DESIGN AND IMPLEMENTATION OF A VoIP PHONE USING A MICROPROCESSOR BOARD

Metin ÇIKRIKÇIOĞLU

Master of Science Thesis

Computer Engineering Program

August-2008

JÜRİ VE ENSTİTÜ ONAYI

Metin Çıkrıkçıoğlu'nun "**Mikroişlemci Tabanlı Bir VoIP Telefon Tasarımı ve Gerçeklemesi**" başlıklı **Bilgisayar Mühendisliği** Anabilim Dalındaki, Yüksek Lisans Tezi 02.07.2008 tarihinde, aşağıdaki jüri tarafından Anadolu Üniversitesi Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin ilgili maddeleri uyarınca değerlendirilerek kabul edilmiştir.

	Adı-Soyadı	**İmza**
Üye (Tez Danışmanı)	**: Yard. Doç. Dr. CÜNEYT AKINLAR**	**…..……..**
Üye	**: Yard. Doç. Dr. YUSUF OYSAL**	**…..……..**
Üye	**: Yard. Doç. Dr. AHMET YAZICI**	**..……..**

Anadolu Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun …… tarih ve …… sayılı kararıyla onaylanmıştır.

Enstitü Müdürü

ABSTRACT

Master of Science Thesis

DESIGN AND IMPLEMENTATION OF A VoIP PHONE USING A MICROPROCESSOR BOARD

Metin ÇIKRIKÇIOĞLU

Anadolu University
Graduate School of Sciences
Computer Engineering Program

Supervisor: Assist. Prof. Dr. Cüneyt AKINLAR
2008, 65 pages

Transmission of voice over an IP network (VoIP) is very important as it lays the ground for data and voice network convergence, which allows tremendous cost savings for both enterprises and carriers. Session Initiation Protocol (SIP), developed by the Internet Engineering Task Force (IETF), is the emerging VoIP signaling protocol that is used to setup, terminate and modify associations between Internet end systems, including conferences and point-to-point calls. With the emergence of VoIP using SIP, many SIP phones have emerged on the market. These phones both come in software, called a SoftPhone, and hardware varieties. A hardware SIP phone is implemented using a cheap microprocessor board with a network interface such as Ethernet or 802.11. Such phones must have a full-fledged SIP stack with a small footprint requiring minimal system resources.

In this thesis, our goal is to design and implement a hardware SIP phone using a microprocessor board. The main motivation is to obtain a standards-compliant, small footprint SIP stack implementation that would require minimal resources when run in an embedded system. We have implemented a layered SIP stack and deployed it on the Ubicom microprocessor board. To test the implemented SIP stack, a SIP proxy server, a SIP presence server and a SIP registrar have also been implemented. Finally, we have integrated a commercial SIP SoftPhone, X-Lite, into the system, and successfully tested the compatibility of our SIP stack implementation with X-Lite.

Keywords: VoIP, SIP, Embedded System, Proxy Server, Presence Server, Registrar Server

ÖZET

Yüksek Lisans Tezi

MİKROİŞLEMCİ TABANLI BİR VoIP TELEFON TASARIMI VE GERÇEKLEMESİ

Metin ÇIKRIKÇIOĞLU

Anadolu Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Yard. Doç. Dr. Cüneyt AKINLAR
2008, 65 sayfa

IP ağları üzerinden ses iletimi (VoIP), veri ve ses ağlarını bir birine yakınlaştırıp, şirketlerin maliyetlerini büyük ölçüde azaltmasına yardımcı olduğu için çok önemli hale gelmiştir. Internet Engineering Task Force (IETF) tarafından geliştirilen Session Initiation Protocol (SIP), İnternet uç sistemleri arasında oturumların kurulmasını, sonlandırılmasını ve değiştirilmesini sağlayan ve aynı zamanda konferansların yapılmasına ve uçtan uca aramalara imkan veren ve son zamanlarda öne çıkan bir oturum kurma protokolüdür. SIP kullanan VoIP'in ortaya çıkmasıyla, pazarda çok sayıda SIP telefon görünür hale geldi. Bu telefonlar hem SoftPhone denen yazılımlar şeklinde hem de donanım olarak yapıldılar. Donanımsal bir SIP telefonu, üzerinde Ethernet veya 802.11 ağ arayüzü olan ucuz bir mikroişlemci kartı ile gerçeklenebilir hale geldi. Bu telefonlar, minimum sistem kaynağı gereksinimi olan eksiksiz bir SIP yığınına sahip olmalıdır.

Bu tezde amacımız, gömülü bir kart kullanarak, donanımsal bir SIP telefonu tasarlamak ve gerçeklemektir. Ana motivasyonumuz, standarda uyumlu, minimum SIP yığın gereksinimlerini karşılayan ve gömülü bir sistemde çalıştığında en alt seviyede kaynak kullanacak bir SIP yığını elde etmektir. Bu tezde katmanlı bir SIP yığını gerçekledik, ve bu yığını Ubicom mikroişlemci kartına yükledik. Gerçeklenen SIP yığınını test etmek için bir SIP vekil sunucu, bir SIP kaydedici sunucu ve bir de SIP durum sunucusu gerçeklendi. Son olarak yazılım tabanlı ticari X-Lite telefonu sisteme entegre edildi, ve SIP yığın gerçeklememizin uyumluluğu başarılı bir şekilde test edildi.

Anahtar Kelimeler: VoIP, SIP, Gömülü Sistem, Vekil Sunucu, Durum Sunucusu, Kaydedici Sunucu

CONTENTS

LIST OF FIGURES

ABBREVATIONS

ARM : Advanced RISC Machine

DHCP : Dynamic Host Control Protocol

FQDN : Fully Qualified Domain Name

HTTP : Hypertext Transfer Protocol

RISC : Reduced Instruction Set Computer

RTP : Real-Time Protocol

SDP : Session Description Protocol

SIP : Session Initiation Protocol

SMTP : Simple Mail Transfer Protocol

TCP : Transport Control Protocol

UDP : User Datagram Protocol

URI : Uniform Resource Identifier

1. INTRODUCTION

Voice over IP (VoIP) offers the opportunity to design a global multimedia communications system that will eventually replace the existing century-old telephony infrastructure [11]. This is due to the fact that VoIP offers enormous cost savings and ease of development both for enterprises and carriers, while integrating new services [17]. Internet telephony differs from Internet multimedia streaming primarily in the control and establishment of sessions, i.e., "signaling". While we usually assume that a stored media resource is available at a given location, participants in a phone call are not so easily located. Personal mobility, call delegation, availability, and willingness to communicate make the process of signaling more complex. Session Initiation Protocol (SIP) is the emerging VoIP signaling protocol, which is used to set up, modify and tear down multimedia communication sessions over the Internet [7]. SIP also provides instant messaging and presence to end points [19-21].

This thesis concentrates on the design and implementation of a SIP stack on an embedded system. The goal is to obtain a small-footprint, standards-compliant SIP stack that would require minimal resources in an embedded system. To test the designed SIP stack, an instant messaging and presence awareness application have been implemented. In order to fully provide a working system, a proxy server and a registrar server have also been implemented for the PC architecture. To test the compatibility of our SIP stack with existing SIP stack implementations, a commercial SoftPhone from CounterPath Corporation, called X-Lite [10], is used. We demonstrate that our SIP stack implementation not only requires a meager 10 Kbytes, but also interoperates seamlessly with X-Lite SoftPhone.

Our SIP stack has a layered architecture. Specifically, there are four layers. The first is the parsing/generation layer, which accomplishes SIP message generation and parsing. The next is the transport layer, which provides networking by using TCP or UDP. The transport layer achieves retransmissions of requests or responses,

matching of responses to requests and handling of timeouts. The third layer is the transaction layer, which initiates transactions and uses them for specific SIP methods. Finally there comes the application layer, which provides message transmission, reception, registration and getting/sending presence information.

As our embedded system, we use the Ubicom board, which has a unique architecture for networking and media processing. Ubicom has a 32-bit processor called IP5K. The processor has powerful multithreading functionality, called Multithreaded Architecture for Software I/O (MASI), which allows the execution of threads at fully deterministic time intervals.

To have users interact with the Ubicom board and to display messages and presence information, a companion embedded board is used. This is a MCBSTR9 Evaluation Board from Keil Company, and has an ARM core processor. The board has an LCD display and some buttons, which are required for user interaction. The processor used on this board is STR9 produced by STMicroelectronics Company, and has an ARM9 core. In order to achieve the communication between Ubicom board and the MCBSTR9 companion board, we have implemented a software module that runs over TCP/IP.

Our SIP stack for the Ubicom board is implemented using the C programming language, and consists of 4362 lines of C code. When compiled, the stack footprint is 10,348 bytes, which easily fits on the Ubicom's 196,608 bytes flash memory

SIP proxy and registrar servers are implemented using Java SIP servlet of cafesip, which is an open source project. These java SIP servlets are called Jiplets. Cafesip provides a jiplet container, which allows developing SIP server-side applications. Proxy and registrar server software is developed using Jiplet API, and is run in a Jiplet Container. The container enables application developer to create server-side SIP applications using a component-based model similar to the J2EE architecture.

The rest of the thesis is organized as follows: In Chapter 2, components of the SIP architecture is explained in a detailed manner. In Chapter 3, we talk about the details of our SIP stack, proxy and registrar software. We also give the architectural details of the embedded boards used in this project. Finally, chapter 4 describes the integration and testing of the developed system with the X-Lite SoftPhone.

2. SESSION INITIATION PROTOCOL (SIP)

It is very important to be able to create and manage sessions among applications in the Internet for such purposes as data exchange and communication. But there are some difficulties to make this reality possible. In particular, users may have transient positions, and may use different media types to communicate. This dynamic nature of the users' locations and preferences necessitates a signaling protocol for on-demand session establishment. Session Initiation protocol (SIP) [1] developed by Internet Engineering Task Force (IETF) is one such signaling protocol, which helps an Internet endpoint discover other endpoints and agree on a description of the session that they would like to set up. SIP is a general protocol that allows Internet end points to send registrations, invitations for sessions, and other requests with no dependency to the underlying transport protocol.

2.1 SIP Functionality

SIP is a text-encoded protocol based on elements from HyperText Transport Protocol (HTTP) [2] and also Simple Mail Transport Protocol (SMTP) [3]. It is an application layer protocol [18]. It can establish, modify and terminate multimedia sessions. SIP can also invite participants to existing sessions. SIP supports name mapping and redirection services so that personal mobility can be achieved. In other words, a user just has to deal with one external identifier regardless of her/his network location. There are five facets of establishing and terminating multimedia sessions using SIP. In a nutshell these are:

User Location

Frequently a user might be reached at several locations during the day. For example, a student using computer laboratory in the university generally works on a different computer each time s/he logs into the system. Therefore, s/he can be reached at different IP addresses. Another user may want to receive session invitations at lunch in her/his office using desktop computer, using a notebook at home in the evening, and at a mobile terminal when s/he is traveling [4].

Uniform Resource Identifier (URI) and Registration concepts serves this user identity and location problems. A person using SIP should have SIP URI, which is similar to an email address and looks like: sip:bob@somecompany.com. Bob registers to a registrar using this URI. He will then be accessible through his URI.

User Availability

Users can define if they are willing to accept a call.

User Capabilities

During session setup, media and media parameters can be determined.

Session Setup

Session parameters are established for both called and calling end.

Session Management

Transfer or termination of sessions and changing session parameters and invoking services are legitimate.

2.2 SIP Enabled Network

Figure 2.1 shows the components of a SIP-enabled IP communication network. SIP servers accomplish functions specific to SIP. SIP servers can be stateless, similar to other Internet devices. SIP servers can be deployed in geographically distributed clusters to avoid service failures. All this ensures very fast response time and avoids failures in the network to disable calls, since the call state is kept at the periphery of the network and not in the core. Users do not depend on any potential central points of failure in the network and can communicate as long as they have working end devices.

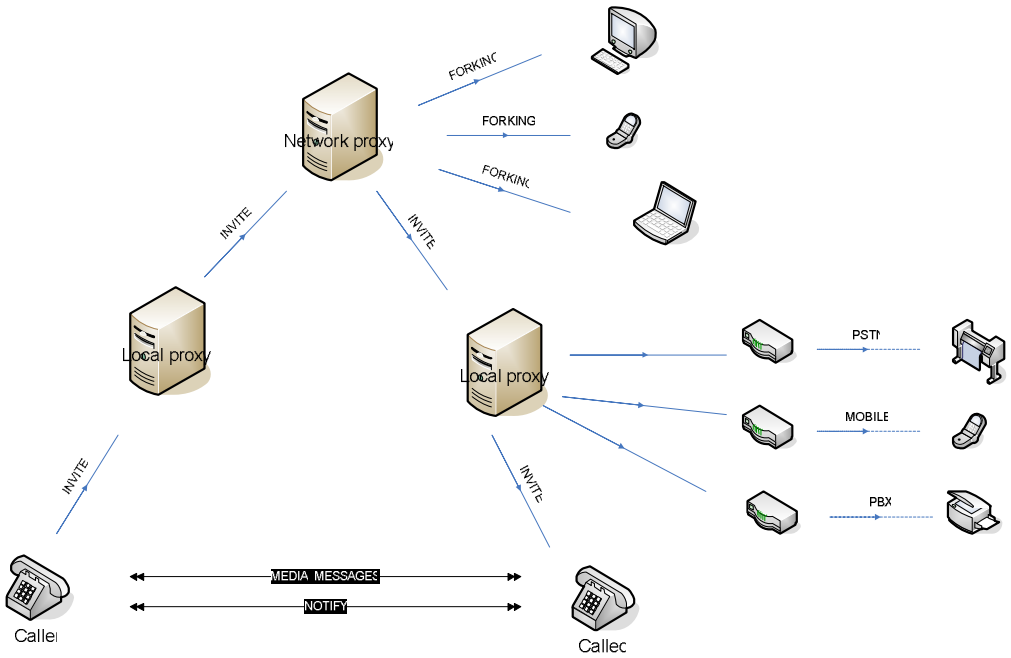

Figure 2.1 Components of a SIP Enabled IP-Communication Network

A caller can send an INVITE message to establish a session to the called party, without knowing exactly where the other endpoint may be, and the SIP servers will route the call to the destination. The route to the destination can be forked in the network in order to find the other endpoint. The same infrastructure can also serve for an instant message and presence service. A watcher can subscribe to a presentity, and receive NOTIFY messages from the presentity. The watcher and presentity can exchange short text messages using SIP itself, or Real-Time Protocol (RTP) packets for any other communication media: audio, various data applications, video, or games for instant communications.

Major benefits of SIP for end-points are the following:

Web-style and telephony-type addressing: SIP devices can use URIs that are location-independent, and URLs that point to a specific host. Addresses can take the form of e-mail addresses or telephone numbers.

Registration: Devices connected to the network are registered in order to route calls to and from the device. Users may register themselves using their URIs to get access to their particular information and services, independent from the device registration. This is similar to e-mail access from web kiosks or Internet cafes. Such dynamic routing to/from the user is accomplished without needing "switch translations" or other static routing tables.

Security: SIP is designed to use the Internet security mechanisms so as to protect sensitive signaling information from various types of attacks. User location and traffic patterns can be kept confidential. SIP security can be quite complex and uses the advances in all generic IP security mechanisms.

Redirect: A SIP server can redirect a request to another address.

Forking: A request from a user can be forwarded in several directions simultaneously, as, for example, when trying various locations where the called party may be found.

Rendezvous and presence: The active form of rendezvous consists of routing a request for call setup to another server or endpoint where the desired service may be performed, such as communication with an individual, or with a machine. The passive form of rendezvous consists of presence information (that is, letting someone know that a party of interest is connected to the network and its communication state, such as available or busy).

Mobility: Users may have many communication devices such as phones, fax machines, computers, palm computers, and pagers, at home, at work, and while traveling. User devices can be attached to various types of networks, if proper gateways are provided: IP, PSTN, mobile telephony, wireless mobile data, or paging. SIP call setup can proceed without regard to the type of network or type of device the parties may use at a certain instance.

User preferences: Callers can specify how servers and the network should handle their requests, and also specify what type of service is desired or acceptable, whom they would like to reach, and whom they would like to avoid (for example, to avoid making calls to busy lines or to speak to machines). Called parties can specify how to handle incoming calls, depending on a very large set of criteria, such as who the caller is, where the call is coming from, time of day, the communication device, and others.

Routing control: The route taken by SIP messages can be specified and recorded for various services [5].

2.3 SIP Messages

A SIP Message could be request or response, which could be sent from client to the server or from server to the client. These are called, SIP Request and SIP Response. These two messages have something in common; they both have a start line, some header fields, and an empty line, which indicates the end of the message. At the end they may have an optional body [13]. Lines are distinguished from each other by two consecutive ASCII characters, which are Carriage Return (CR) and Line Feed (LF), characters 13 and 10 respectively. So the general structure of a SIP message is shown below:

```
start-line CRLF
message-header1 CRLF
message-header2 CRLF
.
.
.
CRLF
```

[message-body]

Start-line can be a request line or a response line depending on the type of the SIP message. After the start-line and each of the header lines, there has to be CRLF (Carriage Return and Line Feed characters in order), which indicates the end of the line. Before SIP message-body there has to be a secondary CRLF even if message-body does not exist in the SIP message.

2.3.1 SIP Request

Start-line of a SIP Request is a request line, which has to be ended with a CRLF. A request line contains three elements: (1) a method name, e.g., REGISTER, INVITE, ACK, BYE, CANCEL, OPTIONS, etc., (2) a request-URI, (3) the protocol version, e.g., SIP/2.0. These three elements have to be separated from each other with exactly one space character (SP). It is not allowed to have more than one space characters between these elements. At the end of the protocol version CRLF sequence must exist, which denotes the end of the line.

There are 6 major SIP requests listed in SIP RFC [16]: These are REGISTER, INVITE, ACK, CANCEL, BYE, OPTIONS. Additional methods are defined in RFC 3428, which are SUBSCRIBE and NOTIFY. REGISTER method is used for registering the client at the location server, called a registrar. INVITE, ACK and CANCEL methods are used for session setup. BYE method is used for ending a session. By using OPTIONS method, someone can query the capabilities of a server. SUBSCRIBE method is used when someone wants to know the real-time presence of her/his friend. The server uses the NOTIFY method to inform the user of her/his friend's presence status.

Another element of the request line is request-URI. The request-URI is a SIP or SIPS uri. This URI is directly related to the method of the request. It can not have unescaped characters or control characters.

And the last element of the request-line is SIP-Version. This part shows the SIP version in use. In this thesis, SIP version 2.0 is used. That's why all requests contain version as "SIP/2.0". RFC3261 requires the version to be sent in uppercase.

2.3.2 SIP Response

A SIP Response is sent for a SIP request indicating the status of the request. The only difference between a SIP Response and a SIP Request is the status line. Status line for a SIP Response contains three elements which are protocol version, status code and a textual phrase. There has to be exactly one space character between these elements and at the end of the line there has to be CRLF sequence.

SIP version is the same with SIP Request, i.e., "SIP/2.0".

Textual phrase is a representation of the response as a human understandable text. The software written for SIP doesn't have to deal with this textual phrase. It is intended only for humans. In SIP specification these are defined in English but they do not have to be in English only. Depending on the Accept-Language header field, software may choose other equivalents in any language.

The status code is a three digit integer number, which shows the result of a request. It can be thought as a result code. The status codes are grouped by the first digit. Every group, which has the same first digit, has a distinguishing property in a sense. There are six groups of error codes. The first group, error codes in the range 100-199, indicates that a request is successfully received and it is still being processed. The second group, error codes in the range 200-299, indicates that a request is received and processed successfully. The third group, error codes in the range 300-399, indicates that a separate request is required in addition to the request sent recently. This is a redirection response actually. The fourth group, error codes in the range 400-499, indicates that the receiver of the request has encountered a syntax error or cannot process the request due to some lack of implementation. The fifth group, error codes in the range 500-599, is similar to the fourth group but request

does not have to be a syntax error or a meaningful error. The last group, error codes in the range 600-699, indicates a global failure. That kind of response shows that there is a global error, so that the request cannot be processed or received by any server [1].

2.4 SIP Network Elements

SIP Networks are composed of several elements. The basic elements are user agents, proxies, registrars and redirect servers. A minimal SIP network contains at least two user agents. More complex versions may also contain other elements. Note that, these elements are considered to be logical entities, so there might be more than one logical element in one physical entity. For example one server may act as a redirect server but also as a proxy server.

2.4.1 User Agent

An Internet end point that uses SIP to find other end points in the system and to negotiate a session is called a "user agent" [6]. A user agent can be a SIP SoftPhone, software running on a computer, or a gateway that makes SIP phones to receive and make calls to the PSTN [5]. A user agent can be seen as an end system.

There are two types of user agents. The first one is UAC (UAC). UAC is capable of sending a request, and receiving responses for that request and processing them. This request generation is probably an external event such as clicking a button.

The other one is "User Agent Server" (UAS). UAS is capable of receiving a request, and generating a response corresponding request based on a configuration, user input, external event or result of a program execution.

All SIP user agents have both a UAC and a "User Agent Server". In operation both of them are used simultaneously. Compared to the HTTP protocol, SIP differs by having both a client and a server at the same time. For the HTTP protocol, a client

always only generates requests and the server always generates responses. But a SIP end system is a client and a server at the same time.

A request is sent by a UAC through some proxy servers to a "User Agent Server". These proxy servers forward messages among themselves and eventually to the "User Agent Server". The response of this request is generated by the UAC and forwarded to the "User Agent Server" [1].

User Agent Client Behavior

In this section, we explain basic UAC functionality. A UAC is responsible for generating a request, sending a request and processing incoming responses. First Request generation will be explained in a detailed manner.

A valid SIP Request generated by a UAC must contain at least the following header fields: To, From, CSeq, Call-id, Max-Forwards and Via. The existence of these header fields is a must for a SIP request. These header fields help critical message routing services, limiting request propagation, unique transactions, and ordering of the messages.

Request-URI

This value has to be the same as the URI value of To header field. There is only one exception that this value is not the same as the URI value of To header field is REGISTER method. The meaning of this URI is directly related to the method of the message.

To

To header field basically shows the recipient of the request. But this value does not have to be the ultimate destination. This header field can be a SIP or SIPS URI. RFC 3261 compels the support of SIP URI. It may have a display name.

From an implementation point of view, a UAC may acquire To header field in a number of ways. Generally user enters the To header field through a human interface, probably entering value using a keyboard etc, or choosing it from a list of

addresses. But for the sake simplicity, a user should be able to enter just “bob” in order to communicate with Bob. At this point implementation may choose bob as a left hand and the domain that the user registered as a right hand and constructs the value of To header field.

A typical example of To header field is shown below.

```
To: Bob <sip:bob@example.com>
```

In this example Bob shows the display name and the part between “>“ and “<” shows the SIP URI.

From

From header field shows the sender of the request. It also may have a display name similar to To header field, and must have a URI. There might have been some pre-defined rules on the end systems. For example an end system may have rule like when someone calls, automatically reject the call. From header field is used for these kinds of purposes. Note that, From header field does not have an IP address because it is just a logical name of the initiator.

From an implementation point of view, From header field is generally pre-configured before the sending the request and not entered more than once. But there is a possibility that there might be obligatory rules for limiting From header field defined by the administrator of the domain. If a particular UA is used by multiple users, it may have switchable profiles that include a URI corresponding to the identity of the profiled user. Recipients of requests can authenticate the originator of a request to be able to ascertain that they are who their From header field claims they are. From header field must have a tag parameter.

Call-ID

The Call-ID header field acts as the transaction identifier to tie up a series of messages. It should be the same for all requests and responses sent by either UA in a session. It has to be the same in each registration from a UA.

In a new request created by a UAC, which is not in any dialog, the Call-ID header field has to be selected by the UAC as a globally unique identifier over space and time unless overridden by method-specific behavior. All SIP UAs must achieve to guarantee that the Call-ID header fields they produce will not be the same as those generated by any other UAs.

CSeq

This header field shows the order of the transaction of messages. There are two elements in CSeq header field. One of them is sequence number and the other one is method of the request. For the requests other than which are not in a dialog, the sequence number doesn't have any importance. The sequence number is 32 bit unsigned number that's why this value cannot be bigger than 2^32. If these rules are obeyed, the implementation may choose different strategies that generate this header field. An example of CSeq header field is show below.

Call-ID: f81d234234s1d0-a765-00asdfss324f6@foo.bar.com

Max-Forwards

The Max-Forwards header field's aim is to limit the number of hops a request that can transit on the way to its ultimate point. It consists of an integer, that is, decremented by one at each point. If the Max-Forwards value reaches 0 before the request reaches its ultimate point, it will be rejected with a 483 (Too Many Hops) error response.

A UAC has to insert a Max-Forwards header field into each request it originates with a value that could be 70. This number was chosen to be sufficiently large to which makes sure that a request would not be dropped in any SIP network when there were no loops, but not so large as to consume proxy resources when a loop does occur. Lower values should be used with caution and only in networks where structure is known by the user agent.

Via

There are two aims of the Via header field. The first one is to indicate the transport of the transaction. The second one is to indicate where the response should be sent to. At each point, a Via header field is added to the request. There has to be branch parameter, which is a transaction identifier, used by both client and server. This value must be unique like call-id header field. But there is obligatory thing that each branch parameter must start with "z9hG4bk" string. The rest of the parameter is not important as long as it starts with this string and is unique.

Contact

The Contact header field provides a SIP or SIPS URI that can be used to contact that specific instance of the "User Agent" for the following requests. The Contact header field has to exist and contain exactly one SIP or SIPS URI in any request that can result in the creation of a dialog.

Which URI, SIP or SIPS URI is selected depending on the Request-URI. If Request-URI has a SIP URI, contact header field is also a SIP URI else Request-URI has a SIPS URI, then contact header field is also a SIPS URI.

Supported and Require

If there is capability of the server which can provide support for some other SIP methods, it is best to add a supported header field which contains a list of options of supported methods, to the request.

If the UAC wishes to insist that a "User Agent Server" understands an extension that the UAC will apply to the request in order to process the request, it has to insert a Require header field into the request listing the option tag for that extension. If the UAC wishes to apply an extension to the request and insist that any proxies that are traversed understand that extension, it has to insert a Proxy-Require header field into the request listing the option tag for that extension.

The destination for the request is computed. Unless there is a local policy specifying otherwise, the destination has to be defined by applying the DNS procedures as follows. If the first element in the route set indicated a strict router, the procedures have to be applied to the Request-URI of the request. Otherwise, the procedures are applied to the first Route header field value in the request (if one exists), or to the request's Request-URI if there is no Route header field exist. These procedures yield an ordered set of address, port, and transports to try. Independent of which URI is used as input, if the Request-URI specifies a SIPS resource, the UAC MUST follow the procedures as if the input URI were a SIPS URI.

Local policy might specify an alternate set of destinations to try. If the Request-URI contains a SIPS URI, any alternate destinations has to be contacted with TLS (Transport Layer Security). Beyond that, there are no restrictions on the different destinations if the request doesn't have any Route header field. This makes a simple alternative to a pre-existing route set as a way to specify an outbound proxy. However, that way for configuring an outbound proxy is not advised; a pre-existing route set with a single URI could be used instead. If the request contains a Route header field, the request could be sent to the points derived from its topmost value, but might be sent to any server that the "User Agent" is certain will honor the Route and Request-URI policies. In particular, a UAC configured with an outbound proxy could try to send the request to the location indicated in the first Route header field value instead of changing the policy of sending all messages to the outbound proxy.

This guarantees that outbound proxies that do not add Record-Route header field values will go out of the path of subsequent requests. It lets endpoints that cannot resolve the first Route URI to give that job to an outbound proxy.

Sometimes there may not be received a message at all. There might have been a communication error. This error must be treated as 408 (Request Timeout) status error code. But if there is a more serious situation like connection error, it should be treated as a 503 (Service Unavailable) status code.

A UAC may not have implemented all the status codes. Because of this, it can safely assume the status code is x00 equivalent. For example it received a 431 status code response. But it doesn't know the meaning, basically it can understand that there was a problem with the request and treat the response as if it is 400 (Bad Request) response code.

If there is more than one Via header field in the response, this response must be ignored.

User Agent Server Behavior

User Agent Server behavior is atomic. When a request is received, it immediately processed and state changes are applied. There are some steps while processing request. These are:

Method Inspection

When a request is received, firstly method of the request is inspected. If method is recognized successfully, but not supported, 405 (Method Not Allowed) response should be generated and sent. Also Allowed header field has to be added to the response, which lists the supported methods of the endpoint. If method is recognized and supported, processing the request goes on.

Header Inspection

If a "User Agent Server" doesn't recognize a header field in a request, it simply ignores it and continues processing the request. And also any corrupted header fields are ignored, too.

The To header field shows the final recipient of the request where From header field shows originator of the request. At that point, "User Agent Server" may not be the original recipient of the request because of there is possibility that it might be proxy or a redirect server. If the To header value is not its URI, then "User Agent Server" is free to process the request. But it is recommended to do process the

request. However if it decides to reject the request, it has to generate 403 (Forbidden) status code.

Request-URI is another property for “User Agent Server” for deciding if to reject or to process. If scheme of the URI is not known, then 416 (Unsupported URI Scheme) response should be sent. If URI is known but this server is not the recipient of the request, then request should be rejected by sending a 404 (Not Found) response.

After that if “User Agent Server” finds the request correct, it looks for the Require header field.

The Require header field is used by a UAC to tell a “User Agent Server” about SIP extensions that the UAC expects the “User Agent Server” to support in order to process the request properly. If a “User Agent Server” does not know an option-tag listed in a Require header field, it should answer by generating a response with status code 420 (Bad Extension). The “User Agent Server” has to add an Unsupported header field, and list in it those options it does not understand among those in the Require header field of the request.

Presumably “User Agent Server” can understand all extensions that are needed by the client; it checks the body of the message and the related header field. If there are any bodies whose type, language or encoding are not understood, and that body part is not optional, the “User Agent Server” has to reject the request with a 415 (Unsupported Media Type) response. The response has to contain an Accept header field listing the types of all bodies it understands, in the event the request contained bodies of types not supported by the “User Agent Server”. If the request contained content encodings not known by the “User Agent Server”, the response has to contain an Accept-Encoding header field listing the encodings understood by the “User Agent Server”. If the request contained content with languages not known by the “User Agent Server”, the response has to contain an Accept-Language header field

indicating the languages understood by the “User Agent Server”. Other than these checks, body handling depends on the method and type.

If there is no problem until this point, then it generates the response to the corresponding request.

For methods other than INVITE “User Agent Server” should not send a 1xx response to the requester.

From field of the response has to be equal to From header field of the request. The Call-ID header field of the response has to be equal to Call-ID header field of the request. CSeq header field of the response has to be equal to CSeq field of the request. The Via header field values in the response has to be equal Via header field values in the request and the order of the elements of header must be the same.

If a To tag exists in the request, the To header field in the response has to be equal that of the request. However, if the To header field in the request doesn’t contain a tag, the URI in the To header field in the response has to be equal the URI in the To header field; additionally, the “User Agent Server” has to add a tag to the To header field in the response. The same tag has to be used for all responses to that request, both final and provisional.

A stateless User Agent Server is a “User Agent Server” that doesn’t preserve transaction state. It just replies the request normally. If a request, which is a repetition of previous one, arrives to this kind of server, it would re-send the response, as if it encounters the request for the first time. A “User Agent Server” can’t be stateless unless the request processing for that method would always conclude in the same response if the requests are the same. There is an exception for stateless registrars, for example. Stateless “User Agent Servers” don’t use a transaction layer; they receive requests directly from the transport layer and send responses directly to the transport layer.

The most important behaviors of a stateless “User Agent Server” are; a stateless “User Agent Server” must not send 1xx responses, must not retransmit responses has to ignore ACK requests, must ignore CANCEL requests and To header tags have to be generated for responses in a stateless manner - in a manner that will generate the same tag for the same request consistently [1].

2.4.2 Proxy

In general, proxy is a server that acts as an intermediary between a workstation user and the Internet so that the enterprise can ensure security, administrative control, and caching service [14]. Basically, definition of a proxy is SIP requests are received from user agents or some other proxy and forwards the request to another location [5].

In fact it is an entity, which acts as both a server and client and its main purpose is routing the request. Destination of routing is probably a closer location or itself the destination. Of course there are possibilities for local policies for proxies, which can be applied.

A request may pass several proxies on its path to a “User Agent Server”. Each will make routing decisions, modifying the request before forwarding it to the next element. Responses will go through the same set of proxies passed by the request in the reverse order.

For a SIP element, being a proxy is a logical role. When a request comes, an element that can play the role of a proxy first concludes if it needs to respond to the request on its own. For example, the request may be corrupted or the element may need credentials from the client before acting as a proxy. The element may respond with any proper error code.

There are two types of proxy servers. First one is stateful proxy and the second one is stateless proxy. Stateless proxy doesn’t preserve any client or server transactions when it processes the requests. It just forwards every request received to

a single point by making a decision based on the request. And it simply forwards every corresponding response, which receives. When this kind of proxy forwards the request or response, it deletes all the information about the request or response. On the other hand a stateful proxy preserves the client and server transaction state machines while processing the request. It remembers all incoming requests and all requests, which is sent because of the incoming ones. The future of the messages is affected with this kind of proxy. A stateful proxy may fork a request, which means it can send the same request to more than one location. A proxy, which forks a request, must use the stateful mechanism to handle the forking situation. At this point note that a proxy may use a stateful transport mechanism such as Transport Control Protocol (TCP) but this doesn't mean proxy is stateful. It may use the same connection to send some requests which is sign of statefulness but in fact there is no statefulness from SIP point of view.

At any time a proxy server can switch from either from stateful situation to stateless situation or from stateless situation to stateful situation, as long as it prevents to switch to stateless situation, for example forking. When switch from stateful situation to stateless situation, all the states are simply discarded [1].

2.4.3 Registrar

Registrar is a server which receives REGISTER requests and preserves this data on a location service for the domain it is included [1].

Basically, registrar receives a register request and saves this information to a location service. This service could be a database. When the result of this request is successful, a 200 OK response is generated and sent to the user agent. This step is essential for the operation of SIP mechanism because if a user agent doesn't register itself to a registrar, it wouldn't be able to get a message from at all. There is generally a proxy or a redirect server with a registrar, which could send a received message to the appropriate destination based on this registration information.

When a SIP-based endpoint becomes online, the first thing it normally does is to send out a REGISTER message looking for a Registrar [15]. A registrar could be thought as a front door for a location service. Register requests are received and depending on the request, some operations such as reading or writing mappings of users are achieved. Then the proxy server, which is near the registrar takes requests other than REGISTER, and forwards them to proper destinations. An example is shown in figure 2.2

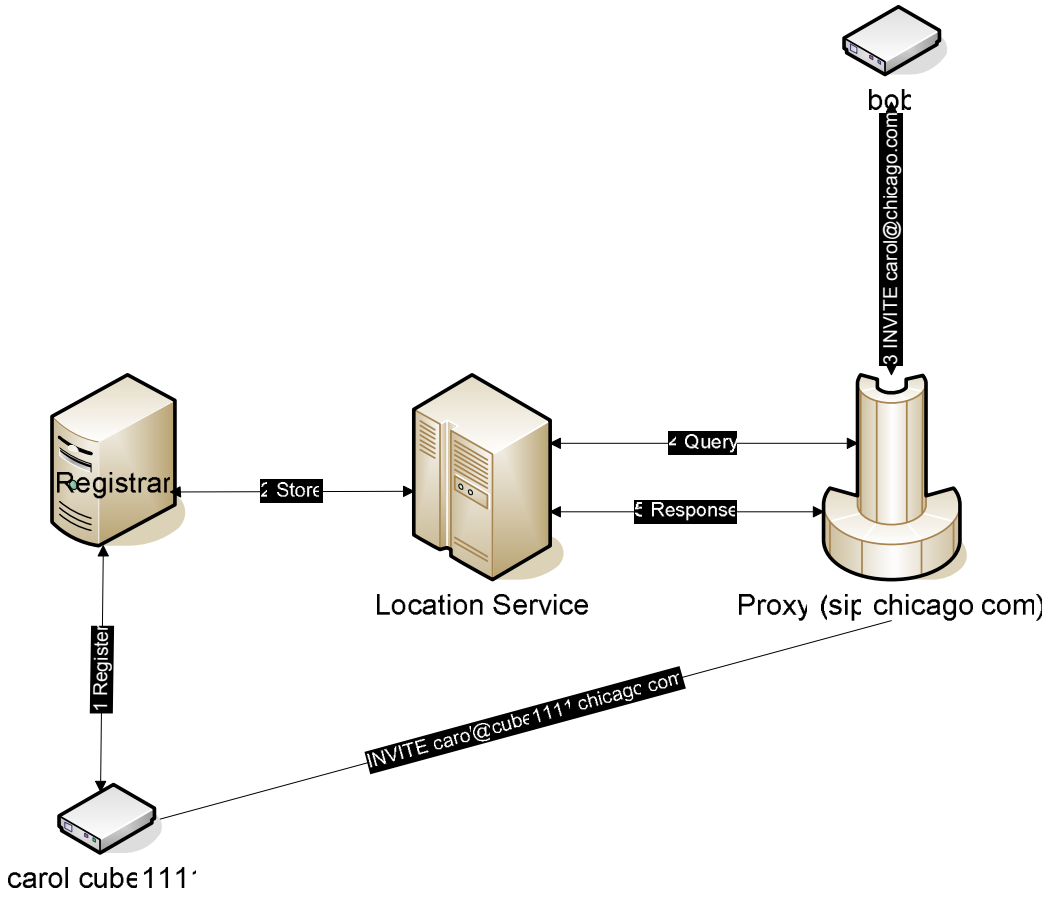

Figure 2.2 Register Example

By using REGISTER request, you can add, remove and query bindings. A REGISTER request can add a new binding between an address-of-record and one or more contact addresses. Registration for a particular address-of-record can be

performed by an authorized third party. A client can also remove bindings or query to determine which bindings are in place for an address-of record at that moment.

For a REGISTER request, Request-URI is a must. For a REGISTER request, Request-URI contains the domain of the location service. To header field in REGISTER request contains an address-of-record. This address-of-record can be anyone whose registration is to be created, queried, or modified. Different from the Request-URI, this header field contains SIP URI. From header field of a REGISTER request contains the address-of-record of the person responsible for the registration. This value is generally the same as To header field. CSeq guarantees the order of the REGISTER requests. For every REGISTER request with the same Call-ID value CSeq number must be incremented by one. Contact header field may have more than one value. The main purpose of this header field is maintaining the binding or bindings. Another important header field in a REGISTER request is Expires. Specified in seconds Expires shows how long the user agent is willing to be registered to the registrar [1].

2.4.4 Redirect Server

A redirect server is a server that generates 3xx responses and returns a list of alternate URIs of the person being queried.

In proxy servers not depending on being a stateless or stateful, all communication goes through proxy back and forth. That sometimes requires huge resources in terms of computing and networking. That's why redirect servers are defined which just generates some alternate URIs and get out from the loop of the transactions which reside between client and servers. This approach, which takes the responsibilities from core and gives them to the edges, allows great network and computing scalability. This kind of server doesn't generate a SIP request for its own. It just accepts or refuses a request. When it accepts the request, it gathers the information from a location service like a registrar and sends them as a final response

to the UAC. When a redirect server accepts the request, it will fill the contact header field with a list of alternative locations [1].

2.5 Basic SIP Operation

This section includes a simple example of SIP operation. Example consists of signal of desire to communicate, negotiation of session parameters to establish the session, and closing of the session. Figure 2.3 shows a typical example of a SIP message exchange between two users, Alice and Bob. In this example, Alice uses a SIP application on her PC to call Bob on his SIP phone over the Internet. Also there are two SIP proxy servers that for Alice and Bob to achieve the session establishment.

Alice calls Bob using his SIP URI, which is sip:bob@bloxy.com. "bloxy.com" is the domain of Bob's SIP service provider. Alice's SIP URI is sip:alice@atlanta.com. Since SIP is based on a HTTP-like request/response transaction model, each transaction consists of a request that invokes a specific method or function, on the server and at least one response. In this example, the transaction begins with Alice's SoftPhone sending an invite request addressed to Bob's SIP URI. INVITE is an example of a SIP method that specifies the action that the requestor (Alice) wants the server (Bob) to take. The INVITE request contains a number of header fields. Header fields are named attributes that provide additional information about a message.

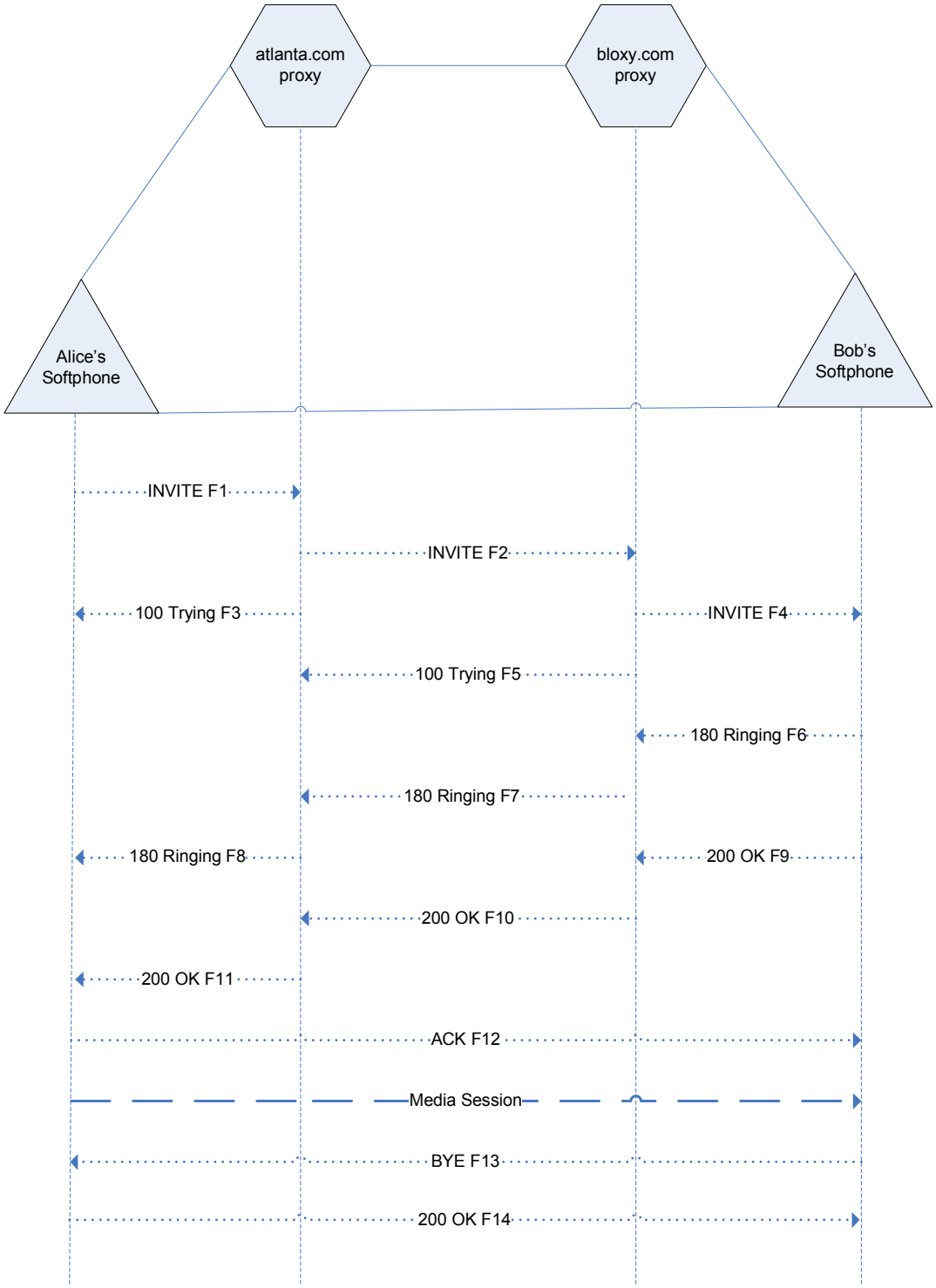

Figure 2.3 SIP Session Setup Example with SIP Trapezoid

The ones present in an INVITE include a unique identifier for the call, the destination address, Alice's address, and information about the type of session that Alice wishes to establish with Bob. The INVITE (message F1 in Figure 2.3) may look like this:

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/sdp
Content-Length: 142
(SDP is not shown)
```

The first line the ASCII encoded message contains the method name, which is the case INVITE now. The following lines are a list of header fields. These fields are compulsory fields.

Via contains the address (pc33.atlanta.com) at which Alice is expecting to receive responses to this request. It also contains a branch parameter that identifies this transaction.

To contains a display name (Bob) and a SIP or SIPS URI (sip:bob@biloxi.com) towards which the request was originally directed.

From also contains a display name (Alice) and a SIP or SIPS URI (sip:alice@atlanta.com) that indicate the originator of the request. This header field also has a tag parameter containing a random string (1928301774) that was added to the URI by the SoftPhone. It is used for identification purposes.

Call-ID contains a globally unique identifier for this call, generated by the combination of a random string and the SoftPhone's host name or IP address. The combination of the To tag, From tag, and Call-ID completely defines a peer-to-peer SIP relationship between Alice and Bob and is referred to as a dialog.

CSeq or Command Sequence contains an integer and a method name. The CSeq number is incremented for each new request within a dialog and is a traditional sequence number.

Contact contains a SIP or SIPS URI that represents a direct route to contact Alice, usually composed of a username at a fully qualified domain name (FQDN). While an FQDN is preferred, many end systems do not have registered domain names, so IP addresses are permitted. While the Via header field tells other elements where to send the response, the Contact header field tells other elements where to send future requests.

Max-Forwards serves to limit the number of hops a request can make on the way to its destination. It consists of an integer, that is, decremented by one at each hop.

Content-Type contains a description of the message body.

Content-Length contains an octet (byte) count of the message body.

Alice's SoftPhone doesn't know the location of Bob's location. That's why her SoftPhone sends the INVITE to the proxy which servers for her domain, atlanta.com. atlanta.com's IP address is already either stored in her SoftPhone or discovered using Dynamic Host Control Protocol (DHCP).

atlanta.com is a SIP server, which is a type of proxy server. A proxy server receives SIP requests and forwards them to for the requesters. In this example proxy server receives Alice's INVITE request and sends 100 Trying response to Alice's SoftPhone. 100 Trying response means, proxy has got the request and is trying to forward the request to the proper destination. Responses are three digit codes and there is reason phrase after the code. In this example the headers of response and request are the same. Reason of that, Alice's SoftPhone can achieve response request relationship. atlanta.com acquires biloxy.com's IP address and forwards request there. Before forwarding the request atlanta.com server adds an additional via header to the request, which contains its address. Likewise when biloxy.com receives the request, sends 100 Trying response to atlanta.com that states request is being processed by biloxy.com server. biloxy.com obtains current IP address of Bob by using some methods such as database look up, and sends the request to Bob's SoftPhone by adding one more additional via header to request, which is its own address. Then Bob's phone receives the request, which comes from Alice. At this point Bob's SoftPhone sends 180 Ringing response to biloxy.com and then Atlanta.com and finally to Alice's SoftPhone. At each point one via header field is removed from the response and obviously each point figures out where to send the response by using these via header fields.

When Bob answers the call, his SoftPhone sends a 200 OK response, which shows that the call was accepted. This response contains a body with a SDP. SDP data contains, which formats of media, is wanted by Bob's SoftPhone. The response, which is sent by Bob's SoftPhone, may look like this:

```
SIP/2.0 200 OK

Via: SIP/2.0/UDP server10.biloxi.com ; branch = z9hG4bKnashds8; r

received=192.0.2.3

Via: SIP/2.0/UDP bigbox3.site3.atlanta.com
```

```
;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2

Via: SIP/2.0/UDP pc33.atlanta.com

;branch=z9hG4bK776asdhds ;received=192.0.2.1

To: Bob <sip:bob@biloxi.com>;tag=a6c85cf

From: Alice <sip:alice@atlanta.com>;tag=1928301774

Call-ID: a84b4c76e66710@pc33.atlanta.com

CSeq: 314159 INVITE

Contact: <sip:bob@192.0.2.4>

Content-Type: application/sdp

Content-Length: 131
```

The first line contains response code and response phrase, which are “200” and “OK” respectively. The other lines are the header fields. These header field values are copied from INVITE request. There is tag attribute in the To header field of the response, which is sent by Bob’s SoftPhone. This tag value is used as an identifier of the dialog, which is established by the call between Alice’s and Bob’s SoftPhones. This tag value is used with every request and response within this call. Contact header field contains the URI, which Bob could be reached directly. The content type and content length header fields contains “application/sdp” and the length of the SDP data respectively.

Finally, Alice’s SoftPhone sends an ACK back to Bob’s SoftPhone. But at this point ACK doesn’t go through proxies because Alice’s SoftPhone has already learned the address of Bob’s SoftPhone. It sends ACK directly to Bob’s SoftPhone. From this point on, proxies are out of the picture, and Alice’s SoftPhone and Bob’s SoftPhone communicates without any delegation.

When the call ends, in other words one party hangs up the phone, BYE message is sent to the other party. At this point, when Bob hangs up the phone, BYE message is sent to Alices's SoftPhone. This BYE message is sent without proxies, too. Alice's SoftPhone sends a 200 OK response, which ends the call [1].

3. DESIGN AND IMPLEMENTATION OF THE SYSTEM

This chapter gives the details of our SIP phone implementation. The hardware and software components of the system and how they are integrated together are explained in a detailed manner. We start by detailing the hardware components: Embedded boards used in the design and how they are plugged in together. We then describe the software components of the system, i.e., the SIP stack and the SIP user agent, and how they are implemented and integrated together.

3.1 Hardware Components of the System

As embedded board of our VoIP Phone, we use Ubicom architecture, which is an evaluation board containing a IP5160 Ubicom processor. For display and user interaction purposes, we use an MCBSTR9 Evaluation Board, which has an ARM core processor manufactured by Keil Company.

3.1.1 Ubicom Architecture

IP5160 processor is a network processor having a multithreaded architecture for software I/O. This feature allows ten threads to run with no context switching overhead. It is a 32-bit CPU supporting ten-way multithreading operation. All tasks to be executed have real-time capability with completely deterministic time constraints. Block diagram of the processor is in figure 3.1.

The evaluation board, which has an IP5160 processor, has 4 Ethernet ports. This allows connectivity not only to the proxy and registrar but also to the user interface board. The evaluation board is shown in figure 3.2.

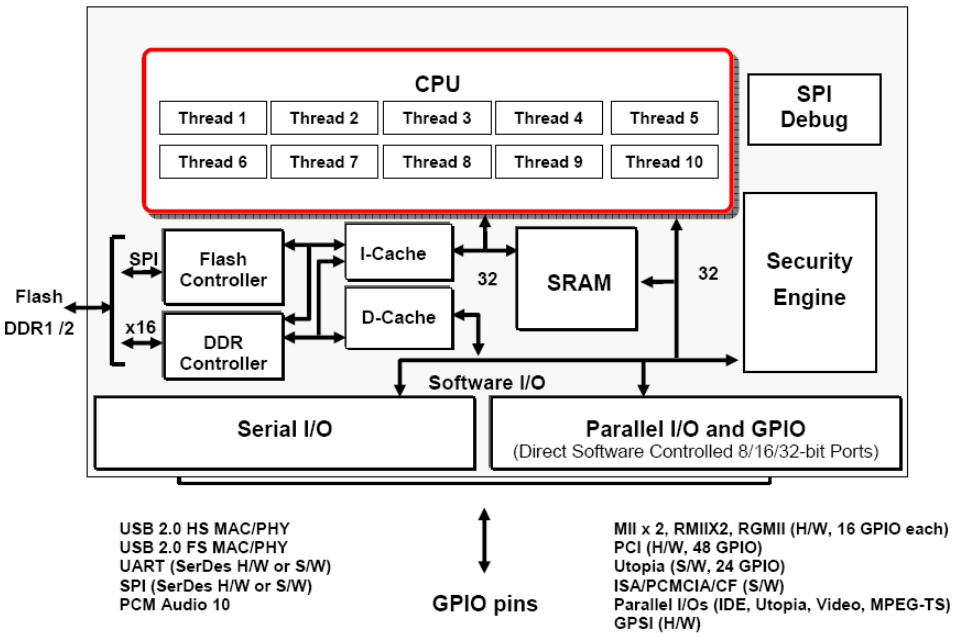

Figure 3.1 IP5160 Block Diagram

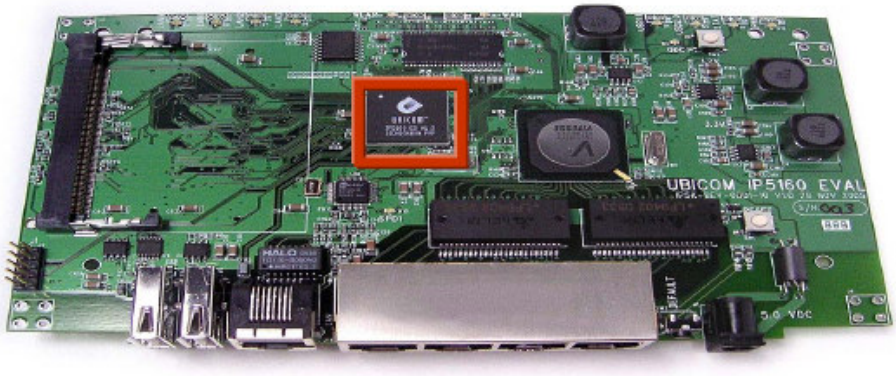

Figure 3.2 Ubicom Architecture Evaluation Board

3.1.2 ARM Architecture

STR912F microcontroller is used for this purpose. It has a RISC core, which is based on the Harvard architecture, and has a 5-stage pipeline and tightly-coupled

memories. Its most important feature is low power consumption [8]. Block diagram of the microcontroller is shown in figure 3.3.

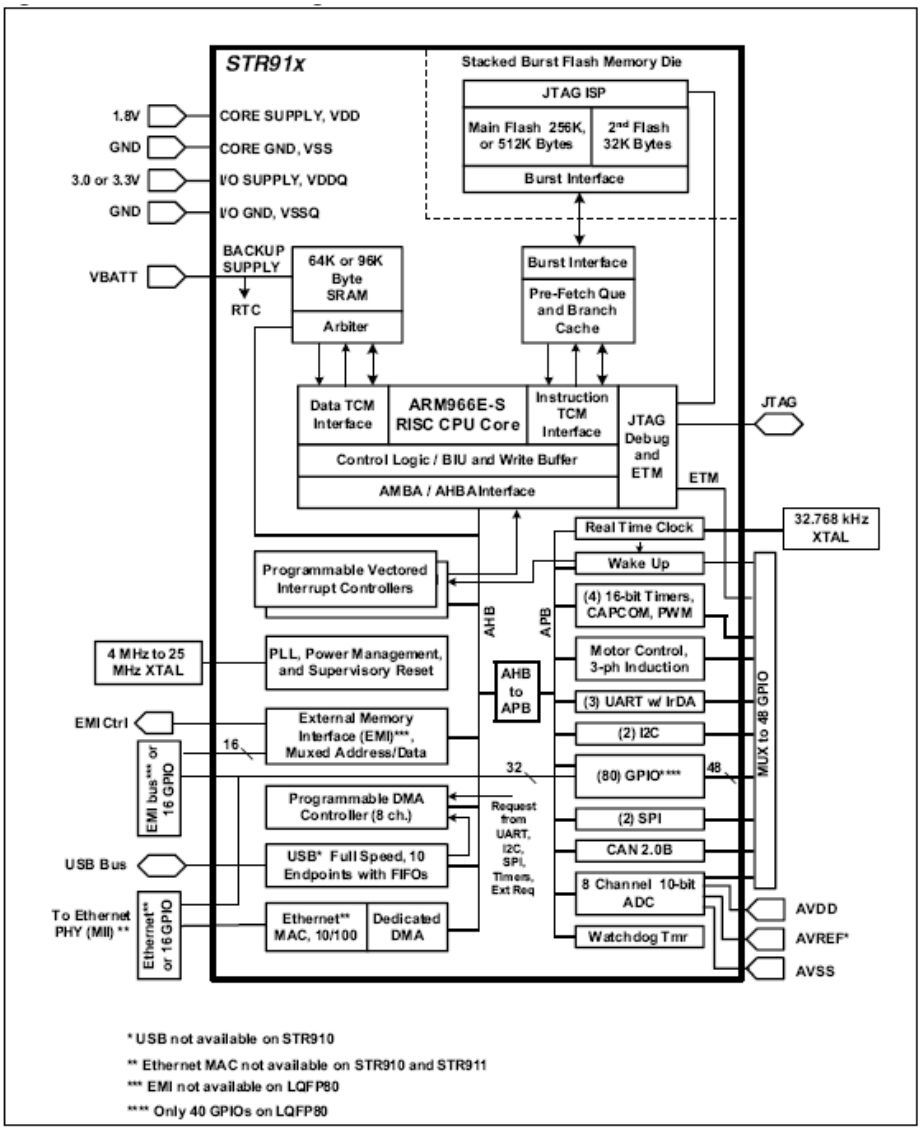

Figure 3.3 STR91 Block Diagram

MCBSTR9 evaluation board has a STR91 microcontroller, a LCD display and a three buttons, which is required for user interaction. The evaluation board is shown in figure 3.4.

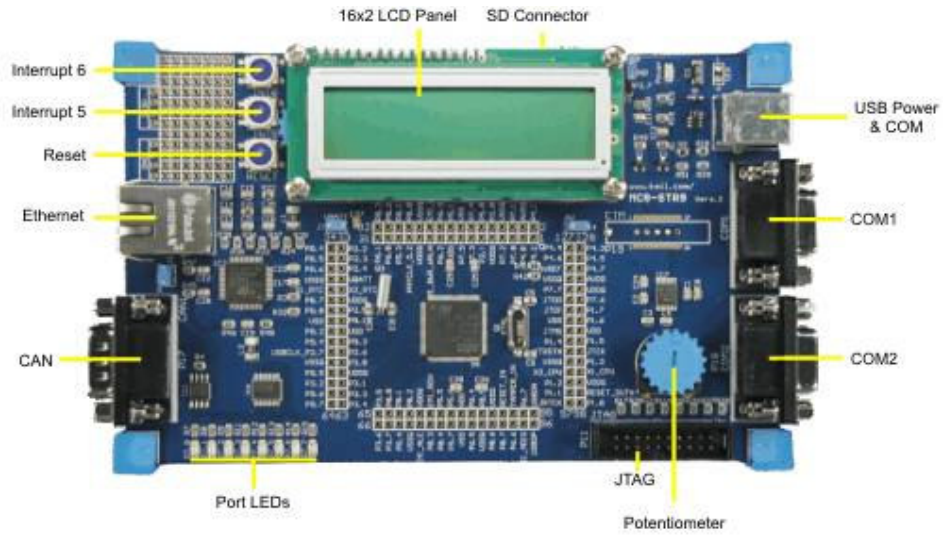

Figure 3.4 MCBSTR9 Evaluation Board

3.2 Software Components of the System

Software components of the system consist of a SIP User Agent implemented on top of our SIP stack, a SIP proxy and a Registrar. In this section we explain these components one at a time.

3.2.1 SIP Stack

SIP stack is at the heart of our implementation and is conformant to RFC3261. The stack is designed and implemented in a modular way, and has a layered architecture. Layering allows different parts of the stack to be independent of each other. In other words, layers are loosely coupled.

There are 5 layers including the application layer. The layers are shown in the figure 3.5.

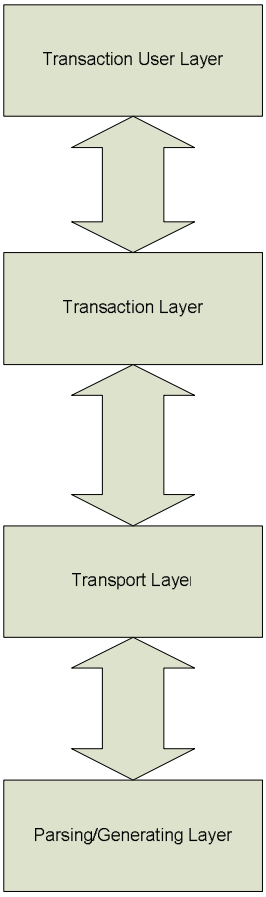

Figure 3.5 SIP Layered Architecture

Each layer and how they're implemented and by what functions are explained below from bottom to top manner.

Parsing/Generating Layer

This is the lowest layer, and is responsible for SIP message parsing and generation. This layer has 5 public interface functions available to the upper transport layer, and 22 private functions that implement the Parsing/Generating Layer itself. The structure of the Parsing/Generating Layer implementation is shown in figure 3.6.

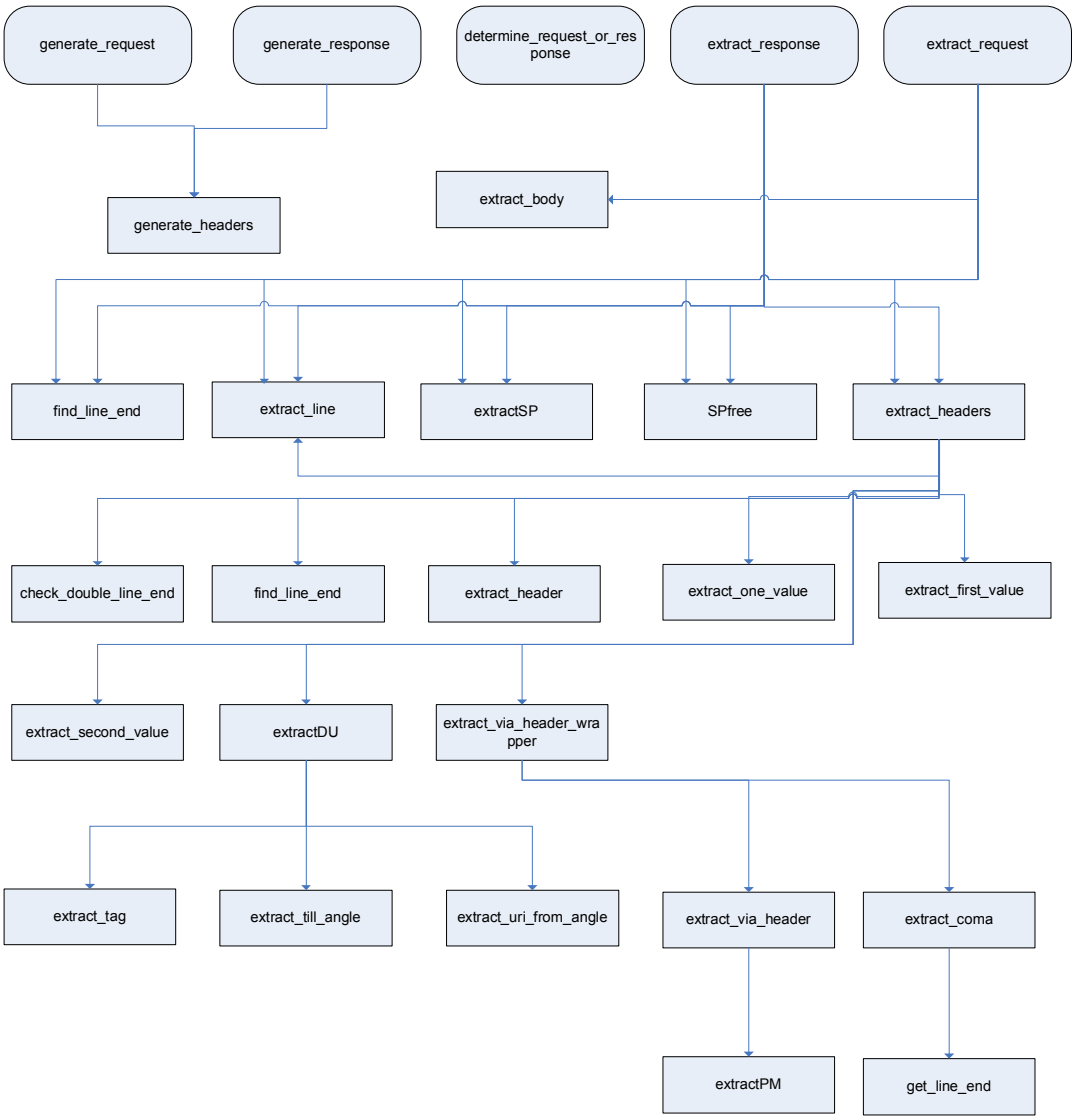

Figure 3.6 The structure of the Parsing/Generating Layer implementation

Functions of the Parsing/Generating Layer

Before explaining the functions of this layer, it is required to point out the structures used by these functions. Below we give the C structures used by the implementation of this layer:

```
struct request {
        request_type_t rt;
        char *request_uri;
        struct Headers headers;
        char *body;
};
struct Headers {
        char *call_id;
        u32_t content_length;
        struct CSeq *cseq;
        struct Contact *contact;
        struct From *from;
        u8_t max_forwards;
        struct Via *via;
        struct To *to;
        char *content_type;
        u32_t expires;
        char *record_route;
```

```
        char *event;
        char *accept;
};

struct CSeq {
        u32_t sequence;
        request_type_t rt;
};
struct Contact {
        char *display_name;
        char *uri;
};
struct From {
        char *display_name;
        char *uri;
        char *tag;
};
struct To {
        char *display_name;
        char *uri;
        char *tag;
```

```
};
struct Via {
        transport_type_t tt;
        char *domain;
        u16_t port;
        char *received;
        char *branch;
        struct Via *next;
};
struct response {
        u16_t status_code;
        char *reason_phrase;
        struct Headers headers;
};
```

Finally, structures used by the private functions of the layer are shown below:

```
struct SP {
        char *afterSP;
        char *extracted;
};
struct DU {
        char *display_name;
```

```
        char *uri;

        char *tag;

};

struct PM {

        char *parameter;

        char *value;

};
```

*struct netbuf *generate_request(struct request *req)*

Netbuf struct is a special kind of network buffer structure provided by Ubicom. "struct request" contains the request type, which can be REGISTER, MESSAGE, SUBSCIBE and NOTIFY. Since every request must have a request URI, this structure has a request_uri attribute, which is in a char array type. It also has a Headers structure. This structure has the minimum requirements of the headers and the attributes needed for specific methods. That's why it has call_id, record_route and accept attributes specified in a char array. It also has content_length, max_forwards and expires attributes specified in numerical values. Struct CSeq shows the sequence number and the request type; struct Contact has display name and uri of the contact; struct From has a display name, URI and a tag value; struct Via has transport type, domain, transport port, received IP and a branch parameter; and finally, struct To has a display name, URI and a tag value.

The job of this function is to take a request structure, and generate a serialized version of the structure ready for transmission. The serialized version is laid into the netbuf buffer, and this value is returned by the function.

*struct netbuf *generate_response(struct response *resp)*

struct response structure is very similar to struct request. The only difference is that it has a reason phrase and a status code, which shows the response code of the answer.

The job of this function is similar to generate_request: It is responsible for taking a response structure and generating a serialized version of the structure, which is ready for transmission. The serialized data is put into the netbuf buffer and this value is returned by the function.

*bool_t determine_request_or_response(struct netbuf *nb)*

This function takes a serialized version of a request or a response, and determines if the buffer contains a request or a response. If the buffer contains a request, the function returns true otherwise it returns false.

*struct response *extract_response(struct netbuf *nb)*

This function takes a serialized response message and extracts the response.

*struct request *extract_request(struct netbuf *nb)*

This function takes a serialized request and extracts the request.

Transport Layer

Transport layer's main duty is actual transmission of requests and responses over the network. If transport layer uses a connection oriented transport protocol such as TCP then it is responsible for persistence of these connections. It is recommended to keep the connections open at least for one transaction duration. It is obligatory to support both TCP and UDP in an implementation and they're both implemented in our stack.

For the client side of the transport layer, it is required to be able to send requests and receive responses. If a request is within 200 bytes of the path MTU, or if it is larger than 1300 bytes when the path MTU is not known, the request has to be

sent using congestion controlled transport protocol, such as TCP. In this thesis, TCP is used as the default transport protocol for transmission of SIP requests.

In general, responses to requests are sent over the same connection, that's why for connection oriented protocols, the response is received over the same connection. Only if an error occurs the response is sent over a new connection.

When a response is received by the transport layer, Via header field value is analyzed. If the value of this parameter is different than the one sent in the request, then the response is discarded.

The transport layer must also be prepared to receive requests at TCP or UDP ports 5060, which is what we use in this thesis [1].

The implementation of this layer has 2 public functions available to the upper Transaction Layer, and 10 private functions that implement that Transport Layer itself. The structure of the Transport Layer implementation is shown in figure 3.7.

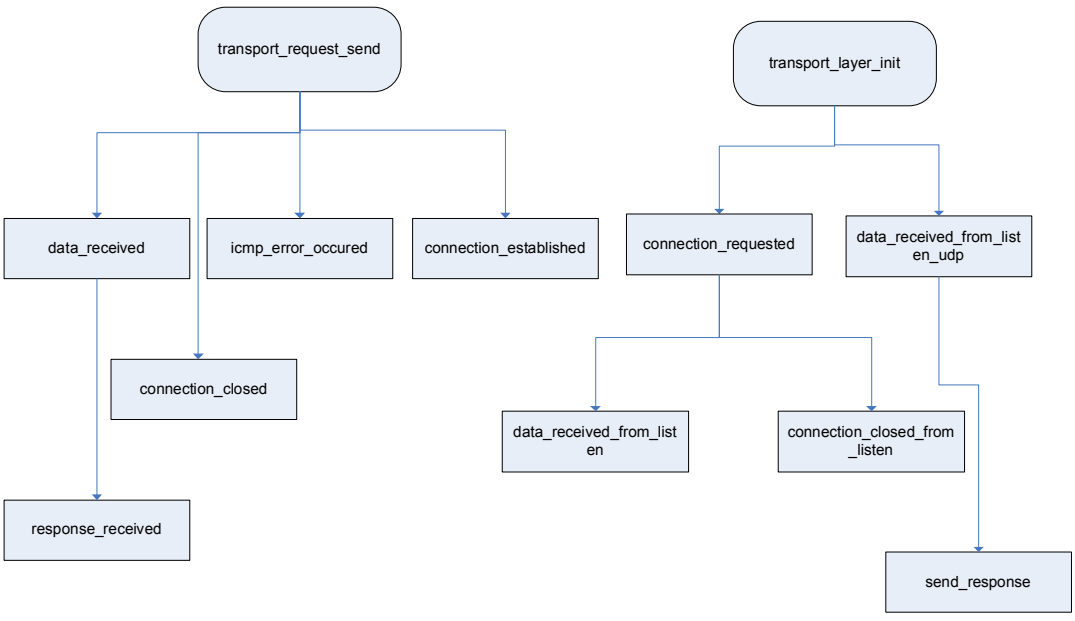

Figure 3.7 The structure of the Transport Layer implementation

Functions of the Transport Layer

transport_layer_init function must be called during initialization. This function initializes the connections and required buffers. Data receiver functions are designed as callbacks, which get invoked when data are received. When sending a request transport_request_send function is used. Before explaining this and the related functions, we need to show the structures used by the public and private functions of the Transport Layer.

```
struct tcp_client_transaction_instance {
        tcp_client_transaction_state_t tcts;
        struct oneshot *timerF;
        proceeding_callback_t pc;
        completed_terminated_callback_t ctc;
        struct response *resp;
        struct tcp_connection *tc;
        struct netbuf *nb;
        char *branch;
        struct CSeq *cseq;
        bool_t send_successful;
};
```

Although tcp_client_transaction_instance structure is mainly used by the transaction layer, it is also required by transport_request_send function.

Detailed explanation of the functions is as follows:

*void transport_request_send(struct request *req, struct cp_client_transaction_instance *tcti)*

This function firstly allocates a connection, and then uses the struct request parameter and the public function generate_request of the Parsing/Generating Layer to generate a serialized version of the request. This serialized data is then sent to the proxy or registrar. While sending the request, some callback functions are setup, which are required to know the state of the connection and make the proper action.

void transport_layer_init(void)

This function is responsible for allocating permanent sockets. Specifically, TCP and UDP ports 5060 are set up to listen to incoming messages, and the required callback functions are installed.

Transaction Layer

SIP is a transactional protocol with request and responses being part of the transactions. Transactions have a client side and a server side. The client side is called as a client transaction and the server side is called a server transaction. The client transaction sends the request, and the server transaction sends the response. Receiving responses is also the responsibility of the client transaction.

For each client transaction, a finite state machine defines the actions to be taken in response to sent/received messages. As an example, we show the finite state machine for the non-INVITE client transaction which is shown in figure 3.8.

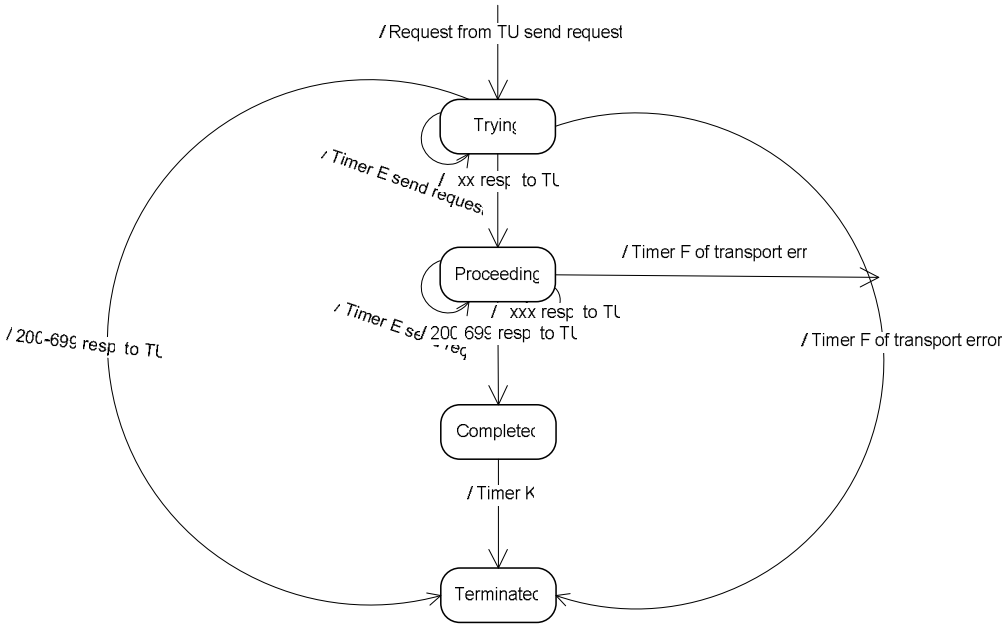

Figure 3.8 Client Transaction

The first state entered is Trying state. At this point a new client transaction is initiated. Timer F is set up to fire in 32 seconds. If that's the case then terminated state is entered. If an unreliable transport is used Timer is set up. If a provisional response is received while in the Trying state, then Proceeding state is entered. If a final response is received in Proceeding state, completed state is entered. This is also the case when final response is received during the Trying state. While in the Completed state, if an unreliable transport is used Timer K is set up to fire in 4 seconds. When this timer is fired, Terminated state is automatically entered. If a reliable transport is used, then the duration to fire for Timer K is zero seconds. In other words, for reliable transport when the state is completed, Terminated state is entered momentarily.

Server transaction has a similar state machine shown in figure 3.9.

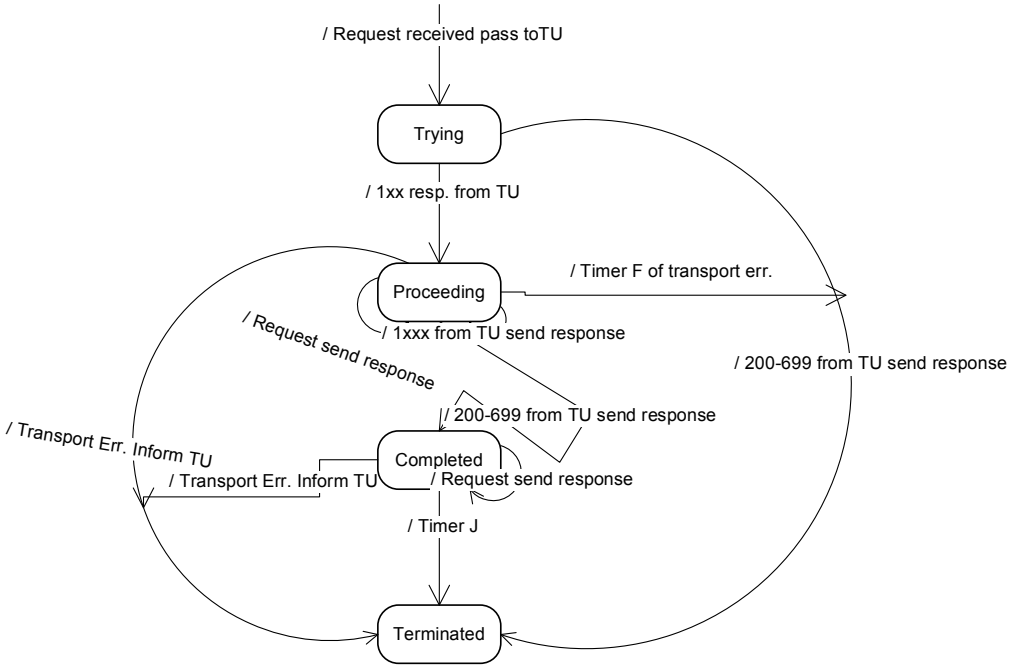

Figure 3.9 Server Transaction

The first state entered is Trying state. When 1xx response is sent from the server, Proceeding state is entered. While in this state if another 1xx response is sent, then this shows that it has to remain in this state. When a final response is sent then it is time to enter Completed state. While in the Proceeding or in the Completed state if a transport layer error occurs then state machine transits to Terminated state. When in the Completed state Timer J is set up to fire in 32 seconds for unreliable transports. If a reliable transport is used then Timer J's fire duration is zero second.

Another important aspect of transaction layer is matching responses to client transactions. There are two required conditions for a response to be owned by a client transaction. First one, the branch parameter value in the top Via header of the response must be the same with the branch parameter value in the top Via header of request, which is created for that transaction. The second one, the CSeq header value of the response must be the same as the CSeq header value of the matching request [1].

The implementation of this layer has 6 public functions available to upper layers, and 4 private functions that implement the Transaction Layer itself. The structure of the Transaction Layer implementation is shown in figure 3.10.

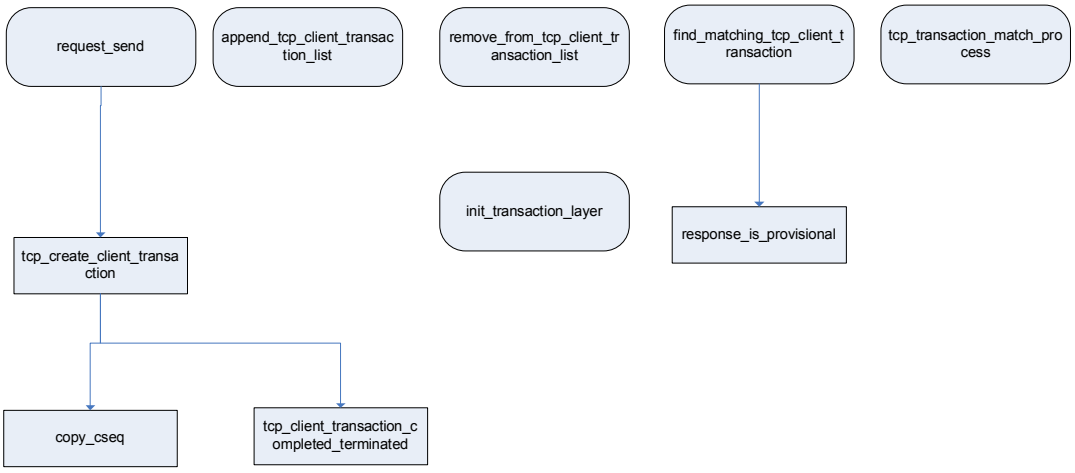

Figure 3.10 The structure of the Transaction Layer implementation

Functions of the Transaction Layer

The first function to be called for this layer is init_transaction_layer. After this initialization, other public functions can be called from upper or lower layers. Private functions on the other hand are only callable from within the Transaction layer. Before explaining the functions of this layer, we need to show structures defined by this layer:

```
struct tcp_client_transaction_list {

        struct tcp_client_transaction_instance *tcti;

        struct tcp_client_transaction_list *next;

}
```

void init_transaction_layer(void)

This function simply calls transport_layer_init function, which is required to not only itself but also to initialize the lower layer.

*bool_t request_send(struct request *req, transport_type_t tt, proceeding_callback_t pc, completed_terminated_callback_t ctc)*

This function takes transport type and depending on the value TCP or UDP transaction is initiated. Req parameter is the request for that transaction. pc and ctc are functions to be called when the transaction transits to proceeding and terminated states, respectively.

*bool_t append_tcp_client_transaction_list (struct tcp_client_transaction_instance *tcti)*

This function takes tcp_client_transaction_instance as an argument and adds this instance to transaction list. If successfully added, then true is returned otherwise false is returned.

*bool_t remove_from_tcp_client_transaction_list(struct tcp_client_transaction_instance *tcti)*

This function takes the transaction instance, finds it in the list and removes it. If it can't be found then false is returned, otherwise true is returned.

*struct tcp_client_transaction_instance *find_matching_tcp_client_transaction(struct response *resp)*

This function tries to find a transaction, which is matched with the response given as a parameter. If a transaction instance is found, than this value is returned, otherwise null is returned.

*void tcp_transaction_match_process(struct tcp_client_transaction_instance *found_tcti, struct response *resp)*

This function takes transaction instance and response as parameters and makes the required state changes for that transaction.

Transaction User Layer

The layer above the Transaction Layer is Transaction User Layer. This layer is responsible for the creation of a transaction based on the method being used, and for sending the transaction to the Transaction Layer. The methods supported by our implementation in this thesis are REGISTER, MESSAGE, SUBSCRIBE and NOTIFY.

This layer consists of 6 public functions, and 14 private functions that implement the Transaction User Layer itself. The structure of the Transaction User Layer implementation is shown in figure 3.11 and in figure 3.12.

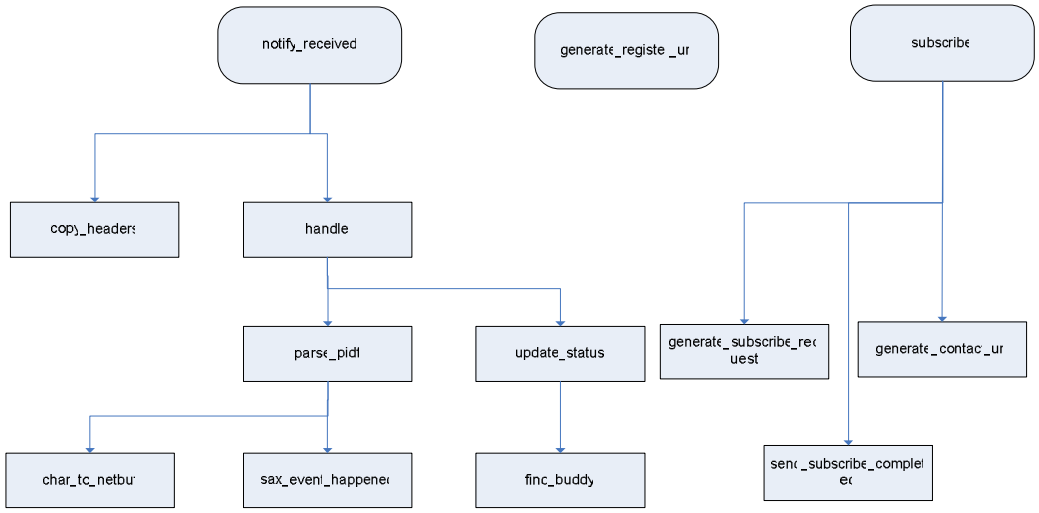

Figure 3.11 The structure of the Transaction User Layer implementation-1

Functions of the Transaction User Layer

The first function to be called in this layer is init_TU_layer, which initializes the Transaction User Layer. The public functions of the layer are explained below:

*struct response *notify_received(struct request *req)*

With the notify method, user information is received. This information is about the status of the users, specifically, if the users are online or offline. This

function gets the user and user information, and updates the buddy list whose status have been subscribed before. The request body has a pidf format which is in xml. That's why some simple functions defined as private are used to parse the pidf data.

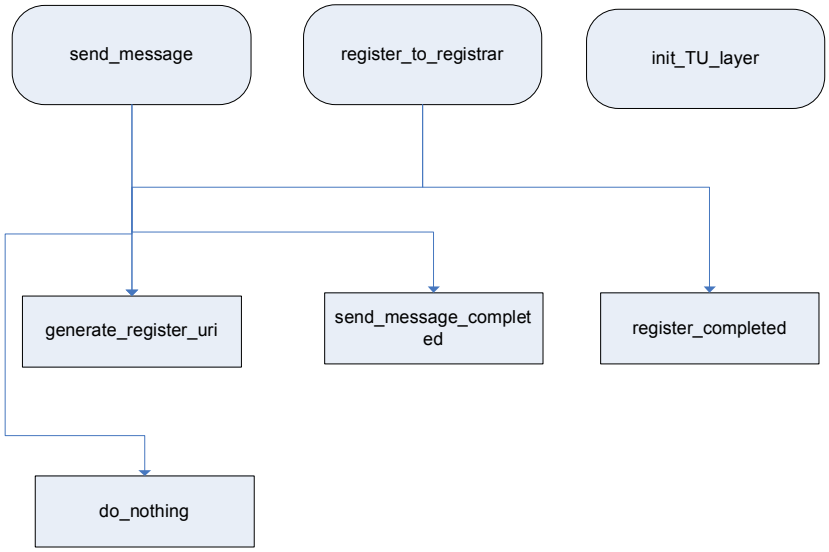

Figure 3.12 The structure of the Transaction User Layer implementation-2

*char *generate_register_uri(void)*

This function generates the SIP URI used for registration with the Registrar.

bool_t subscribe(void)

This function generates subscribe requests for all buddies, and sends these requests to the presence server. If all subscribe requests complete successfully, this function returns true otherwise returns false.

*void send_message(TU_completed_callback_t comp, char *uri, char *message)*

This function takes three parameters. The first one is a callback function to be called when the transaction ends. The second one is the SIP URI of the receiver of the message. And the last one is the message that will be sent to the SIP URI, which is passed as the second parameter.

void register_to_registrar(TU_completed_callback_t comp)

This function's job is to send a Register request to the proxy. The only parameter is a callback function to be called at the end of the transaction. TU_completed_callback_t type is function pointer which takes only one parameter, a Response structure. When the transaction ends, the response is received by calling this function.

*void init_TU_layer(ipv4_addr_t i_outbound_proxy_ip, u16_t i_outbound_proxy_port, u16_t i_tcp_listen_port, u16_t i_udp_listen_port, char *i_user_display_name, char *i_user_name, char *i_register_request_domain, TU_message_received_callback_t i_message_received)*

This function is to be called first before any other function is invoked. The parameters of the functions are i_outbound_proxy_ip, which is the proxy server's IP, i_outbound_proxy_port, which is the proxy server's port, i_tcp_listen_port is the TCP port to be listened, i_udp_listen_port is the UDP port to be listened, i_user_display_name is the display name, i_user_name is the username of the SIP URI, i_register_request_domain is the domain of the SIP URI, i_message_received is the callback function to be invoked when a messaged is received.

3.2.2 Registrar, Proxy and Presence Server

Registrar, Proxy and Presence Server are implemented using jiplets of Cafesip, an open source project. Jiplet stands for Java SIP Servlet. The servers are written in Java using Jiplet API and deployed in the Jiplet container. The container is an open source container for SIP server applications. Support for SIP message parsing and formatting, scoped variables, authentication and authorization, thread-pooling, logging, custom class loading, management interface are provided by the container. So it is possible to create server-side SIP applications such as Registrar, Proxy or Presence Server easily. The Jiplet container can be run in two modes. One of them is standalone mode, the other one is in a J2EE server as a service. The jiplet container provides Java classes that jiplet applications extend or use [9].

The most important part about deployment is “jip.xml” descriptor file. That file defines the jiplets and configurations. It is possible to use “context-mapping” element to define which requests are processed by the container. In this implementation three variables are used for this “context mapping”. These are (1) URI host, (2) To URI host, and (3) From URI host. All of these are in “anadolu.edu.tr” domain. The code snippet regarding this part is shown below:

```
<context-mapping connector="sip-connector">
 <mapping>
   <or>
     <subdomain-of>
       <var>request.uri.host</var>
       <value>anadolu.edu.tr</value>
     </subdomain-of>
     <subdomain-of>
       <var>request.to.host</var>
       <value>anadolu.edu.tr</value>
     </subdomain-of>
     <subdomain-of>
       <var>request.from.host</var>
       <value>anadolu.edu.tr</value>
     </subdomain-of>
   </or>
 </mapping>
```

```
        </context-mapping>
```

Three jiplets are written for this thesis: (1) SIPRegistrarJiplet, which is responsible for registration requests and responses, (2) SIPProxyJiplet, which proxies the requests and response, (3) PresenceServiceJiplet, which handles Subscribe and Notify requests and responses. The code snippet of "jip.xml" regarding this part is shown below:

```
    <jiplet>
      <jiplet-name>SIPRegistrarJiplet</jiplet-name>
      <jiplet-class>org.anadolu.jiplet.SipRegistrar</jiplet-class>
      <jiplet-connector>sip-connector</jiplet-connector>
      <startup-order>0</startup-order>
    </jiplet>
    <jiplet>
      <jiplet-name>SIPProxyJiplet</jiplet-name>
      <jiplet-class>org.anadolu.jiplet.sip.ProxyJiplet</jiplet-class>
      <startup-order>1</startup-order>
      </init-params>
    </jiplet>
    <jiplet>
      <jiplet-name>PresenceServiceJiplet</jiplet-name>
      <jiplet-class>org.anadolu.reference.jiplet.SipPresence</jiplet-class>
      <jiplet-connector>sip-connector</jiplet-connector>
```

```
    <startup-order>2</startup-order>
  </jiplet>
```

Finally, method mappings of the jiplets are accomplished. These mappings show which methods will be handled by which jiplets. The code snippet regarding this part is shown below:

```
  <jiplet-mapping jiplet="SIPRegistrarJiplet">
    <mapping>
      <equals ignore-case="true">
        <var>request.method</var>
        <value>REGISTER</value>
      </equals>
    </mapping>
  </jiplet-mapping>
    <jiplet-mapping jiplet="SIPProxyJiplet">
    <mapping>
                <and>
     <not>
      <equals ignore-case="true">
       <var>request.method</var>
       <value>REGISTER</value>
      </equals>
     </not>
```

```
                        <not>
                                <equals ignore-case="true">
                                        <var>request.method</var>
                                        <value>SUBSCRIBE</value>
                                </equals>
                        </not>
                </and>
  </mapping>
</jiplet-mapping>
        <jiplet-mapping jiplet="PresenceServiceJiplet">
  <mapping>
                <or>
     <equals ignore-case="true">
       <var>request.method</var>
       <value>SUBSCRIBE</value>
     </equals>
  </or>
  </mapping>
</jiplet-mapping>
```

3.2.3 User Interface

To have users interact with our VoIP phone, an auxiliary MCBSTR9 Evaluation board is used. This board has a 2x16 character LCD and an ARM core processor manufactured by Keil Company. Connectivity between MCBSTR9 and Ubicom architecture is achieved by TCP/IP. The physical link between these two architectures is an Ethernet cable.

The software written for MCBSTR9 is a server, which uses TCP port 1999. The software related to User Interface of Ubicom architecture is a client, which also uses TCP port 1999. The server listens to connections at TCP port 1999, and the client connects to the server from TCP port 1999.

The server side is simpler than the client side, because the server side only takes the character stream and displays it to the LCD. The client side on the other hand client, prepares message, cuts it if necessary so that it can be displayed on a 2x16 character display, and sends the stream to the server for display.

4. INTEGRATION AND TESTING OF THE SYSTEM

This chapter explains how the integration and testing of the system is done. For some of the testing purposes, X-Lite SoftPhone from Counterpath Corporation [10] is used as an instant messenger.

4.1 Integration

The Proxy, Registrar and Presence server reside on a PC. Another PC runs the X-Lite SoftPhone and works as an end-system. The other end-system is our VoIP Phone realized by the Ubicom board. An ARM board is connected to the Ubicom board to display incoming messages. This architecture of the system is illustrated in Figure 4.1. Our VoIP phone can send/receive messages, subscribe for other users' presence information and receive notifications from the Presence server.

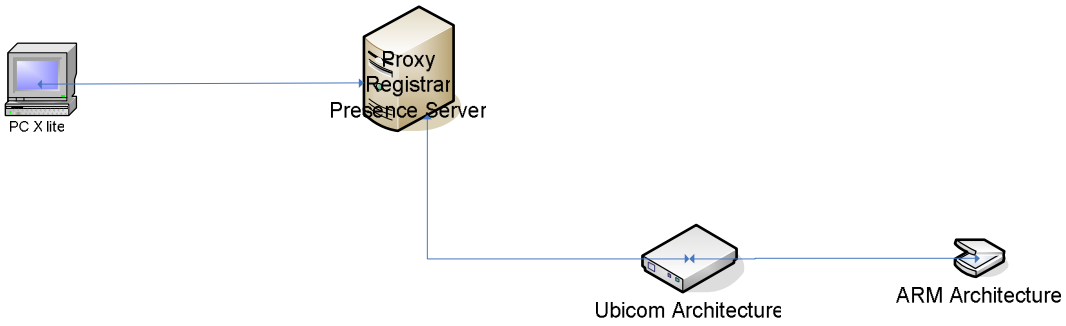

Figure 4.1 Integration of the System

Upon startup, both X-Lite and our VoIP phone register with the Registrar. When a message is sent from either X-Lite to Ubicom or vice versa, the message is taken by the proxy and forwarded to the registered end-point. Both X-Lite and Ubicom subscribes for each other's presence status with the Presence server. Thus, when one of them is online or offline, the presence information is sent to the other one by the Presence server and displayed on the screen.

4.2 Testing

For the testing of the system, a simple firmware was implemented for our VoIP phone. The figure 4.2 represents the general structure of the firmware deployed inside the Ubicom board.

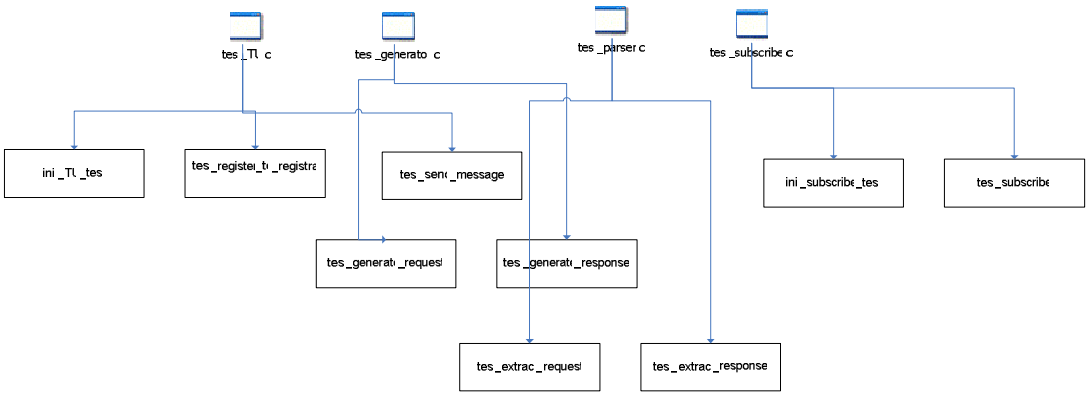

Figure 4.2 Testing Software of Ubicom Architecture

Let's explain each of these test functions:

void init_TU_test(void)

This function configures outbound proxy IP address and initializes Transaction User Layer.

void test_register_to_registrar(void)

This function generates a register request according to the information entered in *init_TU_test* function and registers with the Registrar.

void test_send_message(void)

This function sends a sample message to sip:ali@anadolu.edu.tr.

void test_generate_request(void)

This function generates a test request and prints it to the console. In fact, there is a private function used by this function. This function constructs a request structure

and returns it. Then test function calls *generate_request* of Parsing/Generating Layer, and returned stream is printed and checked visually.

void test_generate_response(void)

This function generates a test response and prints it to the console. There is a private function used by this function that constructs a request structure and returns it. The test function then calls *generate_response* of Parsing/Generating Layer, and returned stream is printed and checked visually.

void test_extract_request(void)

This function uses a private function to generate a request stream. The generated stream is taken by *extract_request* function of Parsing/Generating Layer as a parameter. The returned request structure's elements are compared with those in the request stream. For every successful comparison, a message is printed on the console. If a comparison fails, an error message is printed.

void test_extract_response(void)

This function uses a private function to generate a response stream. The generated stream is taken by *extract_response* function of Parsing/Generating Layer as a parameter. The returned response structure's elements are compared with those in the request stream. For every successful comparison, a message is printed on the console. If a comparison fails, an error message is printed.

void init_subscribe_test(void)

This function constructs a buddy list whose status will be subscribed for, and initializes the Transaction User Layer.

void test_subscribe(void)

This function calls *subscribe* function of Transaction User Layer and subscribes for the status of each person in buddy list.

4.3 X-Lite Demonstration of the System

X-Lite is a commercial SoftPhone by Counterpath Coportation [10]. Figure 4.3 below shows a snapshot of X-Lite when it starts running. In this figure, user "Ali" has logged in from X-Lite and registered himself with the Proxy server. Since the Registration is complete, X-Lite displays the user's name on its screen pad.

Figure 4.3 X-Lite Software, Ali is registered

Although not demonstrated, user Ayse logs in from our VoIP Phone and also registers herself with the Proxy server. The next step for Ali and Ayse is to subscribe for each other's presence status. This is done by sending SUBSCRIBE requests to the Presence server. The Presence server then notifies each VoIP Phone by sending NOTIFY requests. Figure 4.4 shows a snapshot of the X-Lite user interface indicating that Ayşe is online.

Figure 4.4 X-Lite Software, Ayşe is online

Now, Ali and Ayse exchange messages using the SIP MESSAGE request through the Proxy. Messages coming from Ayse are displayed on X-Lite's instant messaging panel as depicted in Figure 4.5. Messages coming from Ali to Ayse at our VoIP phone will be displayed on the 2x16 LCD displayed connected to the MCBSTR9 board.

These exchanges show that our SIP Phone runs a standards-compliant SIP stack and can interoperate with commercial SIP Proxies and SIP phones.

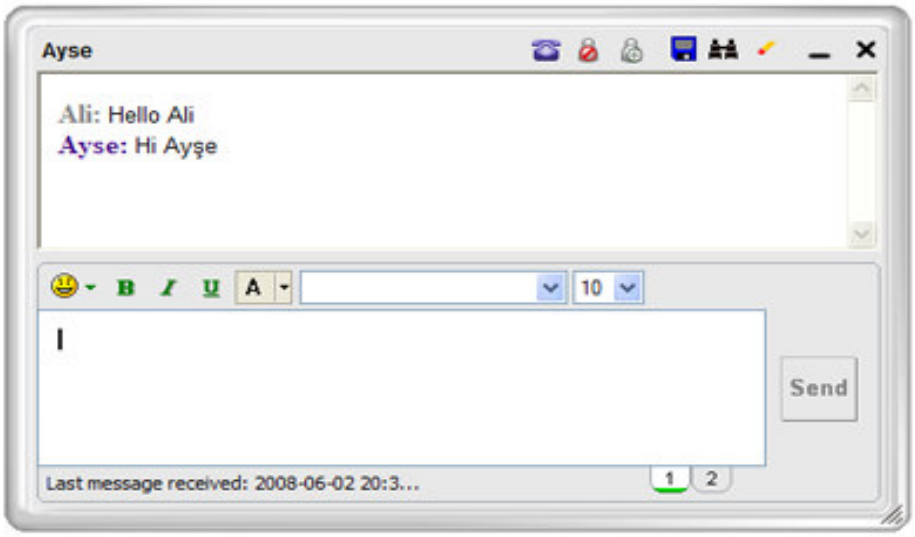

Figure 4.5 Ali and Ayşe sent messages to each other

5. CONCLUSION

In this thesis, a SIP stack has been implemented for Ubicom architecture. This stack structure is built in a layered manner. Instant messaging and presence awareness are written using the implemented SIP stack. In order to support the application, a proxy server, a registrar server and a presence server is written for PC architecture.

There are four layers in the stack. These layers are Transaction User Layer, Transaction Layer, Transport Layer and Parsing/Generating Layer. These layers are introduced in SIP RFC [1]. Transaction User Layer is responsible for the creation of a transaction based on the method being used, and for sending the transaction to the Transaction Layer. Transaction Layer is responsible for handling client and server transactions and state machines that are defined in SIP RFC [1]. Transport Layer is responsible for actual transmission of requests and responses over the network. Parsing/Generating Layer is responsible SIP message parsing and generation.

SIP proxy server, presence server and registrar server are written using an open source project called Cafesip [9]. The registrar server handles the registration process of the application written upon on the stack and the X-lite softphone. The presence server sends the presence information to the subscribed users. And proxy server proxies the messages between two users.

REFERENCES

[1] Rosenberg J., Schulzrinne H., Camarillo G., Johnston A., Peterson J., Parks R., Handley M., and Schooler E., *Sip: Session initiation protocol*, RFC 3261, 2002.

[2] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P. and Berners-Lee T., *Hypertext Transfer Protocol -- HTTP/1.1*, RFC 2616 1999.

[3] Postel J. B., *Simple Mail Transfer Protocol*, RFC 821, 1982.

[4] Gonzalo, C., *SIP Demystified*, McGraw-Hill, 2002.

[5] Sinnreich H., Johnston A.B., *Internet Communications Using SIP*, Wiley Publishing, 2006.

[6] Kuthan J. J. J. and Lancu B., *Sip Express Router V0.8.8-Developer's Guide*, 2002.

[7] Wikipedia Foundation, Inc, *Session Initiation Protocol,* 2007. http://en.wikipedia.org/wiki/Session_Initiation_Protocol

[8] STMicroelectronics, *UM0216 Reference Manual STR91xF ARM9-Based Microcontroller Family*, 2006.

[9] Cafesip, *Cafesip Project,* 2007. http://www.cafesip.org

[10] Counterpath Corparation*, X-Lite SoftPhone*, 2007. http://www.counterpath.com

[11] Low C., *The Internet Telephony Red Herring,* Global Telecommunications Conference, 72-80, 1996.

[12] 3cx Corparation, *Sip phones / VOIP Phones types*, 2007. http://www.3cx.com/PBX/VOIP-phones.html

[13] Johnston A. B., *Sip: Understanding the Session Initiation Protocol,* Artech House, 2001.

[14] *Whatis.com*, 2007. http://www.whatis.com

[15] Dang L., Jennings C. and Kelly D., *Practical VoIP*, O'reilly, 2002.

[16] Davidson J., Peters J., Gracely B., *Voice Over IP Fundamentals,* Cisco Press, 2000.

[17] Smith C., Collins D., *3G Wireless Networks*, McGraw-Hill, 2002.

[18] Leon-Garcia A., Widjaja I., *Communication Networks: Fundamental Concepts and Key Architectures,* McGraw-Hill, 2003.

[19] Campbell B., Rosenberg J., Schulzrinne H., Huitema C., Gurle D., *Session Initiation Protocol (SIP) Extension for Instant Messaging,* RFC 3428, 2002.

[20] Rosenberg J., *A Presence Event Package for the Session Initiation Protocol (SIP),* RFC 3856, 2004.

[21] Sugano H., Fujimoto S., Klyne G., Bateman A., Carr W., Peterson J., *Presence Information Data Format (PIDF)*, RFC 3863, 2004.