Session Initiation Protocol Server Implementation for Linksys Routers

A Senior Project

presented to

the Faculty of the Computer Science department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science, Computer Science

by

Matthew Duder

December, 2009

© 2009 Matthew Duder
Abstract

Session Initiation Protocol (SIP) is a signaling protocol used for network-based media interaction. SIP Servers are required to process and forward requests and responses between end clients. Once connection setup is complete, the end clients utilize some other protocol (such as Real-Time Transport Protocol) to complete their task.

Such a server may be implemented and installed on routers. Unlike the SIP protocol, however, router development currently remains without standards. For this project, Linux-based residential-grade routers had their kernels replaced with an open source version. This allowed for installation of open-source development, such as a SIP server implementation.

This project led to the development and installation of such a server application on two separate Linksys WRT54GL routers. After proper network configuration, these servers communicated with both local clients and each other. These servers forwarded SIP traffic and enabled voice communication over separate networks.
# Table of Contents

Abstract ........................................................................................................................................... i
Overview ........................................................................................................................................1
Technologies ................................................................................................................................3
SIP Signaling Overview .................................................................................................................5
SIP Server Design ........................................................................................................................8
Software Development Process ...................................................................................................9
Router Kernel Image Build Process ............................................................................................10
Project Implementation ................................................................................................................11
Conclusion ....................................................................................................................................13
Source Code ...............................................................................................................................14
OpenWRT Makefiles ....................................................................................................................27
Glossary .......................................................................................................................................28
Bibliography ...............................................................................................................................29

# List of Figures and Diagrams

Figure 1: Linksys WRT54GL Router .............................................................................................3
Figure 2: Linksys PAP-2T ATA ..................................................................................................3
Figure 3: Networks Lab Telephone ..............................................................................................4
Figure 4: Networks Lab Netgear Hub ..........................................................................................4
Figure 5: SIP-enabled Internet ....................................................................................................11
Diagram 1: SIP Signaling and RTP Direct Communication .......................................................2
Diagram 2: SIP Registration, Call Setup and Termination .........................................................7
Overview

Voice over IP technology allows phone calls to be made over multiple networks. VoIP uses two types of protocols to accomplish this. The first protocol, Session Initiation Protocol, must handle device signaling and synchronize the connection. Once devices are synchronized, a second protocol transfers actual voice data. This second protocol is Real-Time Transfer Protocol.

Session Initiation Protocol (SIP) is “an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences.” [1] This protocol operates at OSI layer five and is encapsulated by UDP and IP.

This project implemented an end-to-end VoIP solution using Linksys WRT54GL routers. These routers provided SIP processing between two end clients. For end clients, I chose Linksys P4P Analog Telephony Adapters, or ATAs. The ATAs converted standard phone signals into SIP packets and forwarded them to the local SIP server.

The server application was developed to modify and forward SIP packets between itself and other servers running the same application. This application also talked to end clients.

Once the ATAs established a direct connection, they communicated using Real-Time Transfer Protocol. The ATAs then relayed the voice data to their connected telephones. Once a conversation was finished, a final SIP packet from each client was sent to each other directly, signaling a finished call. See Diagram 1 on the following page for a visual representation of this process.
For my project, I created a SIP server application. This application interfaces with a local ATA and supports communication between itself and a remote ATA. The server applications needed to run on routers local to the ATAs. Being embedded systems, residential routers [3] do not allow simple installation of server applications. For this project, then, I found a router model with an operating system kernel which could be re-written to allow my SIP application to run.

The open-source replacement kernel for these routers is Linux based. This made application development simple, as it could be done and tested in a network lab using desktop PCs as temporary servers until the application was complete. After this, the application was compiled into the Linux kernel necessary for the routers. The routers were flashed with the new kernel and rebooted, with the SIP server application ready to run.

Then final step required network configuration for these routers. For this project, directly connecting the routers’ WAN ports together served as a separate network. Each ATA then connected to its own router’s LAN port. After configuring the networks, calls could be made from one phone to another.
Technologies

Linksys WRT54GL routers (Figure 1) are used in this project to forward SIP packets. The four LAN ports on this router are located in the back, next to the WAN port. The firmware which comes with the router offers web based configuration.

Figure 1: Linksys WRT54GL Router

Linksys PAP-2T ATAs (see Figure 2) convert VoIP signals to telephone signals. The back side of each ATA has two phone ports and one Ethernet port. Like the routers, the ATAs come with a web configuration utility.

Figure 2: Linksys PAP-2T ATA
Any traditional phone will work for interaction with the ATA (see Figure 3). So long as you know the remote client’s phone number, you simply dial it in and you will hear the standard dial tone and busy tone when you normally would while using a phone service.

![Networks Lab Telephone](image)

Figure 3: Networks Lab Telephone

Hubs (see Figure 4) are not required for SIP server functionality. They were used, however, in the debugging and configuration process for picking up communication between an ATA and its router.

![Networks Lab Netgear Hub](image)

Figure 4: Networks Lab Netgear Hub
SIP Signaling Overview [1]

In SIP terminology, end clients are referred to as User Agent clients, or UACs. Their immediate contact points are referred to as User Agent servers, or UASs. These proxy servers respond to and pass SIP messages between each other. Registrars also accept SIP communication from UACs, updating device registration status with phone number and IP address pairs. (Note: for my implementation, I combined these two server types for the sake of simplicity.)

SIP messages are stateless and clear-text based. They contain a series of headers separated by carriage-return and new-line characters. After this header series, depending on the signal, the initial RTP payload may also follow. This RTP payload usually contains its own headers which describe the type of media about to be relayed.

Most initiating clients’ messages come with a type label in their first header, the request line. Examples include REGISTER, INVITE, BYE, ACK and OPTIONS. All responses from every receiving device start with a response line as the first header. This response line states the message STATUS xxx <text>. The x’s represent a three digit number, and the proceeding text summarizes what this number means. See Diagram 2 on page 7 for a representation of SIP message passing using my integrated proxy/registry server design.

UACs may be configured to register when they power on. It contains the registrar IP address in its configuration, and once it receives an IP address from its gateway, it sends a REGISTER message to the registry. If this registration can be made, the registry responds with a STATUS 200 OK message. At this point, the UAC is active.

A proxy server must forward request messages from its local client to the remote proxy server. A proxy server must generate its own status response to exactly one type of client call request message: the INVITE. A client’s INVITE requires an immediate response without waiting for a remote response. This response is known as a provisional response, and it prevents the client from otherwise duplicating INVITES. The specific status response for an INVITE is 100 TRYING.
The INVITE is then forwarded until it reaches the remote UAC. Every proxy that handles a SIP request message must append the message with its own header. This header contains the address of the device. The receiving end client then decides how to handle this message, and sends a response back to its own local proxy. The message is passed back on the same path from which it was received. This time, every device handling the message strips its old header off of the status message (which the receiving client duplicated from the request message) and forwards the message.

At this point, the sending UAC awaits further status messages from the receiving client. If it receives a RINGING message, it signals to its connected telephone to play the traditional ringing tone to the end user. If and when it receives a proceeding OK message, it then directly connects to the receiving UAC. It sends a direct ACK SIP message along with the initial RTP payload. The two UACs then exchange voice data with RTP.

Once a phone conversation is completed, the UAC which hung up first signals a BYE message to the other UAC, which then confirms it and terminates the call on its end. The UAC remains active and available to send or receive until either its registration expires and it can not re-register, or it loses its network connection.
Diagram 2: SIP Registration, Call Setup and Termination

- REGISTER UAC 1 ➔
- 200 OK ←
- REGISTER UAC 2 ➔
- INVITE UAC 2 ➔
- 200 OK ←
- 100 TRYING ➔
- INVITE UAC 2 ➔
- INVITE UAC 2 ➔
- 180 RINGING ←
- 180 RINGING ←
- 200 OK ➔
- 180 RINGING ←
- 200 OK ➔
- 200 OK ←

Voice Communication via RTP

BYE ➔

200 OK ➔
SIP Server Design

This software server may be considered client-server, as its primary design is to pass messages between clients. Per SIP standards, there are three logically separate servers required for implementation [1]. A registrar is a server which stores the phone number and IP address combinations of all registering clients. A proxy server is one which receives client requests and forwards them to remote proxy servers. I rolled these two processes into a single server.

The third server type is a redirect server. This is the server a sending remote proxy connects to in order to locate the appropriate receiving remote proxy before forwarding the packets. To keep the server logic simple, I bypassed this requirement by requiring each server application to take a command-line argument when starting the server. This argument is the IP or hostname of the alternate server. Thus, no redirect is required.

Ports 5060 and 5061 are “well known” SIP ports [1]. As the Linksys ATAs communicate via port 5060, I designed this application to register sockets listening in on that port. I had the servers communicate with each other on the same port. SIP headers provide enough information to distinguish which type of user agent sent the message to the server (whether it was a client or the other server.)
Software Development Process

Before developing the server application, much time was spent researching SIP design. This server only supports basic functionality. Therefore, some of SIP’s design [1] is not implemented. This software is modular however, so supporting more SIP message types requires only the insertion of functions which parse and respond to such type headers, as well as adding in detection for such a header in its parent function. The current message types supported are: REGISTER, INVITE, CANCEL, ACK, and none (which implies a status message.)

This software was built incrementally. First, it needed only to pass the sender’s initial message to the receiver. At this time, I also had to respond to the sender with an immediate status message to prevent duplicate send requests. (Although the remote client will discard duplicate send requests with the same identification number, good system design still calls for minimizing packet flow.)

Once this worked, modifying the software to return messages from the remote, receiving client was simple. I have left my labeled debug print messages in the source code. This may help anyone who picks up this source in the future and decides to modify it. Removing them if found unnecessary is trivial, and will not modify the functionality of the software.

The application Wireshark was used extensively in testing and debugging. This allowed me to detect incorrect response packet formation during the development process. I was notified of any malformed packets sent out by my SIP server.
Router Kernel Image Build Process

Running a developed application on a Linksys router is not straight-forward. The router is an embedded device which runs on a Broadcom chipset. Kernel images vary from chipset to chipset, so I searched for a robust solution which handles this proprietary nature. OpenWRT is an image build suite which abstracts the image build process over the entire range of its supported hardware [2].

Devices supported by OpenWRT guarantee POSIX compatibility. Thus, developing network software will work on this platform if it is developed in a manner similar to programs from both undergraduate networks courses (CPE 464 and 465.) Once the software is finished in development, it can be included in the OpenWRT image build process in two steps: create the correct directory structure, and create OpenWRT-custom Makefiles within that structure.

For the SIP server, the directory package path supplied by OpenWRT is /kamikaze/8.09/package. There exists a directory for each package, so I created a new one: /sips. Inside /sips, there is the package integration Makefile and a /src directory. Inside /src lies the server source code sips.c and the Makefile required to compile the source. The contents of both Makefiles can be viewed on page 27.

Once the image is generated, the router updating process varies by router model. For the WRT54GL, there are multiple options [3]. When upgrading from default firmware to OpenWRT, I used the web configuration tool. I loaded http://192.168.1.1, selected the Administration link, then the Firmware Upgrade link. This lets you browse for the new image on your local directory. When file transfer is complete, the router reboots. The router requires several minutes after reboot to finish updating the image.

If the next image you want to update to is available on a public web server, you can log into your OpenWRT enabled router and use Linux’s wget tool. Otherwise, you can use tftp and upload the image directly from your system. If you need to use tftp from windows, follow this link to acquire Linksys’ application: http://downloads.linksysbycisco.com/downloads/Tftp,0.exe
Project Implementation

Here is the finished physical setup of three networks. I left the ATAs configured in a fashion to use DHCP for an IP address when powered on. This required sniffing network packets to determine which IP address each one acquired. This IP address is required to log into the ATA for web configuration. Thus, these hubs allow the DHCP dialog between the ATA and its router to be seen by the computer connected, for each network, as shown in Figure 5.

![Figure 5: SIP-enabled Internet](image)

Aside from this, you see the telephones are directly connected to the ATAs in line 1. For ATA configuration, from the computer, go to http://ATA.IP.address. Select the “admin” option at the right. Then select the “Line 1” tab. Here you will be able to dictate the phone number for the connected phone, as well as set the ATA’s registrar and proxy servers to be that of the router.
By default, an OpenWRT kernel image installation allows communication between devices on its LAN ports. However, the default network configuration sets up on the 192.168.1.1 domain. Furthermore, it does not come with a routing protocol by default, so unless you set up a static WAN route, it is not going to utilize address learning to determine what the outside network’s IP range is.

To solve this problem, I modified the network configuration for each router and added the other router as the gateway. There are two ways to modify such a configuration: either use the command-line “uci” tool, or modify /etc/config/network inside the router itself. I opted for modifying the network file. Here is the current configuration for the second router:

```
cfg 'switch' 'eth0'
  opt 'vlan0' '0 1 2 3 5*
  opt 'vlan1' '4 5'

cfg 'interface' 'loopback'
  opt 'ifname' 'lo'
  opt 'proto' 'static'
  opt 'ipaddr' '127.0.0.1'
  opt 'netmask' '255.0.0.0'

cfg 'interface' 'lan'
  opt 'type' 'bridge'
  opt 'ifname' 'eth0.0'
  opt 'proto' 'static'
  opt 'ipaddr' '192.168.2.1'
  opt 'netmask' '255.255.255.0'

cfg 'interface' 'wan'
  opt 'ifname' 'eth0.1'
  opt 'proto' 'static'
  opt 'ipaddr' '192.168.3.2'
  opt 'netmask' '255.255.255.0'
  opt 'gateway' '192.168.3.1'
```

For the first router, change the LAN’s ipaddr to 192.168.1.1, its WAN’s ipaddr to 192.168.3.1 and the WAN’s gateway to 192.168.3.2. Regardless of whether you use uci or modify the network file, you must always issue the command “ifup wan” (for interface update – WAN) before it will re-configure the WAN portion.

Also note, I had to disable OpenWRT’s firewall to allow this traffic. OpenWRT’s firewall relies on iptables, which I found to be a nightmare and did not configure. I would not recommend running without a firewall up in practice, but iptables was too complex to learn in the amount of time allocated for the project.
Conclusion

Implementing SIP is inherently modular. The process described earlier allows straightforward, incremental implementation. However, the kernel image building process was a real headache to figure out. Open source documentation (or lack thereof, rather) has a reputation, and OpenWRT certainly lived up to it. The best way to generate a Makefile for a new application to be installed in the kernel image is to dig through all the Makefiles for other packages. Then, exhaustively copy one Makefile to your own directory structure, modify it in the hopes that it retains validity with the build chain, and try to run the make configuration. If your package does not show up, exit, copy the next Makefile and try again.

This is necessary because there is only a tiny amount of documentation on this process. As of right now, that documentation is absolutely obsolete. Major updates to OpenWRT seem to change the format of the Makefile requirements, so what is documented from 2006 or 2007 no longer retains validity.

Also, I recommend locating two “optional” kernel additions and installing them. One is a RIP application which would allow you to actually route with the router. The second is a web-configuration GUI. I believe the current version is LuCI, but the specific GUI used also changes every now and then. This web configuration tool not only allows easier network configuration changes, but also allows a kernel update much like the default firmware did before changing over to OpenWRT.
/**
* Bare Functionality Session Initiation Protocol server
* Matthew Duder 2009  CSC 491,492
***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <sys/select.h>

/* Prototype Declarations */
void validateArg(int, char**);
int sockSetup(uint16_t);
int selectCall(int, int, int);
int getRemoteType(char*, int);
int getMsgType(char*, int);
int getProxyMsgType(char*, int);
void handleRegister(int, char*, int, int, struct sockaddr_in, socklen_t);
void tryingResponse(char*, int, int, struct sockaddr_in);
void insertAndForward(char*, int, int, struct sockaddr_in, char*);
void stripAndForward(char*, int, int, struct sockaddr_in, char*);
void confirmCancel(char*, int, int, struct sockaddr_in);

/* Global Variables */
static int kMaxSize = 1458; // Maximum packet size minus Eth, IP & UDP headers
static uint16_t kSipPort = 5060; // Default port for SIP implementation
static char kCrgRtn = 13; // ascii value for a carriage return
static char kLnFd = 10; // ascii value for a linefeed
static char kSpace = 32; // ascii value for a space
static char kSemi = 59; // semi-colon

/**
* Required main function
*/
int main(int argc, char* argv[]) {
    struct sockaddr_in addr;
    socklen_t addrLen = sizeof(addr);
    struct sockaddr_in client;
    socklen_t clientLen = sizeof(client);
    struct sockaddr_in proxy;
    struct sockaddr_in local;
    socklen_t localLen = sizeof(local);
    struct hostent* hp = NULL;

    /* Set up remote proxy information */
    validateArg(argc, argv);
    if (!(hp = gethostbyname(argv[1]))) {
        perror("gethostbyname");
        return 0;
    }
memcpy((char*)&proxy.sin_addr, (char*)hp->h_addr, hp->h_length);
proxy.sin_port = htons(kSipPort);
proxy.sin_family = AF_INET;

/* Set up the server socket and loop on select call for connections */
sockNum = sockSetup(kSipPort);
if (getsockname(sockNum, (struct sockaddr*) &local, &localLen) < 0) {
  perror("getsockname");
  return 0;
}
localIP = (char*) inet_ntoa(local.sin_addr);

while (1) {
  if (selectCall(sockNum, 1, 0)) {
    count++;
    memset(msgBuf, 0, kMaxSize);
    msgLen = recvfrom(sockNum, msgBuf, kMaxSize, 0,
                      (struct sockaddr*) &addr, &addrLen);
    if (msgLen < 0) {
      perror("recvfrom");
      exit(-1);
    }
    /* String compare user-input proxy with remote's IP to determine source */
    remoteType = strcmp(argv[1], (char*) inet_ntoa(addr.sin_addr));
    if (remoteType != 0) {
      /* Packet from client; must forward to proxy and/or reply with status */
      msgType = getMsgType(msgBuf, msgLen);
      if (msgType == 1) {
        /* Client is registering */
        client = addr;
        clientLen = addrLen;
        handleRegister(msgType, msgBuf, msgLen, sockNum, client, clientLen);
      }
    }
    else if (msgType == 2) {
      /* Client is inviting */
      tryingResponse(msgBuf, msgLen, sockNum, client);
      insertAndForward(msgBuf, msgLen, sockNum, proxy, localIP);
    }
    else if (msgType == 3) {
      /* Client is cancelling an invite */
      confirmCancel(msgBuf, msgLen, sockNum, client);
      insertAndForward(msgBuf, msgLen, sockNum, proxy, localIP);
    }
    else if (msgType == 4) {
      /* Receiving Client is sending an ACK to the sending Client */
      insertAndForward(msgBuf, msgLen, sockNum, proxy, localIP);
    }
    else {
      /* Client is responding with a status message */
      stripAndForward(msgBuf, msgLen, sockNum, proxy, localIP);
    }
  }
  else {
    /* Packet is from proxy; it must only go to the local client */
    msgType = getMsgType(msgBuf, msgLen);
    if (msgType > 0) {
      /* A non-status msg from the initiating client – add a via header */
      insertAndForward(msgBuf, msgLen, sockNum, client, localIP);
    }
    else {
      /* Status message from the receiving client – strip a via */
      stripAndForward(msgBuf, msgLen, sockNum, client, localIP);
    }
  }
}
return 0;
/**
 * Validate remote proxy argument
 *
 * argc     number of arguments
 * argv     vector of actual arguments
 */
void validateArg(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s remote SIP proxy\n", argv[0]);
        exit(1);
    }
    if (!(gethostbyname(argv[1]))) {
        printf("Bad argument supplied (invalid address). Exiting.\n");
        exit(1);
    }
    return;
}

/**
 * Server socket setup
 * This function was provided by Dr. Hugh Smith for CPE 464
 *
 * port         port to which the socket is bound
 * return       socket value
 */
int sockSetup(uint16_t port) {
    int sockNum = -1;
    struct sockaddr_in local;
    socklen_t len= sizeof(local);
    if ((sockNum = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket call");
        exit(1);
    }
    local.sin_family= AF_INET;
    local.sin_addr.s_addr= INADDR_ANY;
    local.sin_port= htons(port);
    if (bind(sockNum, (struct sockaddr *) &local, sizeof(local)) < 0) {
        perror("bind call");
        exit(-1);
    }
    if (getsockname(sockNum, (struct sockaddr*) &local, &len) < 0) {
        perror("getsockname call");
        exit(-1);
    }
    return sockNum;
}

/**
 * Timed socket probe
 * This function was provided by Dr. Hugh Smith for CPE 464
 *
 * socket       the socket to probe
 * seconds      amount of seconds for each probe
 * useconds     amount of milliseconds for each probe
 * return       1 if socket ready for read, 0 if not
 */
int selectCall (int socket, int seconds, int useconds) {
    static struct timeval timeout;
    fd_set fdvar;
    timeout.tv_sec = seconds;
    timeout.tv_usec = useconds;
    FD_ZERO(&fdvar);
    FD_SET(socket, &fdvar);
    if ((select (socket + 1, (fd_set*) &fdvar, (fd_set*) 0, (fd_set*) 0, &timeout)) == -1) {
        perror("select error");
        exit(-1);
    }
if (FD_ISSET(socket, &fdvar)) {
  return 1;
} else {
  return 0;
}

/**
 * Determine if connected remote type is a client or a proxy
 *
 * buffer       the received message
 * length       message length
 * return       1 if message came from a client, 2 if proxy
 */
int getRemoteType(char buffer[kMaxSize], int length) {
  int count = 0, viaCount = 0, headerFlag = 0;

  /* Walk through the buffer and count the 'Via' headers */
  while (!headerFlag) {
    while (buffer[count] != kLnFd && !headerFlag) {
      count++;
    }
    if (buffer[count + 1] == kLnFd &&
        buffer[count + 2] == 'V' &&
        buffer[count + 3] == 'i' &&
        buffer[count + 4] == 'a' &&
        buffer[count + 5] == ':') {
      viaCount++;
      count++;
    } else if (buffer[count + 1] == kLnFd &&
               buffer[count + 2] == kCrgRtn &&
               buffer[count + 3] == kLnFd) {
      headerFlag = 1;
    } else if (buffer[count + 1] == kLnFd) {
      count += 2;
    } else { // TEST PRINT
      printf("\nUnknown error at count of %d. Exiting.\n", count);
    }
  }

  printf("\n(Total via count: %d)\n", viaCount); // TEST PRINT
  return viaCount;
}

/**
 * Determine which type of client message was received
 *
 * buffer       the received message
 * length       message length
 * return
 */
int getMsgType(char buffer[kMaxSize], int length) {
  int count = 0, msgType = 0;
  char typeBuffer[32] = {0};
  char* type = NULL;

  while (buffer[count] != kSpace && count < 31) {
    typeBuffer[count] = buffer[count];
    count++;
  }
  printf("\nClient message type: %s\n", typeBuffer); // TEST PRINT
  if (!(strcmp(typeBuffer, "REGISTER"))) {
    msgType = 1;
    printf("Returning type REGISTER (1)\n");
  } else if (!(strcmp(typeBuffer, "INVITE"))) {
    msgType = 2;
    printf("Returning type INVITE (2)\n");
  } else { // TEST PRINT
    printf("\nUnknown error at count of %d. Exiting\n", count);
  }
  return msgType;
}
```c
msgType = 2;
    printf("Returning type INVITE (2)\n");
} else if (!strncmp(typeBuffer, "CANCEL")) {
    msgType = 3;
    printf("Returning type CANCEL (3)\n");
} else if (!strncmp(typeBuffer, "ACK")) {
    msgType = 4;
    printf("Returning type ACK (4)\n");
} else {
    msgType = 0;
    printf("Returning type 'status' (0)\n");
}
return msgType;

/**
 * Determine which type of proxy message was received
 * buffer       the received message
 * length       message length
 * return
 */
int getProxyMsgType(char buffer[kMaxSize], int length) {
    int count = 0;
    char typeBuffer[32] = {0};
    char* type = NULL;
    while (buffer[count] != 20 && count < 31) {
        typeBuffer[count] = buffer[count];
        count++;
    }
    /* if ((type = (char*) malloc(count)) < 0) {
       perror("malloc");
       exit(-1);
    } */
    printf("Proxy message type: %s\n", typeBuffer); // TEST PRINT
    return 1;
}

/**
 * Respond to a client's register message
 * msgType      (deprecated)
 * buffer       the client's message
 * msgLen       length of client's message
 * sockNum      socket to communicate to client with
 * client       client's internet contact information
 * cLen         length of the client's info
 */
void handleRegister(int msgType, char buffer[kMaxSize], int msgLen,
                     int sockNum, struct sockaddr_in client, socklen_t cLen) {
    int count = 0, bufCount = 0, stringLen = 0, matchFound = 0;
    int tempCount = 0, sendLen = 0;
    char bufChar = 0;
    char response[kMaxSize];
    char fromLine[256] = {0};
    char toLine[256] = {0};
    memset(response, 0, kMaxSize);
    if (msgType == 1) {
        char statLine[] = "SIP/2.0 200 OK";
        stringLen = strlen(statLine);
        memcpy(&(response[count]), statLine, stringLen);
        count += stringLen;
        response[count] = kCrgRtn;
        response[count + 1] = kLnFd;
        count += 2;
```
/* Write the Via header */
while (!matchFound && bufCount < msgLen) {
    if (buffer[bufCount] == kCrgRtn &&
        buffer[bufCount + 1] == kLnFd &&
        buffer[bufCount + 2] == 'V' &&
        buffer[bufCount + 3] == 'i' &&
        buffer[bufCount + 4] == 'a' &&
        buffer[bufCount + 5] == ':') {
        bufCount += 2;
        bufChar = buffer[bufCount];
        while (bufChar != kSemi) {
            response[count] = bufChar;
            count++;
            bufCount++;
            bufChar = buffer[bufCount];
        }
        response[count] = kSemi;
        count++;
        /* Here, half of the sender's via is copied; buffer[bufCount] == kSemi */
        char recvLine[] = "received=";
        stringLen = strlen(recvLine);
        memcpy(&response[count], recvLine, stringLen);
        count += stringLen;
        char* clientIP = (char*) inet_ntoa(client.sin_addr);
        printf(">>> TESTING: Client IP is %s\n", clientIP); // TEST PRINT
        stringLen = strlen(clientIP);
        memcpy(&response[count], clientIP, stringLen);
        count += stringLen;
        response[count] = kSemi;
        count++;
        bufCount++;
        bufChar = buffer[bufCount];
        while (bufChar != kCrgRtn) {
            response[count] = bufChar;
            count++;
            bufCount++;
            bufChar = buffer[bufCount];
        }
        response[count] = kCrgRtn;
        response[count + 1] = kLnFd;
        count += 2;
        bufCount += 2;
        matchFound = 1;
    }
    bufCount++;
}
if (bufCount >= msgLen) {
    printf(">>> ERROR parsing <Via> header. Not transmitting a response.\n");
    return;
}

bufCount--; // To account for logic from the Via loop ....
matchFound = 0;
/* Via header in response should now be written; now write To and From headers */
bufChar = buffer[bufCount];
while (bufChar != kCrgRtn && bufCount < msgLen) {
    fromLine[tempCount] = bufChar;
    tempCount++;
    bufCount++;
    bufChar = buffer[bufCount];
}
if (bufCount >= msgLen) {
    printf(">>> ERROR parsing <From> header. Not transmitting a response.\n");
    return;
}
tempCount = 0;
bufCount += 2;
bufChar = buffer[bufCount];
while (bufChar != kCrgRtn && bufCount < msgLen) {
  toLine[tempCount] = bufChar;
  tempCount++;
  bufCount++;
  bufChar = buffer[bufCount];
}
if (bufCount >= msgLen) {
  printf(">>> ERROR parsing <To> header. Not transmitting a response.\n");
  return;
}
bufCount += 2;

/* Writing To header, then From header */
stringLen = strlen(toLine);
memcpy(&(response[count]), toLine, stringLen);
count += stringLen;
response[count] = kSemi;
count++;
char tagLine[] = "tag=oicu812-1";
stringLen = strlen(tagLine);
memcpy(&(response[count]), tagLine, stringLen);
count += stringLen;
response[count] = kCrgRtn;
response[count + 1] = kLnFd;
count += 2;

stringLen = strlen(fromLine);
memcpy(&(response[count]), fromLine, stringLen);
count += stringLen;
response[count] = kCrgRtn;
response[count + 1] = kLnFd;
count += 2;

/* Now write Call-ID and CSeq headers (rip from the client message) */
bufChar = buffer[bufCount];
while (bufChar != kCrgRtn && bufCount < msgLen) {
  response[count] = bufChar;
  count++;
  bufCount++;
  bufChar = buffer[bufCount];
}
if (bufCount >= msgLen) {
  printf(">>> ERROR parsing <Call-ID> header.\n");
  return;
}
response[count] = kCrgRtn;
response[count + 1] = kLnFd;
bufCount += 2;
bufChar = buffer[bufCount];
while (bufChar != kCrgRtn && bufCount < msgLen) {
  response[count] = bufChar;
  count++;
  bufCount++;
  bufChar = buffer[bufCount];
}
if (bufCount >= msgLen) {
  printf(">>> ERROR parsing <CSeq> header. Not transmitting a response.\n");
  return;
}
response[count] = kCrgRtn;
response[count + 1] = kLnFd;
count += 2;
bufCount += 2;

/* Write the Allow header */
char allowLine[] = "Allow: INVITE, BYE, CANCEL, ACK, REGISTER, NOTIFY";
stringLen = strlen(allowLine);
memcpy(&(response[count]), allowLine, stringLen);
count += stringLen;
response[count] = kCrgRtn;
response[count + 1] = kLnFd;
count += 2;

/* Finish up with the Contact and Content-Length headers */
matchFound = 0;
while (!matchFound && bufCount < msgLen) {
    if (buffer[bufCount] == kCrgRtn &&
        buffer[bufCount + 1] == kLnFd &&
        buffer[bufCount + 2] == 'C' &&
        buffer[bufCount + 3] == 'o' &&
        buffer[bufCount + 4] == 'n' &&
        buffer[bufCount + 5] == 't' &&
        buffer[bufCount + 6] == 'a') {
        matchFound = 1;
    }
    bufCount++;
}

if (bufCount >= msgLen) {
    printf(">>> ERROR parsing <Contact> header(1).\n"");
    return;
} else if (matchFound) {
    bufCount--;
}

while (buffer[bufCount] != '<' && bufCount < msgLen) {
    bufCount++;
}
if (bufCount >= msgLen) {
    printf(">>> ERROR parsing <Contact> header(2).\n"");
    return;
}
char contactLine[] = "Contact: ";
stringLen = strlen(contactLine);
memcpy(&(response[count]), contactLine, stringLen);
count += stringLen;

while (buffer[bufCount] != kCrgRtn && bufCount < msgLen) {
    response[count] = buffer[bufCount];
    count++;
    bufCount++;
}
if (bufCount >= msgLen) {
    printf(">>> ERROR parsing <Contact> header(3).\n"");
    return;
}
response[count] = kCrgRtn;
response[count + 1] = kLnFd;
count += 2;
bufCount += 2;
char cLengthLine[] = "Content-Length: 0";
stringLen = strlen(cLengthLine);
memcpy(&(response[count]), cLengthLine, stringLen);
count += stringLen;
response[count + 0] = kCrgRtn;
response[count + 1] = kLnFd;
response[count + 2] = kCrgRtn;
response[count + 3] = kLnFd;
count += 4;
/* Now to send out the packet */
if ((sendLen = sendto(sockNum, response, count, 0,
    (struct sockaddr*)&client, cLen)) < 0) {
    perror("sendto");
    exit(-1);
}
printf(">>> Response packet sent.\n"); // TEST PRINT
else {
    printf(">>> Unknown client message type. Ignoring.\n");
} return;

/**
 * Generate a provisional response to client's INVITE request, and send immediately
 * msgBuf       the client's message
 * msgLen       length of client's message
 * sockNum      socket to send SIP info out on
 * client       client's internet address information
 */
void tryingResponse(char msgBuf[kMaxSize], int msgLen, int sockNum, struct sockaddr_in client) {
    int msgCount = 0, rspCount = 0, sLen = 0, matchFound = 0;
    char bufChar = 0;
    char response[kMaxSize];
    socklen_t cLen = sizeof(client);

    // parse the message and send a status 100 trying response; start with status line
    memset(response, 0, kMaxSize);
    char statusLine[] = "SIP/2.0 100 Trying";
    sLen = strlen(statusLine);
    memcpy(&(response[rspCount]), statusLine, sLen);
    response[sLen + 0] = kCrgRtn;
    response[sLen + 1] = kLnFd;
    rspCount = rspCount + sLen + 2;

    // for the Via header, this attempts to respond w/o inserting received tag
    while (!matchFound && msgCount < msgLen) {
        if (msgBuf[msgCount] == kCrgRtn &&
            msgBuf[msgCount + 1] == kLnFd &&
            msgBuf[msgCount + 2] == 'V' &&
            msgBuf[msgCount + 3] == 'i' &&
            msgBuf[msgCount + 4] == ':') {
            matchFound = 1;               // set to 'true'
            msgCount += 2;               // Account for loop logic
            bufChar = msgBuf[msgCount];
            while (bufChar != kCrgRtn) {
                response[rspCount] = bufChar;
                rspCount++;
                msgCount++;
                bufChar = msgBuf[msgCount];
            }
            response[rspCount + 0] = kCrgRtn;
            response[rspCount + 1] = kLnFd;
            rspCount += 2;
        }
        msgCount++;
    }
    msgCount++;          // Account for loop logic
    if (msgCount >= msgLen) {
        printf("ERROR (tryingResponse) writing Via header; response not sent.\n");
        return;
    }

    // The From, To, Call-ID and CSeq headers are all copied from the client message
    for (matchFound = 0; matchFound < 4; matchFound++) {
        while (msgBuf[msgCount] != kCrgRtn) {
            response[rspCount] = msgBuf[msgCount];
            rspCount++;
            msgCount++;
        }
        response[rspCount + 0] = kCrgRtn;
        response[rspCount + 1] = kLnFd;
        rspCount += 2;
        msgCount += 2;
        if (msgCount >= msgLen) {
            printf(">>> Unknown client message type. Ignoring.\n");
        } return;
    }
}
printf("ERROR (tryingResponse) writing header \%d; response not sent.\n", matchFound + 2);
return;
}

// Attach the final header, Content-Length
char cLenLine[] = "Content-Length: 0";
sLen = strlen(cLenLine);
memcpy(&(response[rspCount]), cLenLine, sLen);
response[rspCount + sLen + 0] = kCrgRtn;
response[rspCount + sLen + 1] = kLnFd;
response[rspCount + sLen + 2] = kCrgRtn;
response[rspCount + sLen + 3] = kLnFd;
rspCount = rspCount + sLen + 4;

/* Send the response out */
if ((sendto(sockNum, response, rspCount, 0, (struct sockaddr*)&client, cLen)) < 0) {
    perror("sendto");
    exit(-1);
} else {
    printf("(tryingResponse) Response sent to client.\n");
    return;
}

/**
 * A request message is forwarded from client to proxy, or vice versa
 * 
 * msgBuf      the message
 * msgLen      message length
 * sockNum     communication socket
 * proxy       address of recipient
 * localIP     string representation of local IP address
 */
void insertAndForward(char msgBuf[kMaxSize], int msgLen, int sockNum, struct sockaddr_in remote, char* localIP) {
    int fwdCount = 0, msgCount = 0, sLen = 0;
    char forward[kMaxSize];

    memset(forward, 0, kMaxSize);
    /* Copy the request line from the client's message to the forwarding message */
    while (msgBuf[msgCount] != kCrgRtn && msgCount < msgLen) {
        forward[fwdCount] = msgBuf[msgCount];
        fwdCount++;
        msgCount++;
    }

    if (msgCount >= msgLen) {
        printf("(insertAndForward) ERROR writing the request line.\n");
        return;
    }

    forward[fwdCount + 0] = kCrgRtn;
    forward[fwdCount + 1] = kLnFd;
    fwdCount += 2;
    msgCount += 2;

    /* Write the proxy's own Via next */
    char viaLine1[] = "Via: SIP/2.0/UDP ";
    sLen = strlen(viaLine1);
    memcpy(&(forward[fwdCount]), viaLine1, sLen);
    fwdCount += sLen;
    sLen = strlen(localIP);
    memcpy(&(forward[fwdCount]), localIP, sLen);
    fwdCount += sLen;
    char viaLine2[] = ":5060;branch=z9hG4bK-proxy";
    sLen = strlen(viaLine2);
    memcpy(&(forward[fwdCount]), viaLine2, sLen);
    fwdCount += sLen;
    forward[fwdCount + 0] = kCrgRtn;
    forward[fwdCount + 1] = kLnFd;
    fwdCount += 2;
/* Copy the rest of the client's message into the forwarding message */
while (msgCount < msgLen) {
    forward[fwdCount] = msgBuf[msgCount];
    fwdCount++;
    msgCount++;
}

/* Send the response out */
if ((sendto(sockNum, forward, fwdCount, 0, (struct sockaddr*)&remote, sizeof(remote))) < 0) {
    perror("sendto");
    exit(-1);
}
printf("(insertAndForward) message forwarded.\n"); // TEST PRINT
return;
}

/**
 * confirmCancel(char msgBuf[kMaxSize], int msgLen, int sockNum, struct sockaddr_in client) {
 *       int msgCount = 0, rspCount = 0, sLen = 0, matchFound = 0;
 *       char bufChar = 0;
 *       char response[kMaxSize];
 *       socklen_t cLen = sizeof(client);
 *
 *       // parse the message and send a status 100 trying response; start with status line
 *       memset(response, 0, kMaxSize);
 *       char statusLine[] = "SIP/2.0 200 OK*";
 *       sLen = strlen(statusLine);
 *       memcpy(&(response[rspCount]), statusLine, sLen);
 *       response[sLen + 0] = kCrgRtn;
 *       response[sLen + 1] = kLnFd;
 *       rspCount = rspCount + sLen + 2;
 *
 *       // for the Via header, this attempts to respond w/o inserting received tag
 *       while (!matchFound && msgCount < msgLen) {
 *           if (msgBuf[msgCount] == kCrgRtn &&
 *               msgBuf[msgCount + 1] == kLnFd &&
 *               msgBuf[msgCount + 2] == 'V' &&
 *               msgBuf[msgCount + 3] == 'i' &&
 *               msgBuf[msgCount + 4] == ':') {
 *               matchFound = 1;                 // set to 'true'
 *               msgCount += 2;
 *               bufChar = msgBuf[msgCount];
 *               while (bufChar != kCrgRtn) {
 *                   response[rspCount] = bufChar;
 *                   rspCount++;
 *                   msgCount++;
 *               }
 *               response[rspCount + 0] = kCrgRtn;
 *               response[rspCount + 1] = kLnFd;
 *               rspCount += 2;
 *           }
 *           msgCount++;
 *       }
 *       msgCount++; // Account for loop logic
 *       if (msgCount >= msgLen) {
 *           printf("ERROR (confirm Cancel) writing Via header; response not sent.\n");
 *           return;
 *       }
 *
 *       // The From, To, Call-ID and CSeq headers are all copied from the client message
 *       for (matchFound = 0; matchFound < 4; matchFound++) {
 *           while (msgBuf[msgCount] != kCrgRtn) {
 *               response[rspCount] = msgBuf[msgCount];
 *               rspCount++;
 *               msgCount++;
 *           }
 *           response[rspCount + 0] = kCrgRtn;
 *       }
 */
response[rspCount + 1] = kLnFd;
rspCount += 2;
msgCount += 2;
if (msgCount >= msgLen) {
    printf("ERROR (confirmCancel) writing header %d; response not sent.\n", matchFound + 2);
    return;
}

// Attach the final header, Content-Length
char cLenLine[] = "Content-Length: 0\n";
sLen = strlen(cLenLine);
memcpy(&(response[rspCount]), cLenLine, sLen);
response[rspCount + sLen + 0] = kLnFd;
response[rspCount + sLen + 1] = kCrgRtn;
response[rspCount + sLen + 2] = kLnFd;
response[rspCount + sLen + 3] = kCrgRtn;
rspCount = rspCount + sLen + 4;

/* Send the response out */
if ((sendto(sockNum, response, rspCount, 0, (struct sockaddr*)&client, cLen)) < 0) {
    perror("sendto");
    exit(-1);
}
printf("(confirmCancel) Response sent to client.\n");
return;

/**
 * A status response is forwarded from client to proxy or vice versa
 */
void stripAndForward(char msgBuf[kMaxSize], int msgLen, int sockNum,
struct sockaddr_in remote, char* localIP) {
    int msgCount = 0, fwdCount = 0, matchFound = 0;
    char forward[kMaxSize];
    memset(forward, 0, kMaxSize);
    while (!matchFound && msgCount < msgLen) {
        if (msgBuf[msgCount] == kCrgRtn &&
            msgBuf[msgCount + 1] == kLnFd &&
            msgBuf[msgCount + 2] == 'V' &&
            msgBuf[msgCount + 3] == 'i' &&
            msgBuf[msgCount + 4] == 'a' &&
            msgBuf[msgCount + 5] == ':') {
            matchFound = 1;
        }
        forward[fwdCount] = msgBuf[msgCount];
        fwdCount++;
        msgCount++;
    }
    if (msgCount >= msgLen) {
        printf("(stripAndForward) ERROR detecting Via header; message not forwarded.\n");
        return;
    }
    forward[fwdCount] = msgBuf[msgCount];
    fwdCount++;
    msgCount++;

    /* At this point, msgBuf points to the first Via; the one to strip */
    /* matchFound = 0; */
    while (!matchFound && msgCount < msgLen) {  
        if (msgBuf[msgCount] == kCrgRtn &&
            msgBuf[msgCount + 1] == kLnFd) {
            matchFound = 1;
        }
        msgCount++;
    }
    if (msgCount >= msgLen) {
        printf("(stripAndForward) ERROR skipping over 1st Via header.\n");
        return;
    }
    forward[fwdCount] = msgBuf[msgCount];
    fwdCount++;
    msgCount++;
msgCount++;

/* msgCount now points to the second Via, where copying shall resume */
while (msgCount < msgLen) {
    forward[fwdCount] = msgBuf[msgCount];
    fwdCount++;
    msgCount++;
}

/* Send the response out */
if ((sendto(sockNum, forward, fwdCount, 0,
            (struct sockaddr*)&remote, sizeof(remote))) < 0) {
    perror("sendto");
    exit(-1);
}
printf("(stripAndForward) message forwarded.\n"); // TEST PRINT
return;
OpenWRT Makefiles

This information goes inside the Makefile found in the /sips/src directory. All it does is compile and clean sips when invoked.

```make
all: sips
%.o: %.c
  $(CC) $(CFLAGS) $(EXTRA_CFLAGS) -c -o $@ $^
sips: sips.o
  $(CC) -o $@ $^
clean:
  rm -f *.o sips
```

This information goes inside the Makefile found in the /sips directory. This valid Makefile allows the OpenWRT kernel configuration menu to list the SIP server as an available package to be included in kernel compilation.

```make
# Copyright (C) 2006-2009 OpenWrt.org
#
# This is free software, licensed under the GNU General Public License v2.
# See /LICENSE for more information.
include $(TOPDIR)/rules.mk

PKG_NAME:=sips
PKG_VERSION:=0.01
PKG_RELEASE:=1.1

PKG_BUILD_DIR:=$(BUILD_DIR)/sips

include $(INCLUDE_DIR)/package.mk

define Package/sips
  SECTION:=utils
  CATEGORY:=Utilities
  TITLE:=Session Initiation Protocol server
endef

define Package/sips/description
  My Session Initiation Protocol server
endef

define Build/Prepare
  mkdir -p $(PKG_BUILD_DIR)
  $(CP) ./src/* $(PKG_BUILD_DIR)/
endef

define Package/sips/install
  $(INSTALL_DIR) $(1)/sbin
  $(INSTALL_BIN) $(PKG_BUILD_DIR)/sips $(1)/sbin/
endef

$(eval $(call BuildPackage,sips))
```
Glossary

ATA: Analog Telephony Adapter
This device interfaces between standard land line telephones and a network for Voice over IP communications.

DHCP: Dynamic Host Configuration Protocol
This is an essential protocol required to label newly-connected devices with an existing network. Devices use this protocol to acquire an IP address.

HTTP: Hypertext Transfer Protocol
This common protocol is used by web servers and clients to fulfill and display media requests. This protocol is easy to implement and utilize, making it a model for other protocols.

IP: Internet Protocol
This protocol lies low in the OSI protocol stack. All protocols at or above its stack level require the knowledge and use of IP addresses for each network entity.

LAN: Local Area Network
Relative to one device, a LAN is the local network shared with all other devices. This network domain is usually separated by a router. Linksys routers may support multiple LANs, but the single LAN by default is all this project needs.

OSI: Open Systems Interconnection (protocol stack)
This is a commonly accepted model depicting the separation of network protocols. Each protocol which builds on a previous one is said to be of a higher layer.

UDP: User Datagram Protocol
This protocol resides just above IP, and maintains its connectionless attribute. This allows a greater network speed, but does not implement any sort of reliable transmission.

WAN: Wide Area Network
Relative to one device, a WAN is the global network residing on the other side of a router which hosts the device’s LAN. Linksys routers are only connected to one WAN.
Bibliography

