Design of VoIP Paralleled Client-Server Software for Multicore

Master Thesis in Electronics and Communication System

ZEESHAN KHAN

Master’s Degree Project
Stockholm, Sweden 2011

TRITA-ICT-EX-2013:214
About

Author
Zeeshan Khan (Student MS Internetworking, Royal Institute of Technology)

Title
Design of VoIP Paralleled Client-Server Software for Multicore

Presentation Date
August 15th, 2011

Examiner
Ahmed Hemani (Professor) Royal Institute of Technology (KTH)
Abstract

As “Voice over IP” has become more prevalent and many client and server applications have been designed for them, the VoIP industry has seen the need for faster, more capable systems to keep up. Traditionally, system speed-up has been achieved by increasing clock speeds but, conventional single-core CPU clock rates have peaked a few years ago due to very high power consumption and heating problems. Recently, system speed-up has been achieved by adding multiple processing cores to the same processor chip called multi-core processors. The existing VoIP applications cannot attain full benefit and efficiency of multi-core processors because of their sequential design. “VoIP paralleled client-server software for multicores” that can split up sequential code and run concurrently on multiple cores instead of trying to exploit single-core hardware is the solution.

We have created a model of generic, open source paralleled VoIP-server (IOpenVoIP) in C that suits multi-core and that can be used as a simulation tool. Furthermore, we have designed and implemented a tool for performance testing. It can be used for performance evaluation of IOpenVoIP and other SIP servers. The tool emulates thousands of communication sessions through a server. Performance testing can help developers to eliminate bottle necks in multi-core server design.

On the other hand side, VoIP clients are not just used for voice and video communication over Internet. Along with audio and video they can carry other real time data i.e. patients ECG signals. Raw data is usually sent from one end and it is processed at other end which is a processor intensive task. We designed and implemented a graphical VoIP-Client which utilizes multi-core processors.
This thesis is part of the Master of Internetworking degree. It is done at department of Electronics System (ES), Royal Institute of Technology (KTH) and Solidux Telecom AB.

I would like to thank my supervisors, Professor Ahmed Hemani, and Dr. Iyad al Khatib for all their guidance, assistance, and valuable feedback throughout the period of this thesis. Additionally, I would like to thank Professor Gerald Q. Maguire Jr, Professor Axel Jantsch, and Muhammad Badawi (PHD Student at KTH) for their supervision. Very special thanks to my program coordinator May-Britt Eklund-Larsson, for her helpful attitude and advices.

Finally, thanks to my parents, my siblings and my friend and colleague Nasir Saeed for providing me moral support during my studies.

Stockholm, August 15th, 2011
Zeeshan Khan
List of Figures

1.1 Software development model ........................................ 3
2.1 Example of SIP Conversation ........................................ 10
2.2 Multi-core Chip Architecture ...................................... 11
2.3 Parallel Computing ................................................... 13
3.1 IOpenVoIP Server and Client ...................................... 18
3.2 IOpenVoIP Server Design .......................................... 21
3.3 IOpenVoIP Client Design .......................................... 23
3.4 IOpenVoIP Call Setup Time ...................................... 25
3.5 Multi-core server design .......................................... 26
4.1 Call scenario with time stamps .................................... 30
4.2 Traffic generator ...................................................... 31
4.3 Detailed testing strategy .......................................... 32
4.4 Test details ........................................................... 33
4.5 Packet loss ........................................................... 34
4.6 TC2 - TC1 ............................................................... 35
4.7 TC3 - TC2 ............................................................... 35
4.8 TC4 - TC3 ............................................................... 36
4.9 TS2 - TS1 ............................................................... 36
4.10 TS3 - TS2 .............................................................. 37
4.11 TS4 - TS3 .............................................................. 37
5.1 GUI with Qt Creator .................................................. 39
5.2 Signal and Slot mechanism ........................................ 40
5.3 AddContact form interaction with Main window ................ 41
5.4 GUI design of FeeaVoIP .............................................. 42
5.5 IPC with QtDBus ....................................................... 43
5.6 Detailed Design of FeeaVoIP ................................. 44
B.1 Splash screen .................................................. 53
B.2 Main window - Contacts TAB ............................... 54
B.3 Main window - Direct call TAB ............................. 54
B.4 Main window - Applications TAB ........................... 55
B.5 Setting Window ............................................... 55
B.6 Log Window .................................................. 56
B.7 Add Contact form ............................................ 56
C.1 Server User Interface ......................................... 57
C.2 User Registration ............................................ 58
C.3 User Login .................................................. 58
C.4 User Search ................................................ 59
C.5 Call setup - Text Message ................................ 59
Contents

Author i

Abstract ii

Acknowledgements iii

List of Figures iv

1 Introduction 1
  1.1 Problem statement ............................... 1
  1.2 Contribution ................................... 2
  1.3 Development Model .............................. 3
  1.4 Research work Activities ....................... 4
  1.5 Structure of the Report ......................... 5

2 Background 6
  2.1 Session Initiation Protocol (SIP) ............... 6
    2.1.1 Components of SIP .......................... 7
      2.1.1.1 User Agents ............................ 7
      2.1.1.2 Servers ............................... 7
      2.1.1.3 Proxy Server ......................... 8
      2.1.1.4 Redirect Server ....................... 8
      2.1.1.5 Registrar Server ....................... 8
    2.1.2 SIP Messages .............................. 8
      2.1.2.1 SIP Requests ......................... 9
      2.1.2.2 SIP Responses ......................... 9
    2.1.3 SIP Headers ................................ 10
    2.1.4 SIP Conservation ......................... 10
  2.2 Multi-core ..................................... 11
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1</td>
<td>Programming for Multi-core</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Parallelization</td>
<td>12</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Threads</td>
<td>13</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Processes</td>
<td>14</td>
</tr>
<tr>
<td>2.2.5</td>
<td>IPC</td>
<td>14</td>
</tr>
<tr>
<td>2.2.6</td>
<td>FIFOs</td>
<td>14</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Message Queues</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>OpenSIPS - Open Source SIP Server</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Qt</td>
<td>15</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Qt Creator IDE</td>
<td>16</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Qt Meta-Object System</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>VoIP Server and Client model</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>IOpenVoIP</td>
<td>18</td>
</tr>
<tr>
<td>3.2.1</td>
<td>IOpenVoIP Server Features</td>
<td>19</td>
</tr>
<tr>
<td>3.2.2</td>
<td>IOpenVoIP Client Features</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Server Design</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Client Design</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>Tests and Results</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Proposed Multi-core Server Design</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>OpenVoIP ServerAnalyzer</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Requirement specification</td>
<td>29</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Required Measurements</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Call Scenario</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Design and Implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Traffic generator</td>
<td>30</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Traffic analyzer and result generator</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Topology and experiments</td>
<td>32</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Tests</td>
<td>33</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Results</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>Multi-core graphical VoIP Client</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Basic design of FeeaVoIP</td>
<td>39</td>
</tr>
</tbody>
</table>
CONTENTS

5.3 Cross platform graphical user interface .......................... 39
5.4 Objects interaction via signals and slots .......................... 40
  5.4.1 FeeaVoIP GUI Interaction .................................. 40
5.5 GUI Design of FeeaVoIP ........................................ 42
5.6 FeeaVoIP scalability on Multi-core ............................... 43
5.7 Complete Design .................................................. 44

6 Conclusion ......................................................... 46
  6.1 Summary ......................................................... 46
  6.2 Future Work ..................................................... 47

7 Bibliography ....................................................... 48

A SIP Miscellaneous .................................................. 51

B FeeaVOIP User Manual ............................................. 53
  B.1 Splash Screen .................................................. 53
  B.2 Main window ................................................... 54
  B.3 Other forms ..................................................... 55

C IOpenVOIP Users .................................................. 57
  C.1 Server User Interface .......................................... 57
  C.2 Client .......................................................... 58
Chapter 1

Introduction

This chapter briefly introduces the reader to the thesis work. Section 1.1 describes the problem that our thesis is going to address. Section 1.2 explains our unique contribution to solution of the problem. Section 1.3 presents the development model we have adopted. Section 1.4 describes the research work activities. Finally, section 1.5 presents the organization of the thesis.

1.1 Problem statement

Design and development of Multi-core VoIP client and server model that suites multi-core and that can be used as a simulation tool.

According to ABI research[1] VoIP market has huge potential, it is set to double over the next few years and expected to reach $20 billion by 2015. Today, VoIP servers can process small number of concurrent users, which is not enough to cope with changing requirements. Until now, buying new hardware with higher CPU clock rate was a solution to such a problem. However, the era of single core processors is over due to clock speed limitations; the era of multi-core processors is here. Multi-core processors offer higher performance, scalability, and energy efficiency in comparison to single-core processors.

Similarly, modern VoIP clients support video and data communication in parallel with audio and text. Encoding a video with display resolution of 1,280 x 720 at 45 frames per second requires 41 million pixels to be processed each second. A tremendous amount of processing is needed for such operations. Multi-core processors are
the only way to meet such processing requirements by distributing the processing load across multiple CPUs.

For these obvious reasons, VoIP server and client software which utilizes multi-core processors is needed. The work done in the thesis targets to the community of researchers, developers, and students who want to explore, improve and learn about multi-core VoIP server and client softwares.

1.2 Contribution

The goal of this thesis is to design and implement a VoIP server and client model for multi-core system. A server model named IOpenVoIP has been developed in C during this thesis work which is a fully functional VoIP server. Our contribution lies in the design and development of server model as well as implementation of session establishment protocol.

We developed a server analyzer tool and named it OVSA (Open VoIP server analyzer). According to analysis performed by OVSA the first prototype of server model (IOpenVoIP) is subjected to 275 % performance degradation at load of 1000 sessions. Based on these tests we proposed a design of server that can utilize multi-core to eliminate performance degradation.

Our server analyzer tool can analyze existing SIP servers. We performed tests on famous OpenSIPS server which are presented later in this thesis. We have also designed and developed a graphical multicore VoIP client in Qt and C++.
1.3 Development Model

Since, the design of a multi-core server and client is a complicated and expensive project, we adopted spiral software development model.

![Diagram of spiral software development model](image)

**Figure 1.1: Software development model**

It allows incremental releases and step by step refinements of software. After each cycle, the prototype is an enhanced version of previous prototype. It allows early involvement in the development process instead of analyzing and designing for a long period of time. The Figure 1-1 illustrates our research work activities and development cycle.
1.4 Research work Activities

In this thesis, the research work performed is divided into five steps: requirement analysis, study, design, implementation and testing.

**Requirement analysis:** The first step is requirement analysis. This thesis aims to design a multi-core VoIP client and server. The server and client should utilize the additional processing power of multi-core CPUs. At this phase we had an abstract idea of end result so, we mustered detailed requirements and scope of the thesis work.

**Study:** This step involves a study of related tools and technologies. We focused mainly on network programming, inter-process communication, and parallel programming. We elected C, C++ and Qt as programming languages. We studied SIP (session initiation protocol) and OpenSIPS (Open SIP Server).

**Design:** Since, VoIP server and SIP server analyzer are automated softwares, so our design consists of flow charts and text describing sequence of events. As for graphical VoIP client which is semi-automated (involves user experience), we need to analyze how users are likely to use this product. Our design of VoIP client consists of flow diagrams and storyboard (a series of illustrations and images for the purpose of pre-visualization).

**Implementation:** It involves writing, testing, debugging and maintaining source code. We have implemented a VoIP client and software model for multi-core namely IOpenVoIP. We have also developed a SIP based graphical client (FeeaVoIP). In addition to client and server; we have developed a SIP server analysis tool called OVSA (Open VoIP Server Analyzer).

**Testing:** We verified the functionality and performance of the developed product after every prototype.
1.5 Structure of the Report

Chapter 2: Background gives a detailed literature review. It enables the reader to understand related tool, technologies and concepts used in this thesis work.

Chapter 3: VoIP Server and Client Model describes C-based paralleled VoIP-server model that suits multi-core. In addition, it presents multi-core server design.

Chapter 4: Open VoIP Server Analyzer presents the server analyzer tool. Moreover, it presents the results produced by the analysis of OpenSIPS server.

Chapter 5: Multi-core Graphical VoIP client Model discusses the design and implementation of graphical VoIP client.

Chapter 6: Conclusions concludes the work in this thesis. In addition, interesting ideas are pointed out for potential future works.
Chapter 2

Background

This chapter helps the reader to understand tool, technologies, and concepts used in this thesis project. It enables the reader to establish background knowledge on VoIP and multi-core systems. Moreover, this chapter provides detailed overview of SIP-base IP telephony systems. This chapter consists of four sections. Section 2.1 describes the Session Initiation Protocol (SIP). Section 2.2 introduces the reader to the new era of multi and many-core systems. Section 2.3 gives details about the Open SIP Server (OpenSIPS). Section 2.4 gives an introduction to QT, a cross platform, graphical application development framework.

2.1 Session Initiation Protocol (SIP)

Session Initiation Protocol (SIP)[2] is an application-layer control (signalling) protocol for session management with one or more participants. The main functions of SIP are: Address resolution, session setup, media negotiation, session modification, session termination, session cancellation and call control[3]. SIP is an application layer protocol and it can use different transport mechanisms such as UDP or TCP. SIP is not an entire communication system. It is rather a component that can be used with other protocols to build a complete communication system[2]. Typically, SIP is incorporated with protocols such as the Real-Time Transport Protocol (RTP), Real Time Streaming protocol (RSTP), Session Description Protocol (SDP) etc.

RTP[4] is used for transporting real time data such as interactive audio and video,
and it also provides QOS feedback. The main functions of RTP are payload type identification, sequence numbering, time stamping, and delivery monitoring. RTSP[5] is used to control the delivery of streaming media (both stored clips and live streaming). Furthermore, it can control multiple delivery sessions. SDP[6] is intended for describing multimedia communication sessions. The main functions of SDP are session announcement, session invitation, and parameter negotiation. On the other hand side, SIP is not dependent on protocols mentioned above and changes in these protocols do not require changes in SIP protocol. Due to this fact, today SIP is widely accepted as a standard protocol for VoIP.

2.1.1 Components of SIP

The main functionality of SIP is to establish and tear down multimedia communication sessions. Other functionalities are user availability and media negotiation. SIP components are used to implement these functionalities. The two main components of SIP architecture are: SIP servers and user agents.

2.1.1.1 User Agents

User agents (UA) are end devices that send and receive media. They are SIP phones running on a dedicated hardware or a software program running on personal computer. Furthermore, they use SIP protocol to initiate, answer/reject, hold/unhold, and transfer a SIP call. Every UA is further divided into two sub-components, UAC (User Agent Client) and UAS (User Agent Server). An UAC initiates SIP calls, it sends SIP requests and waits for answers for those requests. Whereas, UAS waits for the requests, produce response, and send it back to UAC. An UA acts like both UAC and UAS. During session establishment, both UAS and UAC are typically used.

2.1.1.2 Servers

User agents can communicate directly with each other, but it is not practical in reality. Servers are intermediary network devices that are located within the SIP-enabled network. SIP Servers enables SIP end-points to discover one another over a large network such as Internet. Furthermore, SIP severs process user-agents requests
and exchange SIP messages between end-points. Servers keep track of user locations and they perform user authentication. There are three type of SIP servers defined in SIP baseline specification RFC 3261 which are proxy server, redirect server, and registrar server.

2.1.1.3 Proxy Server

Proxy servers routes SIP requests coming from UAC to UAS and responses from UAS to UAC. The routing process can involve one or many proxy servers. The responses take the same path as requests. The SIP specification RFC 3261 defines two types of SIP proxies: stateless proxy and stateful proxy. Stateless does not maintain the state of the transaction while the stateful proxy maintains the state of the transaction. The stateful proxy retains more memory and processing power but it is efficient in terms of calls management.

2.1.1.4 Redirect Server

Redirect servers indicates where the requests should be tried. When a recipient moves from its original location, the redirect server responds with 3xx (redirection) responses. Redirection responses assist the sender to contact an alternate URI (Unique resource Identifier). The client sends the request to the alternate URI.

2.1.1.5 Registrar Server

User-agents send REGISTER requests to registrar server to update its location in location server or another database.

2.1.2 SIP Messages

SIP is a text-encoded protocol that uses UTF-8 encoding. SIP uses a similar semantic to Hypertext Transfer Protocol (HTTP). SIP messages use the generic Internet message format, which is the basic Internet message format defined in RFC 2822[7]. All SIP messages consist of start line, single or multiple header fields, an empty line, and an optional message body. Every line in the SIP message must be terminated by
line-feed sequence CRLF (Carriage-Return and Line-feed). The start-line identifies
the type of the message. SIP messages are of two types i.e. requests or responses.

2.1.2.1 SIP Requests

SIP requests are distinguished from SIP responses by the start-line of the SIP mes-
message. In case of SIP request the start-line of the message is called Request-Line. The
elements of Request-Line are: method name, followed by a space, a Request-URI,
followed by a space, and the SIP protocol version, with each element is separated
by a single space character. Request-Line ends with a CRLF sequence. There are
six basic methods defined in RFC 3261[2] for the client requests:

REGISTER is used for registering user information in the Central SIP server.
INVITE is used to request a callee for session from caller.
ACK is used to acknowledge the session by callee to caller.
CANCEL is used for session cancellation.
BYE is used for session termination.
OPTIONS are used to query optional capabilities.
SIP methods and their detailed description can be found in Table A-2, Appendix
A.

2.1.2.2 SIP Responses

SIP responses are distinguished from SIP requests by the start-line of the SIP mes-
message. SIP response has Status-Line as its start-line. The elements of Status-Line
are: protocol version, a status code, and its Reason-Phrase. Each element of Status-
Line is separated by a single space character. The status code is a number from
100-699. There are five major classes of responses which are as follows:
1xx (Provisional): Tells its recipient that a request is received but not processed
yet.
2xx (Success): Final response of the request.
3xx (Redirection): Used to redirect a caller.
4xx (Client Error): Indicate syntax error in request or request cannot be fulfilled.
5xx (Server Error): Tells client about problem on Server.
2.1.3 SIP Headers

SIP header consists of information that is useful for SIP components. A SIP header consists of one or more header fields. Each header field consists of field name, followed by a colon, and its value. There are seven basic header fields defined in RFC 3261 which are:

**Via:** When a SIP message traverses through the network every proxy adds its information to prevent looping. Furthermore, it shows the transport protocol used.

**Call-id:** Call-id uniquely identifies the transaction. It is used to identify the context.

**CSeq:** It is a random number which uniquely identifies a message.

**From:** It is the address of the initiator of the request.

**To:** It is the address of the recipient of the request.

**User Agent:** It is the name of the client.

**Contact:** It tells the SIP components where to send future requests.

SIP headers and their detailed description can be found in Table A-2, Appendix A.

2.1.4 SIP Conservation

![Figure 2.1: Example of SIP Conversation](image-url)
Figure 2-1 presents an example of SIP conversation between UAs. The following picture shows a typical exchange of SIP requests and SIP responses between caller and callee. The transaction starts when caller sends **INVITE** request to callee through a proxy server in its domain. Secondly, the proxy server sends a “**TRYING 100**” to inform caller about and received request. At the same time, the proxy finds callee in the same domain and forwards the request. When the **INVITE** message is received by callee UA, it replies back with a “**180 RINGING**”. Once callee responds the call, his UA sends “**200 OK**” to the caller. Similarly, caller sends an **ACK** to confirm the media session. From this point on, the media session has been successfully established between caller and callee. When any side disconnects the call, his/her UA sends a **BYE** message which is acknowledged by “**200 OK**” message and finally the session is terminated.

2.2 Multi-core

In modern computer architecture, multi-core is the term which is used to describe multiple CPUs working together on a single microprocessor chip. Speeding up processor frequency is way of the past, as the trend was ended in the earlier part of previous decade. Microprocessors heat up and become difficult to cool at higher clock speeds, and it is estimated that power consumption increases by 60 % with every 400 MHz rise in clock speed[8]. Greater overall system performance (less heat, power and cooling requirements) may be achieved by increasing the number of processing units, each running at normal clock speed. Figure 2-2 shows a multi-core CPU chip which replicates multiple processor cores on a single chip.

Figure 2.2: Multi-core Chip Architecture
These cores can run multiple threads in parallel and operating system takes each core as a separate CPU.

2.2.1 Programming for Multi-core

It is generally believed that an application running on a quad-core machine is four times faster than an application running on a machine with a single core processor. On the contrary, it is not true most of the time. Since, applications are not programmed to take full advantage of multi-core processors. For example, a conventional computer program loads a long set of instructions into memory. Instructions are executed one by one from the start to the end until it reaches to the end of the program. Such program execution on quad core machine only utilizes a single core out of four cores. On the other hand, execution time can be reduced by one fourth if we divide the similar program into four subsets of instructions and execute each subset of instructions in parallel by a separate core. Multi-core CPU offers an advantage over single core when we parallelize our code. Therefore, Instead of sequential code programmers must program by using threads or processes. In summary, for multi-core machines applications must be programmed with simultaneous execution in mind because Multi-core CPUs only offers an advantage over single core when we parallelize our code.

2.2.2 Parallelization

Parallelization (also known as parallel computing) is the only technique to reduce the execution time on multiprocessors. In the simplest sense, parallelization refers to breaking a big problem into smaller problems which are solved concurrently by the use of multiple processing elements (CPU cores). The figure below shows a big computing problem broken apart into discrete problems that can be solved simultaneously on a quad core machine.
Ideally, by using 4 cores the program execution time can be reduced to one fourth of the single core execution time. But the fact is not true in reality and Amdahl’s law is used to predict the effect of multiple processors on execution time. Amdahl’s law\[9\] states that if a fraction f of program is parallelizable and the remaining fraction, 1-f, is totally sequential then the speed up on n processors is governed by:

$$\text{Speedup}(f, n) = \frac{1}{1 - f} + \frac{f}{n}$$

Whereas, Gustafson law \[9\][10] argued Amdahl’s law and it states that a machines with larger computation capability operate on large data sets which increases the size of the problem. According to this law:

$$\text{Speedup} (P) = P - a(P - 1)$$

Where, P is number of processors and a is non-parallelizable fraction of program.

Threads and processes are the main methods of parallelization which are discussed in the section 2.2.3 and section 2.2.4.

### 2.2.3 Threads

A thread is a smallest unit of processing that is scheduled for execution by the operating system on a core[11]. In other words, it is a stream of instructions that executes concurrently and independently. A computer program consists of one or more threads (multi-threaded) where each thread executes independent and concurrent tasks. Threads can improve the throughput and efficiency of a program, by utilizing multiple cores concurrently. Threads do not have their own address space. Rather, they use the address space of the process that created it. Threads are not secure because one thread can write the memory used by other thread. Threads of same process can directly communicate with each other. Context switching between threads is cheaper than processes.
2.2.4 Processes

A process is unit of work created by the operating system\[11\]. It can be assigned to, and executed on, a specific core of processor. A computer program may consist of multiple processes and a process may consist of multiple threads. In other words, system resources are shared primarily by processes and process resources are shared by threads. Processes have their own address space which provides security and isolation because one process cannot corrupt another process. A process consists of: sequence of instructions (code), stack, and data (heap and program variables). A process needs inter process communication (IPC) mechanisms to communicate with other processes. The common IPCs are discussed in the next section.

2.2.5 IPC

IPC\[12\] stands for inter-process communication. IPC mechanisms are ways of message passing between different processes. There are several IPC mechanisms, those used in this thesis are discussed below.

2.2.6 FIFOs

FIFO\[12\] (first in first out) is a unidirectional communication channel, similar to a pipe. But unlike pipes which are used for communication between related processes (childs and parent processes), FIFOs allows unrelated processes to communicate with each other. A FIFO IPC mechanism provides one way (half-duplex) flow of data between processes. FIFOs are also called “named pipes” because they are identified by Unix-like path names. A FIFO can be used by multiple processes for reading and writing. The data is stored in kernel buffers. The kernel passes data between processes without storing it on file system. Furthermore, a FIFO has a read end and a write end. In order to pass data, pipes must be opened from both ends. But unlike pipes, FIFOs can read and write even if no one has opened on the other side. Such FIFOs are “non blocking”.

14
2.2.7 Message Queues

Message queues are link list of messages used for inter-process communication. Unlike pipes and FIFOs message queues are asynchronous, since sender and receiver of the message do not need to open the message queue at the same time. Message queues are slower than pipes and FIFO but provide better functionalities. Sender process can assign a priority to the message. A message with higher priority is added on the top of the link list. Message queues have kernel persistence. Messages remain in the “message queue” even after sender of the message terminates.

2.3 OpenSIPS - Open Source SIP Server

OpenSIPS (Open SIP Server) is an open-source implementation of SIP server and it is based on SIP RFC 3261. OpenSIPS is a continuation of OPENSER project. According to the documentation provided on its official website OpenSIPS is a robust and efficient SIP registrar server, location server, proxy server/router, and redirect server. Furthermore, it offers many features such as small footprint, support of UDP/TCP/TLS/SCTP transport layer protocols, plug and play module interface, stateless and state-full SIP Proxy Processing, IPv4 and IPv6, IP Blacklists, Authentication Authorization and Accounting (AAA), SNMP, etc.

OpenSIPS can be configured by editing the configuration script or by OpenSIPS-CP (OpenSIPS Control Panel). OpenSIPS-CP is a graphical web based control panel application for provisioning OpenSIPS. Installation and configuration of OpenSIPS can be found in Appendix B.

2.4 Qt

Qt is a platform independent application and UI development framework written in standard C++. Qt is pronounced as Cute. Using Qt, applications can be programmed once and run across all major desktop and mobile operating systems. Its code can be compiled on every platform that has a C++ compiler. Developers just need to compile the sources on all platforms where they want to run application. Initially, Qt was designed to create rich GUIs (Graphical User Interfaces) but, now it supports Databases, Multimedia, Networking, Scripting, etc. Applications
which are programmed in Qt give native look and feel on every platform. Qt/C++ has comparable programmer efficiency to that of the Java/AWT/Swing but, it is inferior when it comes to run time and memory efficiency[16]. In short, C++/Qt is better solution, particularly for GUI applications.

2.4.1 Qt Creator IDE

Qt Creator IDE (Integrated Development Environment) is designed to fulfil the needs of Qt developers. It encompasses every tool required for application development process, from UI (User Interface) design to application testing. IDE provides an intelligent C++ code editor which speed up code writing, such as syntax correction features, code suggestions (auto-complete), error highlighting, etc. It has an Integrated UI designer which provides drag and drop GUI layout and forms builder for desktop and mobile UIs. Moreover, it provides project and build management tools, such as version control and debuggers.

2.4.2 Qt Meta-Object System

Qt extends the standard C++ by Meta-Object System. It provide signals and slots mechanism for inter object communication, run time type information, and dynamic property system[17]. Whenever Qt compiler founds a QObject macro inside the C++ source file it generates another source file containing Meta-Object code. The compiler is called Meta-Object Compiler(MOC).
Chapter 3

VoIP Server and Client model

This chapter discusses the C-based paralleled VoIP-server model (called IOpenVoIP Server) that suits multi-core and that can be used as a simulation tool for multi-core SIP server. Our main focus in this chapter is to discuss the design of the client and server model. We have also proposed a design for a modular multi-core server in C based on our results of analysis of model. Section 3.1 is a brief introduction of the chapter. Section 3.2 introduces the reader to IOpenVoIP and describes the features of our Client and Server Model. Section 3.3 describes the Server model. Section 3.4 presents the client model. Section 3.5 shows the bottlenecks in performance of the server due to single core. In section 3.6 we have proposed design for multi-core server in c language. The user manual of IOpenVoIP is provided in Appendix D.

3.1 Introduction

Enhanced multi-core processors are becoming available every next few days in computing industry. Whereas, software developers are not aware what multi-core can do for them. When writing a computer program most developers prefer to extend or enhance existing source codes (libraries, algorithms) which are sequential in nature. Developers often dont like to parallelize existing source code because it is complex to do so and it provides no assurance of success. Transition from sequential programs to parallel programs require analysis of existing sequential code, performance estimation, re-implementation, verification and continuous tuning until desired results are achieved. Hence, rather than going through complex and uncertain development cycles a better approach could be to write softwares from the scratch. This chapter
presents a simple model of parallel VoIP server programmed from scratch which can be very helpful in analyzing and understanding the behaviour and bottlenecks of the system.

## 3.2 IOpenVoIP

IOpenVoIP is a VoIP Paralleled Client-Server model for Multi-core Server and Client. Since it is under the GNU General Public License (GPL) its source code is free to download from http://iopenvoip.lincisco.com. The major components of the system are IOpenVoIP Client and IOpenVoIP Server. The IOpenVoIP communicates with the IOpenVoIP server to establish multimedia sessions with other IOpenVoIP clients across the Internet. It is a concurrent server and it can process multiple clients simultaneously. IOpenVoIP (Server and Client) are intended for audio, video and data communication over the Internet. IOpenVoIP is programmed in GNU C.

![IOpenVoIP Server and Client](image)

**Figure 3.1: IOpenVoIP Server and Client**

The Figure 3-1 above shows the schema of client server relationship of the system.
3.2.1 IOpenVoIP Server Features

The major features which Server contains are the following:

**MYSQL Installation and Configuration:** MYSQL is a relational database management system (RDBMS) that run as a server and provide multi-user access to a number of databases. IOpenVoIP Server installs and configures MYSQL automatically with a Linux script to perform its operations.

**Concurrent Server:** Since, it can process thousands of client requests at the same time it is a concurrent server.

**User Registration:** It is a registrar servers because it stores users information into database efficiently avoiding duplication.

**Presence Server:** It is a presence server since it keeps track of user presence information and sends user status information to interested users.

**Easy Configuration:** It is very easy to configure, it can be configured by a text file provided with the source code namely `iopenvoip.conf`.

**Secure Communication:** It encrypts all the communication between clients and server. It supports all major encryption algorithms. By default it uses AES 64 bit encryption.

**Flexible Source Code:** It is programmed keeping in mind the future enhancements.
3.2.2 IOpenVoIP Client Features

The major features of the client are the following:

**User Registration:** Users can be registered on the server with this client.

**User Authentication:** Client supports password based authentication. Client takes user name and password and sends it to server for validation.

**User Search:** User search function allows searching other registered users on IOpenVoIP server.

**Add Contacts:** After successful user search Add contact function adds the contact information into local database.

**Contact status:** Contact status function displays the contacts presence (online or offline) information.

**Audio and Text messages:** Users can send audio and text messages to each other.

**Secure Communication:** Communication between clients and server is encrypted. Client supports all major encryption algorithms. By default it uses AES 64 bit encryption.

**Easy configuration:** Client can be easily configured by a text file provided with the source code namely `client.conf`. 
3.3 Server Design

Figure 3-2 shows the server modules and their interaction with each other. The numbers in figure below indicates step by step data flow of the server. When the program begins the module file_reader loads configuration parameters from a text file (Step 1).

The **start** module is interactive command-line user Interface that takes configurations and commands from the server administrator (Step 2). In Figure 3-2 the **server** module puts the server into listening state (Step 3). When a client creates a TCP or UDP connection to the server the UNIX fork function creates a new child process to achieve concurrency (Step 4). The **irecv** module reads the client...
requests from the receiver buffer, send it to the **Decrypt** module for decryption (Step 5). The **processvoipclient** module is a parser which takes the client request from **irecv** module and after parsing the request string sends the request parameters to the appropriate module for further processing (Step 6). The IOpenVoIP server supports user registration, user authentication, user search, user presence, and call-setup. Other modules are event driven. If the request is of user authentication the **loginhandler** and **user_auth** module performs authentication (Step 7). If the request is of user registration the **registration** module performs the registration (Step 7). If the request is of user search the **searcher** module performs the search operation (Step 7). If the request is of user presence **status_checker** module performs the operation (Step 7). If the request is of call setup the **call_setup** module is used to establish a VoIP call (Step 7). The **processvoipclient** module sends the response to the **isend** module which encrypts it and sends it back to the client (Step 8).

The **timeofday** and **timediff_val** modules calculates the total duration of the call. The server stores users information, passwords, call records, timing information, login information, user status information in to local MYSQL data base namely **iopenvoip.users**.

The **Figure 3-2** is drawn assuming only one user is connected to the server as only one child process is running at the moment. The server creates a new process for each client and terminates when client tear off the connection.
3.4 Client Design

Figure 3-3 shows the client modules and their interaction with each other. The numbers in figure below indicates step by step data flow of the client. The module \textit{file reader} loads configuration parameters from a text file, the important parameters include \textit{audio receive port}, \textit{text receive port}, \textit{private keys} and other settings (Step 1).

![Client Design Diagram]

Figure 3.3: IOpenVoIP Client Design

The \textit{start} module establishes a connection to the IOpenVoIP server (Step 2). Moreover, it calls the \textit{cserver} module from a sub process (multitasking) which listens for connections on voice and chat port using non-blocking I/O. The \textit{cserver} module
calls the `processaudio` or `processchat` module to process voice and text messages (Step 4). The `processvoipclient` module is interactive command-line user Interface by which users can register themselves on server and it performs user authentication (Step 4). The `registration` module performs user registration at server (Step 5) and `login` module performs user authentication at server (Step 6). Once the user is authenticated the user client menu is displayed on the screen by `client_menu` module (Step 7). The client supports voice calling, instant messaging, user search, saving searched contacts to local database, and updating contacts in local database. The `contact_updater` module synchronizes contacts information of `Iopenvoip.clients` database with server database. It reads the entire contacts user tables and every single contact is synchronized by `update_contact` module (Step 8). The `send_im` module sends instant massages to other users and `send_voice` module makes a voice call with other user.
3.5 Tests and Results

In this section, we measure the performance of the IOpenVoIP server by calculating the call-setup time at varying loads. These tests were performed by OVSA tool which is discussed in next chapter of this thesis. We test the server performance on simple hardware (CPU: Intel Dual CPU E2140 @ 1.60GHz, 1024 KB Cache, 1GB RAM). The first experiment to calculate the call-setup time was performed with just once call on the server. We increased the number of simultaneous calls up to one thousand during the following test. In order to get accurate results, each experiment was performed ten times and average call-setup time is calculated. Figure 3-4 is a visual representation of the relation between the number of simultaneous call and its impact on call-setup time. The x axis (horizontal) shows the number of calls and y axis (vertical) shows the time in microseconds. Figure below clearly demonstrates the decrease in performance and increase in call-setup delay on the server side by 275 %.

![Average call setup on varying load on server](image_url)

Figure 3.4: IOpenVoIP Call Setup Time
3.6 Proposed Multi-core Server Design

Figure 3-5 shows the proposed multi-core server design. The design shows modules, processes, and their interaction with each other. The numbers in figure below indicates step by step data flow of the multi-core server.

The multi-core software design presented in the figure above utilizes processes and
threads. In the design, we divided previous IOpenVoIP server code into separate applications (subsystems). Subsystems are 1000 or less lines of code and they perform only few concurrent and related tasks. The code was portioned into sub-systems to perform one major service e.g. user search, user registration, etc. The sub-systems are Linux processes in our design and they are containers of threads. The sub-systems communicate via inter-process communication mechanisms and IPC router. Code isolation makes code more robust and maintainable. Furthermore, multi core processors can be utilized by processing each subsystem on a separate CPUs core and communicates with each other by a well-defined IPC interface.

Similar to previous server design, in the multi-core design when the server program begins the module `file_reader` loads configuration parameters from a text file (Step 1). The `start` module is interactive command-line user Interface that takes configurations and commands from the server administrator (Step 2). In Figure 3-5 the `server` module puts the server into listening state. Server listens for client TCP or UDP connection and for every connection the UNIX fork function creates a new child process to achieve concurrency (Step 3). The `processvoipclient` module is a SIP parser which processes client requests and after parsing the request strings sends the parameters to IPC router process. The IPC mechanism used at this point is Posix message queue which is discussed earlier in section 2.2.4. The message queue stores the requests in a link list. Requests i.e. call-setup are placed at the top of link list with a higher priority (Step 4). The `IPC Router process` reads the message queues for clients request and sends the request to a particular process for further processing. If there are multiple processes of same type are running the router forwards the request to a particular process on the basis of round robin algorithm (Step 5). Processes manager module binds the `call-setup`, `authentication`, and `searcher` process to one or more CPUs by changing the affinity mask. Moreover, it assigns priority to processes. Processes register themselves to the router process when they come to life (Step 6). Requests are processed and reply is sent back to the client (Step 7).
Chapter 4

OpenVoIP ServerAnalyzer

This chapter discusses the Open VoIP server analyzer (OVSA) which we have developed. Moreover, it presents the results produced by the analysis of famous open source SIP server namely OpenSIPS. These results can determine the capacity and bottlenecks of server which is part of our research to design multi-core VoIP servers. Section 4.1 is a brief overview of OVSA. The requirements of the tool are addressed in Section 4.2. Section 4.3 presents the design and implementation of OVSA. Section 4.4 describes the topology and experimental environment. The results are presented in section 4.5.

4.1 Introduction

The throughput of a SIP Sever can be determined by the total number of SIP requests it can process. Another important factor to access the performance is the time needed for processing a request. The common method for performance analysis of SIP server is to simulate SIP calls. The purpose of the simulation is to generate huge amount of SIP calls to the target system. There are several open source tools for performance evaluation of the SIP servers. Nevertheless, Interpretation of results and drawing conclusions are very difficult because such tools are non-generic and deep understanding of tools is required for customization. For these reasons, tools for performance evaluation that are easy-to-use and easy-to-customize are needed. Such tools should help in easy interpretation of results.
In this chapter, we present the Open VoIP Server Analyzer (OVSA) tool for simulating sip calls, capturing SIP traffic, and results generator. The OVSA tool has three parts:
(i) SIP traffic generator
(ii) Traffic analyzer
(iii) Results generator.

4.2 Requirement specification

This section describes the requirement specification of OVSA. Section 4.2.1 defines the required measurements. Section 4.2.2 discusses the call scenario that should be supported by our tool.

4.2.1 Required Measurements

We are focusing on two main parameters: call setup time and packet losses. In the context of SIP call-setup time is the total amount of time a caller waits to get an answer of SIP INVITE back from callee. The callee responds to SIP INVITE with SIP 200 OK message. The packet losses caused by server overload or network congestion cause delays in call-setup time. Server designers should predict and eliminate causes of packet losses in advance.

In summary, to do an accurate analysis of server performance we need to measure the amount of time SIP takes during every protocol interaction.

4.2.2 Call Scenario

Figure 4-1 depicts the SIP session that OVSA tool should support for analysis and benchmarking. We take the OpenSIPS as the target system (SIP proxy). The OVSA traffic generator should generate the sequence of SIP messages as shown in figure below. The timestamps (TC1 - TC4) denotes the time at which certain SIP message is sent or received by caller or callee. Whereas, the timestamps (TS1 TS4) denotes the time at which certain SIP message is sent or received by the target system.
CHAPTER 4. OPENVOIP SERVER ANALYZER

The similar SIP message flow is discussed earlier in section 2.1.4.

4.3 Design and Implementation

This section presents the design and implementation of OVSA. As mentioned earlier in section 4.1, that the OVSA consists of “SIP traffic generator”, “SIP traffic analyzer”, and “SIP results generator”. Section 4.3.1 presents the design of traffic generator. Section 4.3.2 discusses the traffic analyzer and result generator.

4.3.1 Traffic generator

The traffic generator consists of two C programs: caller.c and callee.c. The caller.c program simulates SIP user agents and acts like initiator of calls. Whereas, the callee.c program simulates SIP user agents and acts like the receivers.

Figure 4-2 shows the interaction of these programs with the target system. The figure is drawn assuming that only one SIP session is established through the
target system as one caller and callee sub-process is running. The **caller_main** and **callee_main** module can be configured to create thousands of SIP sessions. The **generate_invite**, **generate_register**, **generate_ack**, **generate_try**, **generate_ringing** and **generate_sdp** modules generate SIP messages.

![Traffic generator](image.png)

**Figure 4.2: Traffic generator**

### 4.3.2 Traffic analyzer and result generator

We have used Wireshark[18] as SIP traffic analyzer which is a free and open-source network protocol analyzer for UNIX and Windows. Wireshark stores the SIP captured packets in a file and typically its extension is pcap file format. “OVSA Results generator” which is a C program takes the packet capture (pcap) files and produce excel documents. The excel documents contains server time stamps (TS1-TS4) and caller time stamps (TC1-TC4) which were discussed earlier in section 4.2.2.
4.4 Topology and experiments

The detailed topology and experimental environment is illustrated in Figure 4-3. This infrastructure consists of traffic generator caller program that initiate calls to callee program. The callee programs replies with appropriate SIP messages. The call traverses through the target system. Our target system during the tests is Open source SIP server (OpenSIPS).

![Figure 4.3: Detailed testing strategy](image)

We capture SIP packets on traffic generators and OpenSIPS with wireshark. Later, the result generator program takes the packet capture files (.pcap) as input and produce user understandable office excel files. The excel files contains the timing information (timestamps). Using time stamps user can generate graphical diagrams.
4.4.1 Tests

We conducted nine tests to measure the packet loss and call-setup delay. Table 4-1 illustrates that only one SIP session was created during the first test and it is repeated 10 times for accuracy. We increased the number of SIP sessions gradually to evaluate the behaviour of the system due to load. The test 2 was performed to investigate the effects of network congestion on call processing times. During test 2 we have overloaded the network using Pktgen which is an open source Linux kernel module. The number of SIP sessions during each test is described in the table below. Furthermore, the test results are compared and discussed in next section.

<table>
<thead>
<tr>
<th>Test#</th>
<th>SIP sessions</th>
<th>Repetitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>300</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>1000</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 4.4: Test details
4.4.2 Results

This section presents the results of our tests. Figure 4-4 presents packet losses which clearly show that OpenSIPS cannot process every call request and is subject to SIP packet loss as the number of calls starts to increase. Not all losses are due to server loading, test 2 is the exception in Figure 4-4 that shows 40 % packet loss due to network congestion. Figure 4-5 to 4-10 shows that losses caused by ‘network overload” and ‘server overload” would definitely cause delays and packet losses during the SIP session establishment and would hence reduce the perceived service quality of the users. In order to be able to take counter measures developers and researchers should utilize multi-core processors efficiently to eliminate such effects.

![Percentage Loss Graph](image)

Figure 4.5: Packet loss
CHAPTER 4. OPENVOIP SERVERANALYZER

Figure 4.6: TC2 - TC1

Figure 4.7: TC3 - TC2
CHAPTER 4. OPENVOIP SERVER ANALYZER

Figure 4.8: TC4 - TC3

Figure 4.9: TS2 - TS1
CHAPTER 4. OPENVOIP SERVER ANALYZER

Figure 4.10: TS3 - TS2

Figure 4.11: TS4 - TS3
Chapter 5

Multi-core graphical VoIP Client

This chapter discusses the design and implementation of multi-core graphical VoIP client which we have developed. Section 4.1 is a brief overview of OVSA. The requirements of the tool are addressed in Section 4.2. Section 4.3 presents the design and implementation of OVSA. Section 4.4 describes the topology and experimental environment. The results are compared and discussed in section 4.5 and section 4.6 respectively.

5.1 Introduction

VoIP stands for Voice over Internet protocol. It is assumed that VoIP softwares only make telephone like voice conversation over the Internet. Nowadays, VoIP clients support high definition video as well as data communication. VoIP softwares are primarily of two types: clients and servers. The move towards high definition video and data processing has made multi-core platform an essential for VoIP clients because there is no other way around it. In this thesis, we have designed and implemented a graphical VoIP client. We demonstrated how a VoIP client can be divided into sub-systems and how they can communicate with each other. Similar to multi-core server design presented earlier real time tasks can be assigned a higher priority and they can be assigned to a specific CPU by changing affinity mask of the process.
5.2 Basic design of FeeaVoIP

FeeaVoIP has two layers: Application layer and PJSIP layer. The two-layered design separates the code of front end (the GUI) from the SIP implementation (PJSIP). The front end is programmed using Qt and C++. We have discussed Qt earlier in section 2.4. We have used PJSIP as back end of FeeaVoIP. PJSIP[18] is an open-source SIP and media stack. The front end consists of graphical forms, header files and programming logic. The GUI supports account management, contacts management, instant messaging, and voice calling. Whereas, PJSIP is used to performs SIP signalling and audio communication.

5.3 Cross platform graphical user interface

The graphical user interface (GUI) is designed to enhance the user experience. FeeaVoIP has a cross platform rich graphical user interface which is created using QtCreator. Qt creator has a designer window which makes use of drag and drop to create user interfaces. Designer window stores the description of designed form into an XML file that has .ui extension. Qt is equipped with a unique user interface compiler (UIC) to get cross platform development. UIC reads the XML description of user interface (.ui) and generates a platform specific C++ header file (.h). Finally, in order to use the GUI we include the header files into our C++ code.

![Figure 5.1: GUI with Qt Creator](image-url)
5.4 Objects interaction via signals and slots

Qt is equipped with signals and slots by which we can split applications into self-contained and small objects (classes). The objects interconnect with each other via Qt signals and slots. The signals and slots are dynamic mechanism to interconnect two existing classes. A signal is emitted when a particular event occurs e.g. **button clicked, status changed, instant text message received**, etc. A signal connects to any number of slots. When a signal is emitted it executes all slots connected with it. A slot is an ordinary method that performs a specific action e.g. **send instant message, make voice call, update contact information** etc. In order to create a connection between a signal and a slot the `connect` method is used. The figure below shows three objects interconnected via signals and slots.

![Signal and Slot mechanism](image)

**Figure 5.2: Signal and Slot mechanism**

5.4.1 FeeaVoIP GUI Interaction

Figure 5-3 shows how FeeaVoIP GUI forms interact with each other. The user interface compiler (UIC) reads the `addcontact.ui` (XML description of addcontact GUI) file and generates a platform specific C++ header file `ui_addcontact.h`. The following header file implements the GUI and it contains the methods by which in-
CHAPTER 5. MULTI-CORE GRAPHICAL VOIP CLIENT

...iteration with GUI. The **AddContact** is a self-constructed class which contains the methods, signal and slot to interact and control the **ui_addcontact** form. The **addcontact.h** file contains the function prototypes and type definitions of **AddContact** class. Whereas, **addcontact.cpp** file contains the source code. The **FeeaVoIP** is the core class (main window) of front-end. It implements a signal namely **On_addContactButton_Clicked** which emits when user clicks on **add-contact** button on GUI. In response of signal the **addcontact** form is displayed to the user. The main window interacts with all other forms in the same fashion. The complete GUI design is illustrated in next section.

Figure 5.3: AddContact form interaction with Main window
5.5 GUI Design of FeeaVoIP

Figure below shows the hierarchal GUI design of FeeaVoIP. The forms interact with each other by the same way as explained earlier in section 5.2.3. A specific form is created in the same manner as illustrated earlier in section 5.2.1. The graphical illustration and description of forms can be found in **Appendix C** of this thesis.

Figure 5.4: GUI design of FeeaVoIP
5.6 FeeaVoIP scalability on Multi-core

FeeaVoIP utilized multiple cores by splitting code into separate applications that use IPC (Inter-process communication) to communicate with each other. These applications can be scheduled to run on separate cores. The mechanism used to perform IPC is D-BUS protocol. QtDBusAPI[19] implements D-BUS protocol in Qt. The most important feature of QtDBus is mapping of Qt signals and slots to D-BUS. Therefore, signals emitted from an application can execute a slot in another application. Furthermore, QtDBus perform message routing and service activation. An application exposes itself by advertising remotely callable methods and signals (Interfaces) on the D-BUS. All communication done over the bus is done in the form of messages. Every object on a bus is identified by a unique address which consists of: interface, service and object name. Where, service name refer to an application connected to D-Bus and object name refer to a particular component of the application.

![IPC with QtDBus](image)

Figure 5.5: IPC with QtDBus

The Figure 5-5 shows the main application FeeaVoIP communicating with audio and video application over D-Bus. Audio and video applications are separate applications and can be scheduled to different CPU cores. The audio application handles incoming and outgoing audio traffic. On the other hand side, the video application handles incoming and outgoing video traffic.
5.7 Complete Design

Figure 5-6 explains the complete design of VoIP client. The application layer was discussed in previous sections. The PjSIP layer performs SIP signalling and audio communication. In the figure below, **PjCallback** is self-constructed class that interacts with PJSIP library. The main libraries used are PJMEDIA, PJSIP, PJSIP-UA, PJSIP, PJUA and PJSIP-SIMPLE[20].

![Diagram of FeeaVoIP design](image-url)
The PJMEDIA implements SDP (Session Description protocol) and it is a fully featured media stack. The PJSIP is a high performance and efficient media stack. PJSIP-UA is a high level user agent library. It performs client registration, client INVITE and Call transfer. The PJSUA is very high level API for constructing SIP User Agents. PJSIP-SIMPE is used for instant messaging and presence.
Chapter 6

Conclusion

6.1 Summary

The major goal of this thesis was to design and develop an open source VoIP server and client software that can help researchers, developers, and students who want to explore, improve and learn about multi-core VoIP softwares. Multi-core VoIP softwares are very complex as compared with traditional softwares for single core systems. We choose spiral software development model that allows incremental releases (prototypes or models) of actual software. We designed and implemented the VoIP server model (IOpenVoIP) in C. It is not a multi-core server; rather it is a concurrent server. IOpenVoIP is the first prototype of actual server intended to test the functionality of software. A client model is also implemented in C to perform the functional testing. The server performed as it was expected, clients were able to locate each other over internet through server, sessions were established successfully, and the communication between clients was successful.

For performance testing we have designed a developed a performance evaluation tool called OVSA. It measures number of communication session (calls) a server can process and call-setup times at various call rates. We performed evaluation of IOpenVoIP using this tool and noticed 275 % increase in call-setup delay on the server side.

We also studied multi-core systems on Linux and based on our study and analysis we presented a design of multi-core VoIP server model (second prototype). In the design, we decomposed IOpenVoIP server model into number of subsystems.
Subsystems are 1000 or less lines of code and they perform only few concurrent and related tasks. The code was portioned into sub-systems to perform one major service e.g. user search, user registration, etc. The sub-systems are Linux processes in our design and they are containers of threads. The processes communicate with each other via Inter process communication mechanisms and an IPC router. The IPC router routes the client requests to other sub-systems based on round-robin algorithm. Sub-systems that provide real time or delay sensitive services are assigned to a specific CPU by changing the affinity mask of the process and to avoid context switching delays we suggested to assign higher priority to processes such as call-setup.

OVSA can be used for performance evaluation of SIP servers. It can be helpful in determining the bottle necks in existing SIP servers. We have performed performance analysis of OpenSIPS server which shows losses and call setup delays directly proportional to server and network overloading.

We have also developed a graphical VoIP client (FeeaVoIP) using Qt and C++. We have developed the graphical user interface of graphical client and used an open-source SIP stack PJSIP for SIP signalling. We presented a methodology for graphical SIP clients so that they can utilize multi-core processors in an efficient way.

6.2 Future Work

The design of the multi-core VoIP server is presented based on our study however the second prototype needs to be implemented and verified. The similar methodology for functional and performance testing can to be adopted which we used for IOpenVoIP server model, until a stable and robust server software is achieved. Moreover, the performance evaluation tool and results of OpenSIPS analysis can be used to decompose the processor hungry parts of existing server code into sub-systems and scheduled them on independent processor cores.

Our graphical VoIP client demonstrates how tasks can be decomposed into sub-systems. Furthermore, it presents a communication paradigm via Qt-DBus. Hence, tasks like HD video processing can be decomposed into smaller sub-systems.
Chapter 7

Bibliography


   http://www.ietf.org/rfc/rfc3261.txt


   http://www.ietf.org/rfc/rfc1889.txt

   http://www.ietf.org/rfc/rfc2326.txt

   http://www.ietf.org/rfc/rfc2327.txt
   http://www.ietf.org/rfc/rfc2822.txt


   http://www.opensips.org/

   http://opensips-cp.sourceforge.net/

   http://qt.nokia.com/

[16] Matthias Kalle Dalheimer, "A Comparison of Qt and Java for LargeScale, Industrial-Strength GUI,” Klarlvdalens Datakonsult AB,. 
    http://doc.trolltech.com/4.2/metaobjects.html

    http://www.wireshark.org/


    http://www.pjsip.org

# Appendix A

## SIP Miscellaneous

<table>
<thead>
<tr>
<th>SIP Headers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>via</td>
<td>It is used to record the route taken by SIP request and is used to route a response on same path back to the originator.</td>
</tr>
<tr>
<td>To</td>
<td>Shows the recipient of the requests. It contains the name and SIP URI of the recipient.</td>
</tr>
<tr>
<td>From</td>
<td>Indicates the initiator of the request. Similar to To header it contains the name and URI of the initiator.</td>
</tr>
<tr>
<td>Max-Forwards</td>
<td>Maximum number of hops that a SIP request may take is specified in Max-Forwards header field. At each hope the value of the field is decremented by one.</td>
</tr>
<tr>
<td>Call ID</td>
<td>It is used to uniquely identify the call between two user agents. Call-ID is generated by user-agent and never modified by servers.</td>
</tr>
<tr>
<td>CSeq</td>
<td>It contains a decimal number (sequence number) which identifies the order of transaction.</td>
</tr>
<tr>
<td>Contact</td>
<td>Identifies the request originator. It contains the SIP URI.</td>
</tr>
<tr>
<td>Content-Type</td>
<td>Indicates the Internet media type in the message body.</td>
</tr>
<tr>
<td>Content-Length</td>
<td>It is used to indicate the number of octets in the message body (Length of message content).</td>
</tr>
</tbody>
</table>

Table A.1: Table A-1: SIP Headers (summarised from [21])
### APPENDIX A. SIP MISCELLANEOUS

<table>
<thead>
<tr>
<th>SIP Methods</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVITE</td>
<td>Indicates session setup request from a caller to callee.</td>
<td>RFC 3261</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement of final response to Invite</td>
<td>RFC 3261</td>
</tr>
<tr>
<td>BYE</td>
<td>Indicates session termination by either caller or callee.</td>
<td>RFC 3261</td>
</tr>
<tr>
<td>CANCEL</td>
<td>Cancel any pending session by either caller or callee.</td>
<td>RFC 3261</td>
</tr>
<tr>
<td>REGISTER</td>
<td>Register the address mentioned in SIP URI in to Servers DB.</td>
<td>RFC 3261</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Query additional capabilities of Servers.</td>
<td>RFC 3261</td>
</tr>
<tr>
<td>REFER</td>
<td>Transfer user to a URI (Call transfer)</td>
<td>RFC 3515</td>
</tr>
<tr>
<td>UPDATE</td>
<td>It updates session information without changing the state of dialog.</td>
<td>RFC 3311</td>
</tr>
<tr>
<td>PRACK</td>
<td>Provisional response acknowledgement</td>
<td>RFC 3262</td>
</tr>
<tr>
<td>INFO</td>
<td>Mid-call signalling transport</td>
<td>RFC 6086</td>
</tr>
<tr>
<td>PUBLISH</td>
<td>Upload presence state to a server.</td>
<td>RFC 3903</td>
</tr>
<tr>
<td>NOTIFY</td>
<td>Transport of subscribed event notification.</td>
<td>RFC 3265</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>Request notification of an event.</td>
<td>RFC 3265</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>Transport of an instant message body.</td>
<td>RFC 3428</td>
</tr>
</tbody>
</table>

Table A.2: Table A-2: SIP Methods (adapted from [3])
Appendix B

FeeaVOIP User Manual

B.1 Splash Screen

Figure B.1: Splash screen
APPENDIX B. FEEAVOIP USER MANUAL

B.2 Main window

![Figure B.2: Main window - Contacts TAB](image1)

**Edit Contact Button**
Select the contact from the list and then click this button to edit the contact information.

**Delete Button**
Select the contact from the list and then click this button to delete the contact information.

**Connect Button**
SIP registration is performed using this button.

**Add Contact Button**
New users can be added by pressing this button.

**User Status**

**Contact list**

**Call status**

![Figure B.3: Main window - Direct call TAB](image2)

**Direct Call Button**
Enter the SIP URI in list box and click on this button to make a voice call.

**Chat Button**
Enter the SIP URI in list box and then click this button to open chat window.

**Monitor Button**
Click this button to see the logs from PJSIP (SIP STACK).

**Call Button**
Select the contact from the list and then press this button to make a call.

**Settings Button**
Click this button to configure the client.

**Calendar**
Click on the calendar to see your appointments.
B.3 Other forms

Figure B.5: Setting Window
Figure B.6: Log Window

Figure B.7: Add Contact form
Appendix C

IOpenVOIP Users

C.1 Server User Interface

The graphical user interface of the IOpenVoIP server will appear as shown in Figure D-1.

Figure C.1: Server User Interface
C.2 Client

Figure D-2 shows the user registration process. Only ID and password is required for registration. Client submits all other registration parameters from the configuration file i.e. IP address, port numbers, etc.

![User Registration](image)

Figure C.2: User Registration

Figure D-3 shows user authentication at server via IOpenVoIP client.

![User Login](image)

Figure C.3: User Login
Figure D-4 shows user search at server via IOpenVoIP client.

![User Search](image)

Figure C.4: User Search

Figure D-5 shows call setup through server between two clients.

![Call Setup](image)

Figure C.5: Call setup - Text Message