NULL                |       |
| sip_methods   | varchar(100)             | YES  |     | any                 |       |
+---------------+--------------------------+------+-----+---------------------+-------+
8 rows in set (0.00 sec)

mysql> desc aliases;
+---------------+---------------+------+-----+---------+-------+
| Field         | Type          | Null | Key | Default | Extra |
+---------------+---------------+------+-----+---------+-------+
| alias         | varchar(255)  |      | PRI |         |       |
| primary_user  | varchar(255)  |      |     |         |       |
| last_modified | timestamp(14) | YES  |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Default values are shown below.

```
user        : default@cs.columbia.edu
hash_value  : 421f557ae325c7b739a135e4683706ef
realm       : cs.columbia.edu
sip_groups  : cgi voicemail
auth        : never
algorithm   : MD5
sip_methods : REGISTER INVITE any
remote_user : NULL
busy        : rtsp://SERVER/audio/welcome.au
norespone   : rtsp://SERVER/audio/welcome.au
message_template:
 To: [email $to]
 Reply-To: [email $from]
 Subject: voice mail -- $subject
 Priority: $priority

 Dear $name,
```

```
 Voice mail from [dataurl $from $from] has arrived.
 You can play the message by using QuickTime using
 url $rtspurl or
 by going to the web page at $httpurl.

 - $administrator

 --
 This mail was automatically generated by the vmail system.
um_timeout  :  10
max_msgsize_kb:200
last_modified: 20010424085810
gwclass     : student
email       :default@cs.columbia.edu
```

The list of gateway classes are shown below.

```
mysql> select * from gwclass;
+---------+
| gwclass |
+---------+
| faculty |
| phd     |
| staff   |
| student |
+---------+
4 rows in set (0.02 sec)
```

A.5 DNS SRV record

SIP clients use DNS SRV records if available. A sample DNS zone file entry is shown below:

```
sip.tcp         SRV 0 0 5060 sip-server.cs.columbia.edu.
                SRV 1 0 5060 backup.ip-provider.net.
sip.udp         SRV 0 0 5060 sip-server.cs.columbia.edu.
                SRV 1 0 5060 backup.ip-provider.net.
```

According to RFC 2782, the protocol designations are to be prefixed by an underscore, so that the correct entries are:

```
_sip._tcp         SRV 0 0 5060 sip-server.cs.columbia.edu.
                  SRV 1 0 5060 backup.ip-provider.net.
_sip._udp         SRV 0 0 5060 sip-server.cs.columbia.edu.
                  SRV 1 0 5060 backup.ip-provider.net.
```

DNS SRV records are supported by BIND 4.9.6 and newer, generally installed as named.

Currently registered SRV records:

```
sip.tcp.cs.columbia.edu SRV     0 0 5060 conductor.cs.columbia.edu
sip.udp.cs.columbia.edu SRV     0 0 5060 conductor.cs.columbia.edu
_sip._tcp.cs.columbia.edu SRV   0 0 5060 conductor.cs.columbia.edu
```

```
_sip._udp.cs.columbia.edu SRV   0 0 5060 conductor.cs.columbia.edu
_sip._tcp.cs.columbia.edu SRV  10 0 5060 erlang.cs.columbia.edu
_sip._udp.cs.columbia.edu SRV  10 0 5060 erlang.cs.columbia.edu
_sip._tcp.cs.columbia.edu SRV  20 0 5060 backbay.cs.columbia.edu
_sip._udp.cs.columbia.edu SRV  20 0 5060 backbay.cs.columbia.edu
```