TOWARDS AN IMPLEMENTATION OF AVOIP: ANONYMOUS VOICE-OVER-IP

by

BIKAS RAJ GURUNG

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2008

To my family - who has always been there for me...

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervising professor Dr. Matthew Wright. He is kind, considerate, patience, friendly, helpful, knowledgeable and everything else that makes a great teacher. He is a good mentor and a great person. I find myself fortunate to get an opportunity to work with him on this research project. I really thank him for his patience towards me during my failures and shortcomings and for inspiring me to overcome them. I would also like to thank him for providing financial support for my studies.

I am grateful to Dr. Gergely Zaruba and Mr. David Levine for their interest in my research and for taking time to serve on my thesis committee.

I wish to thank all the members of iSEC lab at UTA for their help and support.

Most importantly I would like thank my parents and my brothers for their continuous support and encouragement.

November 19, 2008

ABSTRACT

TOWARDS AN IMPLEMENTATION OF AVOIP:
ANONYMOUS VOICE-OVER-IP

Bikas Raj Gurung, M.S.
The University of Texas at Arlington, 2008

Supervising Professor: Dr. Matthew K. Wright

Voice over IP (VoIP) is one of the fastest growing telecommunication market today. However with great popularity comes greater security risk. When we communicate over the Internet we expose our identity to all the prying eyes, even if we do not want to. Even though VoIP data can be secured by using encryption, sometimes the relationships between users can be sensitive. To provide stronger privacy for these users, we propose anonymous VoIP (aVoIP) - a means to secure user privacy in VoIP.

The biggest challenge in introducing anonymity to VoIP like system is in providing acceptable Quality of Service (QoS). Users expect a voice quality comparable to that of traditional telephone network. However to achieve anonymity we may need to employ cryptographic operations, random re-routing and traffic manipulation - all of which might add some extra delays in VoIP operation.

For aVoIP to be feasible, we need it to be reliable, scalable and provide acceptable QoS. Rather than reinventing a wheel, we looked for existing anonymity systems cur-

rently deployed over the Internet to base the architecture of aVoIP on. With hundreds of thousands of users all over the world, Tor has proved to be a reliable, scalable and stable anonymity system. Hence we foresee aVoIP to have architecture similar to Tor. However, Tor has been designed to provide low-latency anonymity only to TCP-based applications, whereas most of the real-time and multimedia systems need fast and efficient connection-less protocol like UDP. So in our quest to determine the feasibility of aVoIP over current Internet infrastructure, we worked towards modifying the Tor system to support UDP. We studied the Tor system in depth, analyzed the complications raised while supporting UDP, proposed a new design and implemented it. We have also developed a test environment to test our design over the PlanetLab nodes.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Anonymity

With more and more people relying on the Internet for both personal and commercial services, security and privacy have become a growing concern. Even though different security protocols have been developed to provide data security, sometimes user privacy may also be of greater interest. For example, people may want to publish socially sensitive materials but need to hide their location and identity in fear of retribution from oppressing regime or hard-line organizations, law enforcement agency may need to monitor certain sites without leaving its footprints, journalists may need to communicate safely with whistle-blowers without revealing each others identities and individuals may want to prevent websites tracking their web behaviors. To provide this blanket of security over user identity, we cannot rely only on encryption of data packets. Because of inherent nature of Internet protocols that requires source and destination address to be embedded on each packet for successful routing, user identity is easily revealed to anybody in a position to intercept/monitor the traffic between sender and recipient.

Anonymity by definition means a state of being unknown or unidentified. An anonymous communication system provides user privacy by hiding user identity and blending the user traffic among others with a goal of removing the linkability between sender and recipient. Out of several anonymous systems proposed, few have been successfully deployed over the Internet. Tor [4] is one of the most popular anonymous system currently deployed with thousands of users all over the world.

1.2 Anonymous Voice-over-IP(aVoIP)

Voice over IP (VoIP) or Internet Telephony, as the name signifies, is a process of transmitting voice over packet switched network or the Internet. With affordability, convenience and acceptable quality associated with VoIP, it is not surprising that it is one of the fastest growing telecommunication market today, specially in the area of long distance call and internal corporate network. However with increasing popularity comes greater security risks. Whereas Public Switched Telephone Network (PSTN) limits a security risk to a physical interception of its phone lines and equipments, VoIP with its voice packets routed through open and independent Internet provides a greater challenge in ensuring user privacy. As VoIP is a real-time multimedia application, it demands a good Quality of Service (QoS) to provide acceptable performance. For VoIP to be adequate for users used to the existing PSTN network, its voice quality and call setup functionality should be comparable to a traditional telephone network. However in order to provide user privacy in VoIP application, we need to tunnel voice packets through suitable anonymity system. Added cryptographic computations and random rerouting associated with any anonymity system introduces extra delay in already QoS-strained VoIP application. So a question arises - is the current Internet infrastructure capable of supporting *anonymous VoIP (aVoIP)* with acceptable performance? We seek to answer this very question with our first step towards the implementation of aVoIP.

As we mentioned earlier, Tor is one of the most popular anonymous system currently deployed over the Internet. Tor offers anonymity by tunneling an encrypted user traffic through virtual circuit created across randomly chosen multiple hops. Each hop in the path knows only its predecessor and successor, thus ensuring unlinkability of *initiator* and *responder*. Since its initial deployment Tor has already proved its stability,

scalibility and reliability by providing anonymity services to its huge userbase distributed all over the world. Hence we foresee our *aVoIP* system to have an architecture similar to Tor. However Tor has been designed to provide low-latency anonymity services only to TCP-based applications like web-browsing, instant messaging, secure shell etc. If we are to support real-time applications like VoIP, we have to transcend the inherent limitations of TCP protocol and rely on fast and efficient UDP. Thus we need *aVoIP* to utilize Tor like architecture based on UDP protocol.

With an introduction of UDP several issues arise, especially due to unreliable nature of UDP protocol. Even though we do not need guaranteed and in-order packet delivery for normal data traffic, we need reliability for the Tor circuit creation and management. Tor has been distributed as a free software under 3-clause BSD license and its source code is freely available. We get the latest stable version of the Tor and study its flow in detail. We analyze the issues raised in replacing TCP with UDP and seek appropriate solutions to the problems. We then implement our system-design and also develop a suitable testing environment to test it over PlanetLab [22] distributed network.

1.3 Thesis Organization

In Chapter 2, we cover the necessary background as a foundation of our system design. In Chapter 3 we identify issues with Tor in supporting UDP and discuss our solutions. In Chapter 4, we explain our experimental setup. Chapter **??** concludes with ideas for future work.

CHAPTER 2

BACKGROUND

In this chapter, we will provide brief insights into various components related to our work. We will first present a short introduction on low-latency anonymity system, discuss various anonymity systems proposed and proceed towards discussing the Tor. As our work revolves around modifying the Tor to support UDP communication, we delve little deeper into Tor operation and management. We then talk about VoIP, PlanetLab and few other supporting components. We also introduce various nomenclatures that will be used throughout the document in this chapter.

2.1 Low-Latency Anonymity Systems

Anonymity system can be broadly classified into two groups - high-latency and low-latency anonymity system. High-latency systems are appropriate for applications like email which do not require faster response whereas low-latency anonymity system enforces a limit on delays incurred and demands real-time or near real-time responses. Hence it is suitable for applications like web browsing, instant messenger, SSH etc.

Ranging from a system with only one server manipulating a traffic flow or concealing the sender's true identity to the complex systems with multiple nodes and random rerouting, low-latency anonymous systems aim to offer various aspects of anonymity, viz.: sender anonymity, recipient anonymity and sender-receiver unlinkability. Among them, simplest ones are single server based anonymous communication systems like Anonymizer [1], LPWA [2], FindNot [3]. With only one intermediate node in a path, this form of anonymous system combines simplicity with good performance. However,

presence of only a central proxy server results in a single point of failure and makes this kind of system quite vulnerable to an attacker who can monitor traffic entering and leaving the server.

Another variants of low-latency anonymous systems are circuit-based systems like Tor [4], WebMixes [5], Crowds [6], Freedom [7], Tarzan [8], MorphMix [9], Hordes [10], ISDN-MIXes [**?**] that route messages through multiple hops so as to hide the true identity of an initiator and/or recipient. When initiator wants to begin a session with a recipient, it first establishes a fixed route or *circuit* across a number of randomly chosen nodes in the network. Once the circuit is established, subsequent communication between initiator and responder takes place through that circuit. Each hop in the circuit shares pairwise symmetric keys with corresponding hop and this key is used to encrypt/decrypt the messages sent across the circuit. Different circuit-based systems differ in the ways subsequent nodes are chosen in the path. Some systems like Crowds and MorphMix allow each node in the circuit to randomly choose next one where as in a system like Tor, initiator determines all the nodes in the path. In whatever way the nodes in the path are selected, all the circuit-bases anonymity systems try to ensure that each node in the path only knows its predecessor and successor and nobody in the link. We will discuss in detail how the Tor establishes a circuit in the section 2.2.

These circuit-based systems come in various flavors with different design resulting in diverse system performance. Crowds, Tarzan and MorphMix incoporate Peer-to-Peer (P2P) architecture where all the peers participate in generating and relaying the traffic. In Tor and Freedom user establishes end-to-end circuits by randomly choosing a number of nodes from the network of servers. WebMixes, on the other hand, aspires to provide anonymity and unobservability in the Internet by tunneling the web traffic

through cascades of MIXes. Hordes uses multicast to provide initiator anonymity.Apart from difference in design, various systems offer mechanism like cover traffic, link padding, defensive dropping and traffic shaping as means to improve anonymity of the users which, in turn, may incur additional latency to the system. Even though extra delays can be attributed to circuit establishment and re-routing of the messages, these circuit-based systems provide higher degree of anonymity than their single server counterparts along with comparable performance appropriate for most Internet applications. In fact, some systems like Crowds, MorphMix and WebMixes have been specifically designed to provide anonymity for Web traffics. Tor, one of the most successfully deployed anonymous system in the Internet today, has now more than thousand servers and hundreds of thousands of users and is said to offer good performance in providing anonymity for various Internet application like web browsing, instant messaging, SSH, FTP etc.

Apart from Tor, another system of particular interest to us is ISDN-MIXes. Proposed by Pfitzmann et all, ISDN-MIXes aims to provide anonymity for telephony services in narrow-band ISDN network. Each subscriber maintains a set of two separate sending and receiving channels with its local exchange. Each local exchange has set of mixes called *MIX-cascade* that performs cryptographic operation and traffic manipulation to provide complete untraceability within a set of subscribers in that exchange. Apart from traffic shaping, ISDN-MIXes also employ layered encryption and dummy traffic. Call between A and B is established by connecting sender-channel of A with receiving-channel of B through each other's local-exchanges with/without long-distance network connecting the two exchanges. Even though ISDN-MIXes has been specifically designed to provide untraceable communication for telephony, its operation and management is tied to a service on physical medium. Hence it is not as relevant to us as Tor is.

Performance of anonymous system for general web browsing can be inferred from the successful deployment of low-latency systems like Anonymizer, FindNot, Tor and Java Anon Proxy (JAP) [11]. However, effect of multimedia communication in end-to-end performance of anonymized network has yet to be measured. Most of the multimedia traffic uses Real-Time Transport Protocol (RTP) over UDP for delivering audio and video over the Internet. Hence, anonymous systems like Tor, WebMixes, Crowds and MorphMix that utilize TCP as transport protocol present a significant obstacle in analysis of multimedia performance. Even though there are systems like Freedom and Tarzan which operate on IP layer, there has not been any extensive studies done to test the viability of anonymous system for multimedia communication.

2.2 Tor

Tor(The Onion Router) is a distributed overlay network that aims to provide its user a low-latency anonymous communication service. It is based on Onion Routing design originally developed by David Goldschlag, Michael Reed, and Paul Syverson [12] [13] [14] [15] and marks an improvement mainly in terms of perfect forward secrecy, directory servers, rendezevous points, end-to-end integrity checking and variable exit policies. It was first proposed by Roger Dingledine, Nick Mathewson, and Paul Syverson at the 13th USENIX Security Symposium [4] and has been succesfully deployed over the Internet with a very large userbase all over the world.

Tor protects user against traffic analysis where an attacker tries to link initiator with receiver by sniffing the packet sent over the newtork. Cryptographic manipulation like data encryption/decryption can only provide data security; but due to inherent nature of routing protocol, user identity is still visible to anyone who has an ability to read the packet headers. Tor provides user privacy by re-routing the package through

number of relays distributed all over the world in such a way that no single point along the path knows anyone except its predecessor and successor. We will describe how Tor achieves this next.

2.2.1 Tor operation

Tor is an overlay network that relies on volunteer nodes to relay the network traffic. Before delving into basic Tor operation, let's look into fundamental building blocks of Tor network:

- *Onion Proxy (OP)*: Onion Proxy is a point of contact for user application to Tor network. It is a Tor client running in a user's sytem. Applications like web-browser, messenger clients, ssh clients connect with OP using well-known SOCKS [16] protocol. OP is responsible for all the connection management and data handling on behalf of a user.
- *Onion Router (OR)*: Onion Router is same as OP but with added responsibility of relaying traffic from other users. It can be a dedicated server or a normal system that has agreed to be an OR.
- *Directory Server (DS)*: When a client first joins a Tor network, it needs to have some information about current network status and lists of ORs available. These informations are maintained by a group of dedicated well-known routers called Directory Servers. Each client has a list of directory servers that comes pre-loaded when it installs Tor software. ORs periodically updates its status to directory server which then use them along with its own observation to publish network status. Apart from DS, Tor also encourages ORs with sufficient capability to provide directory services using cached status in order to better handle the load on directory servers. Client connects with directory servers using HTTP protocol.

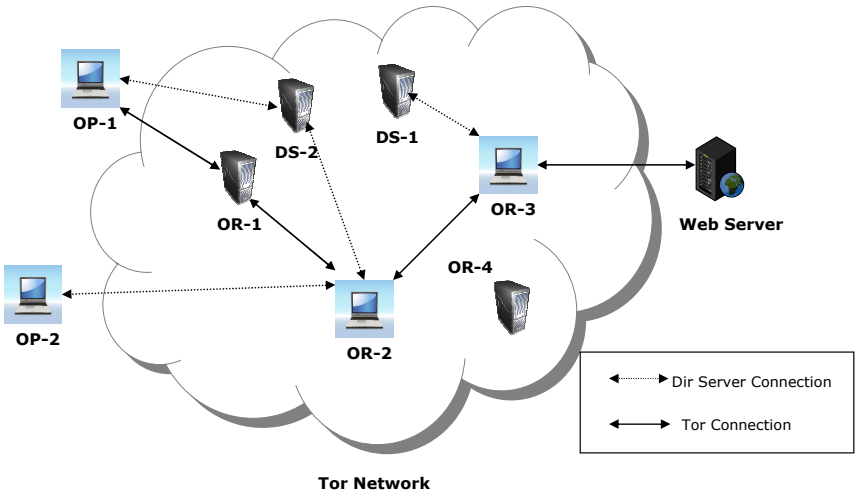

Figure 2.1 Basic Tor Architecture

Figure 2.1 depicts the basic Tor architecture. When a user joins the network for the first time, it contacts the directory servers and download the network status and router lists. Later if user wants to tunnel data through Tor network, it chooses appropriate routers from the router lists and initiate TLS [17, 18, 19] handshake with the first onion router. OP builds circuits incrementally, one hop at a time, while negotiating session key with each hop. For example, in Figure 2.1 OP-1 will setup TLS connection with OR-1 and sends *CREATE* cell with a payload containing first half of Diffie-Hellman handshake encrypted with OR-1's public key [1]. This payload is called 'onion skin'. Once OR-1 receives *CREATE* cell it does necessary housekeeping and sends back *CREATED* cell with second half of DH handshake key if it decides to accept the connection. Now, to extend its circuit to OR-2, OP-1 sends *EXTEND* relay cell to OR-1 with payload of new onion skin ecrypted with OR-2's public key. OR-1 in turn sends *CREATE* cell to OR-2 with the received onion skin as its payload. As OR-1 cannot read payload without knowing OR-2's private key, it cannot gain access to embedded key to be shared between OP-1

[1] Actually Tor uses some hybrid encryption mechanism, but for simplicity we will assume a public key cryptography

and OR-2. Similarly, OR-2 receives the *CREATE* cell from OR-1 and completes DH handshake without knowing that it has actually originated in OP-1. It sends *CREATED* cell back to OR-1 which then packs its payload into *EXTENDED* relay cell and send back to OP-1. OP-1 can now extend its circuit to OR-3 and finally to its destination. Each router in the path only knows its predecessor and successor and has no other information to link initiator with receiver.

Figure 2.2 shows a typical cell that OP uses to send data to its final destination (here a web request to web server). First, it'll encrypt the message with session key shared only with OR-3, then with OR-2 session key and finally with key shared with OR-1. As the cell passes through the circuit, each hop peels the upper layer and passes the cell forward until it receives the web server. Similarly, a reply from web server follows the same circuit back and is encrypted in each hop with corresponding symmetric keys which can then be decrypted only by OP-1.

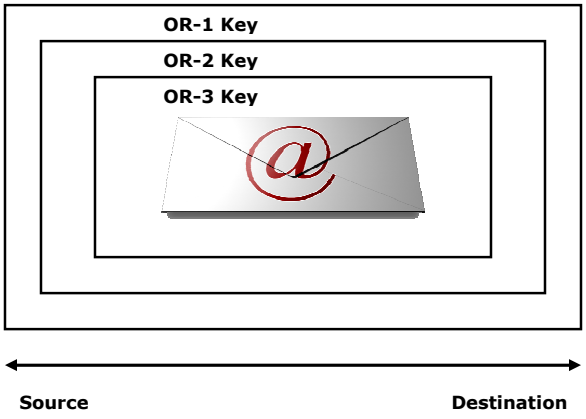

Figure 2.2 An Onion

Tor maintains both symmetric and asymmetric keys. While asymmetric keys are used to sign TLS certificates, router descriptors (by OR) and directories (by DS) and during connection setup to negotiate ephemeral keys, symmetric keys are used for layered encryption/decryption of relay cells. Later on we will investigate some issues related to use of symmetric key in our design.

As of now, Tor provides anonymity to TCP-based applications like web-browsing, instant-messaging, bittorrent, FTP, secure shell etc only. Even though TOR provides low-latency anonymity service, there are some applications like Voice over IP and multimedia stream that demand real-time performance and hence cannot be supported by Tor running on TCP. For this purpose we need a support of unreliable but fast UDP protocol and that is a basis of our work.

2.2.2 Tor Cell format

Cell is a basic unit of communication in Tor. Tor uses fixed-size cell of 512 bytes and has following fields:

Field	*Size*
CircID	[2 bytes]
Command	[1 bytes]
Payload	[PAYLOAD_LEN bytes]

The *CircID* denotes the circuit to which the corresponding cell belongs to. The *Command* field may hold one of the following values:

Command	*Value*	*Purpose*
PADDING	0	Padding
CREATE	1	Create a circuit
CREATED	2	Acknowledge create
RELAY	3	End-to-end data
DESTROY	4	Stop using circuit
CREATE_FAST	5	Create a circuit, no PK
CREATED_FAST	6	Acknowledge create fast

Depending upon *Command* field, payload may contain different information. Tor uses *RELAY* packets to tunnel end-to-end commands across the circuit. When the *Command* is of type *RELAY* the payload of unecrypted cell consists of:

Field	*Size*
Relay Command	[1 byte]
Recognized	[2 bytes]
StreamID	[2 bytes]
Digest	[4 bytes]
Length	[2 bytes]
Data	[Remaining bytes]

Out of these, *Digest* is of particular interest to us. This field is used by Tor for end-to-end inegrity checking. It contains the first four bytes of running SHA-1 digest of all the bytes that have been sent to this hop of the ciruit or originated from this hop of the circuit, including the entire payload (with *Digest* field set to 0) of the *RELAY* cell [38]. It does not include any bytes from relay cells that do not originate or end at this hop of the cirucit. We will come back to it when we discuss problem with end-to-end inegrity checking while using UDP.

The relay commands are:

Command	*Value*
RELAY_BEGIN	1
RELAY_DATA	2
RELAY_END	3
RELAY_CONNECTED	4
RELAY_SENDME	5
RELAY_EXTEND	6
RELAY_EXTENDED	7
RELAY_TRUNCATE	8
RELAY_TRUNCATED	9
RELAY_DROP	10
RELAY_RESOLVE	11
RELAY_RESOLVED	12
RELAY_BEGIN_DIR	13

2.2.3 Supporting Packages

We will now briefly describe few supporting pillars of Tor software that need to be considered while designing UDP Tor.

2.2.3.1 Libevent

Libevent /citelibevent, as its name suggests, is a library that provides set of APIs to handle asynchronous events in an application. It invokes a user-registered callback function whenever an event occurs on a specified descriptor or when event times out. Libevent provides support for both multi-threaded and multi-platform application and facilitates user to dynamically add or remove the event notification once the required

structure is initialized. Tor uses libevent to handle read/write events on its different sockets. It was developed by Niels Provos and has been released under 3-clause BSD license.

2.2.3.2 OpenSSL

OpenSsl /citeopensslweb is one of the most widely used and freely available implementation of SSL and TLS protocol. It is a multi-platform open source library based on SSLeay library developed by Eric A. Young and Tim J. Hudson. Apart from SSL/TLS implementation, OpenSSL also provides a robust general-purpose cryptographic library and utility functions. The first version of OpenSSL - 0.9.1c was released on 23rd December 1998 and with version 0.9.8, support for Datagram Transport Layer Security (DTLS) has also been added. Tor uses SSL/TLS and provided cryptographic algorithms for link authentication and encryption in various levels.

2.2.3.3 SOCKS

SOCKS [16] is a protocol, originally developed by David Koblas, that allows the clients transparent access to servers across the network firewall. As it works in Session Layer of OSI model, it is independent of application layer protocol and hence can be useful to various client-server applications like HTTP, FTP, telnet etc. When a client inside/outside a firewall needs to connect with a server across the firewall, it sends its request to SOCKS proxy server. The SOCKS server, after necessary authentication and eligibility check, will relay the request to/from appropriate server. Tor supports both the versions of SOCKS - SOCKS 4/4a and SOCKS 5 [20]. Apart from more choices of authentication, SOCKS 5 extends SOCKS 4a by providing a support for UDP. However, currently Tor does not support UDP connection through SOCKS. User application can

only connect with Tor through SOCKS protocol and may have to use some external proxy server like Privoxy [21].

2.3 Datagram Transport Layer Security: TLS with D!

Each OP/OR has a TLS connection with other OR. Tor uses TLS for mutual authentication as well as key negotiation. Once the connection is established, the cells are embedded in TLS records and sent across the Tor network. As TLS functionality depends upon underlying reliable transport layer protocol, we cannot use TLS to secure our UDP datagrams. In search of comparable protocol for unreliable datagram traffic, Modadugu and Rescorla have proposed new protocol called Datagram Transport Layer Security (DTLS) [34, 35]. DTLS has been designed as close to TLS as possible providing similar security guarantees to datagram-based applications. It has also been incorporated in OpenSSL library from 0.9.8 version and provides similar well-known interface as TLS. However, as can be expected from any datagram protocol, unlike TLS, DTLS does not provide reliable, in-order and guaranteed delivery.

2.4 PlanetLab

PlanetLab [22] is a distributed overlay network serving as a test-bed and deployment platform for large scale network services. Started with initial deployment of 100 nodes at 42 sites as a joint effort of few Universities and industrial research labs, PlanetLab now boasts more than 900 nodes distributed all over the world. It is managed by a PlanetLab Consortium, a group of academic, government and industrial institutions, which provides a support and development of PlanetLab hardware and software infrastructure as well as defining a policy guideline. PlanetLab provides researchers appropriate platform to run their experiments at large scale under real-world condition

spanning multiple administrative domains. As it is just an overlay network, data sent across PlanetLab nodes encounter same network characteristics - delay, latency, congestion, jitter - as real Internet. This gives researchers more concrete result from his experiment than possible through network simulation tool only. Apart from serving as an overlay network test-bed, PlanetLab also servers as a platform to deploy long-running services.

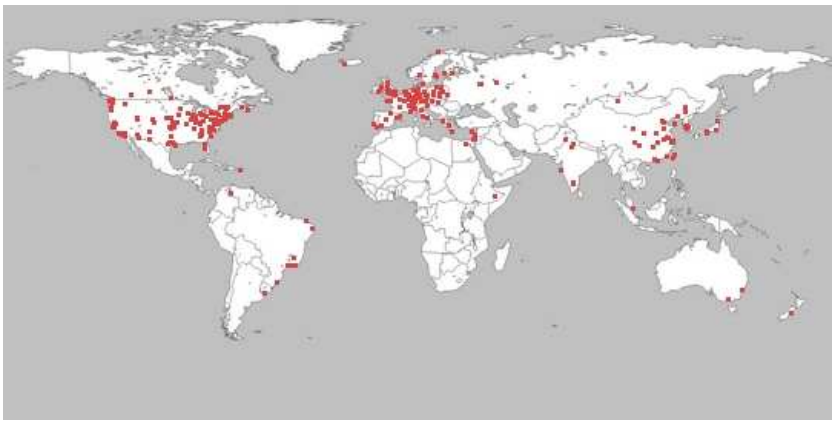

Figure 2.3 Distribution of nodes in PlanetLab network [22]

Any user seeking to use PlanetLab for a research must first create an account through his home institution, provided the institution is already a member of PlanetLab Consortium. Each account holder is provided a slice where user can choose any nodes distributed all over the world and add them to the slice. Any process running within a slice is logically isolated from other users processes which may be running concurrently in the same node. Virtual Machines (VM) in each node monitors and limits resources like CPU, bandwidth etc. used by each slice. User accesses the PlanetLab nodes using SSH.

Several services like network monitoring, network measurement and infrastructure services run continuously on PlaentLab. These services and tools can be of great assistance to a user resulting in a better nodes management and convenient experiment deployment. We would like to briefly describe two of the services that have been pretty helpful during our research.

2.4.1 CoMon

CoMon [23] is a PlanetLab infrastructure monitoring tool and is a part of Content Distribution Network test-bed (CoDeeN) [24] built on top of PlanetLab by researchers at Princeton University. It provides a mean to monitor the statistics of PlanetLab at both the node and slice levels. Node level view will show the name, address and location of each node, its current response time, its status, number of slices in that node, current CPU and memory usage and host of other information. Similarly, slice-centric view will allow user to get statistics about each slice through its name, like total number of bytes received/transmitted, number of processes currently running, percentage of CPU and memory used, total number of nodes in each slice. Information can either be viewed by visiting CoMon website or better through a dynamic query that display results based on user-specified criteria. C-style expressions used in CoMon queries facilitate the use of scripting language to automate the process and retrieve the data.

2.4.2 Stork

Stork [25] is a software deployment and installation utility maintained by researchers at University of Arizona. It is similar to yum and apt and provides efficient package management for PlanetLab users as well as private home users. Stork has a central repository to host tarball, RPM and other software packages. Instead of replicating multiple copies of same package in different slices of a single node, Stork allows user to

share the packages securely and efficiently among slices. It provides easy-to-use Stork Slice Manager that facilitate user to browse current repository, add new custom packages and install, upgrade or remove them to/from a node or group of nodes.

2.5 Voice over IP (VoIP)

Voice over IP can simply be defined as a process of transmitting voice over the Internet or any packet-switched network. VoIP calls are made using softphones which can either be stand alone IP phones like Cisco 7900 series or VoIP service clients like Skype, Google Talk, Vonage etc. Most of the VoIP services are based on IETF's Session Initiation Protocol (SIP) [26]. SIP is an application-layer protocol that is primarily used for establishing, modifying and terminating multimedia communication sessions. International Telecommunication Union-Telecommunication (ITU-T)'s H.323 [27] is another competing standard that governs the protocol to address multimedia communication over packet-switched networks. Both SIP and H.323 protocols follow client-server architecture where various network elements (like Proxy Server, Registrar, Redirect Server, Gateways in case of SIP and Gatekeepers in H.323) are responsible for user registration, end-point location, authentication, call management and termination. Apart from these standards, there are other VoIP services that use their own proprietary system. Skype [28] is one of the most widely used peer-to-peer proprietary VoIP services. Even though Skype handles user regsitration and authentication centrally, all other operations are managed in completely decentralized and distributed fashion [29].

Irrespective of network architecture, all VoIP system has to follow certain steps in order to transmit voice over the packet-switched network. In their technical report, Mehta and Udani [30] have befittingly categorized VoIP into following four components:

- *Signaling*: As in normal telecommunication network, signaling deals with control of connection setup and management. It assists user to locate and inform the end-point about incoming call, establish and initiate the connection and finally terminate the session.
- *Voice Codec*: We need to convert analog voice signal to compressed digital version in order to transmit over the network. There are different codecs available to perform this task with different characterics - bandwidth usage, sound quality, computation requirement, silence compression, intellectual property etc [30]. G.711, G.726, G.723.1 and G.729AB are ITU specified narrowband codecs where as G.722 and G.722.2 are ITU wideband codecs. Wideband codecs (16KHz sampling rate) provide better sound quality than narrowband (8KHz sampling rate) but at the expense of higher bit rates and more processing power. Apart from that, we would also like to mention about Speex [31] which is an open source speech codec that is specifially designed for VoIP application. Currently open source VoIP services like LinPhone, Ekiga and Asterisk are using Speex as voice codec. We are using simple application based on Speex codec to test our UDP Tor design.
- *Transport*: Once the voice is encoded, it needs to be framed in appropriate format to transmit it over the Internet. Voice packets are typically transported using Real-time Transport Protocol (RTP) /citertprfc, which is generally run on top of UDP. RTP provides end-to-end transport service for real-time data but does not guarantee in-order delivery nor ensures any quality of service [30].
- *Gateway Control*: Gateway control deals with inter-operability of different VoIP services or between Internet and PSTN network.

The possible reasons for degradation of Quality of Serivce (QoS) in VoIP are latency, jitter, packet loss, low bandwidth, and possibly added security measures. According to the ITU-T G.114 [32] recommendations, an upper bound on one-way latency is 150 ms for

acceptable voice quality. A delay of 150-400 ms can be deemed tolerable for international calls, and delay above 400 ms is unacceptable. Voice digitization takes around 1-30 ms, depending upon the coding scheme and quality of reproduced signal [33], further reducing the time available for packetization and routing through the Internet. Similarly, packet loss has a huge impact on QoS of VoIP network. For VoIP traffic encoded with G.711 (an international standard for encoding telephone audio on 64 kbps stream), a packet loss of 5% can cause the QoS of VoIP to drop below that of PSTN , even if the latency is bounded within 150 ms [33]. For VoIP over anonymous channels, we assume that users would be willing to suffer some performance loss, such as could be expected from an international call.

CHAPTER 3

SYSTEM DESIGN

Tor is specifically designed to provide anonymity to TCP-based applications only. So the whole Tor framework has its foundation built on and around the inherent strength of TCP - connection-oriented, reliable and in-order packets delivery. However, due to this very nature of TCP, it induces long delays which may not be suitable for real-time applications such as VoIP. We have to rely on unreliable but fast and efficient UDP and hence have based our Tor design on UDP protocol. In this chapter, we will discuss about our system design of UDP Tor. We have tried to make as few changes as possible in original Tor desgin, but due to unreliable nature of UDP there arise few complications and issues that we have to deal with. We will briefly describe each component of original Tor, challenges we face on introducing UDP and our solution or walkaround to those issues.

3.1 Unreliability

Tor operation and management relies on the reliability provided by the underlying TCP protocol. For multimedia applications, we no longer need the in-order and guaranteed delivery provided by TCP. However, Tor still requires these features for circuit creation and management. For example, when OR wants to set up a connection, it sends *CREATE* request to an appropriate hop and waits for *CREATED* response. Unless it receives the *CREATED* response back it does not proceed forward. As UDP does not provide guaranteed packet delivery, these requests/responses might get lost on the

way. Hence we need to introduce some kind of timeout and retransmission mechanism to ensure reliability in unreliable UDP protocol.

3.1.1 Event Notification, Timeout and Retransmission

Tor uses Libevent to handle asynchronous events. Whenever an event occurs, a user registered call-back function is invoked. Currently Tor registers events through Libevent API without any timeout specified, i.e. a callback function is called only when the matching event occurs on listening socket. So in order to implement a timeout and retransmission mechanism, we first need to enable timeout on an expected event.

Once we send a cell to the other hop, we buffer it and set a retransmission timer (only if we are expecting a response). If the timer expires before we receive a valid response, we reset the timer and transmit the buffered cell again. However, we need to limit the number of times we retransmit the same request. So if we transcend a pre-determined retransmission limit, we need to take necessary action (like close the circuit). When the receiver receives a response it initializes/updates necessary data structure and performs requested operation, prepares a response cell and buffer it before sending it. It however does not set retransmission timer.If the receiver receives the same request again, it retransmits the buffered response rather than repeating the same task. The number of retransmission is again limited by a pre-determined limit.

Choosing an appropriate initial retransmission timeout (RTO) value is little tricky. Even though it may be desirable to calculate RTO based on round trip time (RTT) estimate as described in [37], it introduces unnecessary complexity. Hence we are keeping the things simple by using pre-determined initial value. We will then use *exponential backoff* mechanism to calculate subsequent RTO if the timer expires. For example, if our

first timeout is 1 sec, we will set next timeout to be 2 sec. If we don't get reply within that time, we set the timeout to 4 sec and so on.

3.2 Relay Encryption/Decryption

Tor uses 128 bits Advanced Encryption Standard (AES) in counter mode (AES/CTR) [39] to perform layered encryption/decryption of relay payload. As can bee seen from the figure 3.1 (referenced from [**?**]) in AES/CTR mode encryption and decryption are the same operation. It uses successive counter blocks along with initialization vector (IV) to generate keystream which is then XORed with plaintext to form ciphertext (Tor uses IV of all 0 bytes). In order to decrypt the ciphertext, decrypter needs to have the same counter block. As TCP guarantees reliable and in-order delivery, Tor keeps this counter implicitly by counting each cell sent or received. However, with UDP we may need to have some mechanism to share counter block between sender and receiver as we will not be able to correctly decrypt record N+1 without receiving record N. For this purpose we use explicit sequence number in each cell that will allow us to decrypt the cells independent of one another.

With addition of sequence number, each cell now has the following fields:

Field	*Size*
CircID	[2 bytes]
Command	[1 bytes]
Sequence Number	[2 bytes]
Payload	[507 bytes]

So now when OR receives a *RELAY* cell, it uses a sequence number to synchronize ciphertext and keystream. The sequence number is set by originator of the stream. The first sequence number is generated randomly and then incremented by one for each cell.

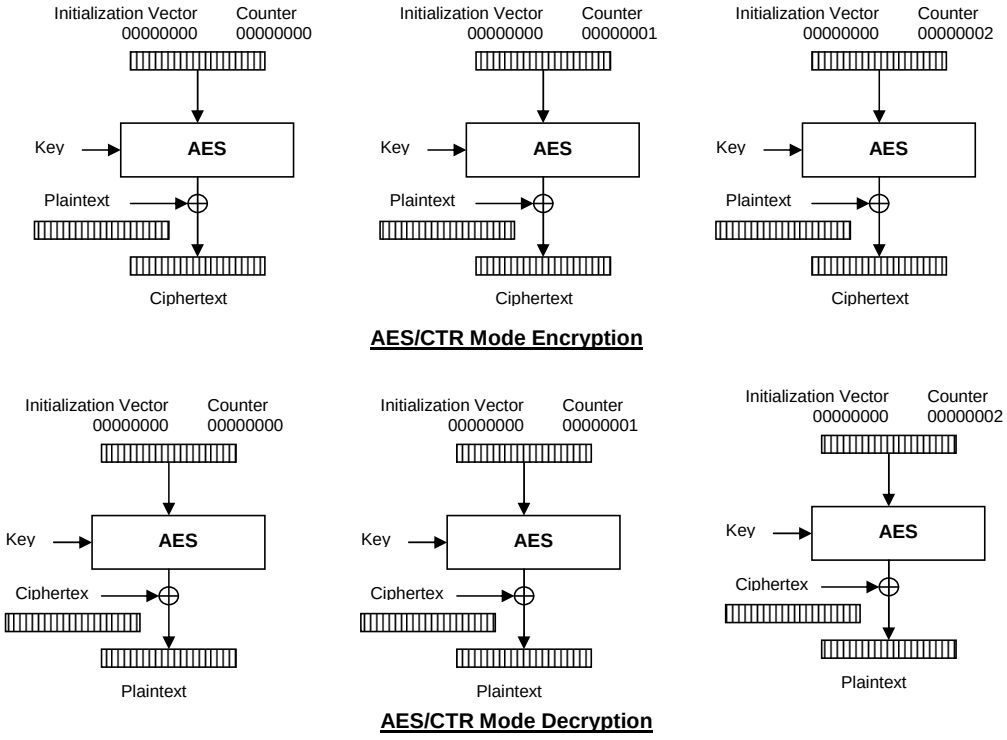

Figure 3.1 AES/CTR Mode Encryption/Decryption

Each sequence number is of 2 bytes and if it transcends the boundary or rolls over, we start from 0.

When an OR receives a cell with sequence number older than last one, it needs to identify whether it is a duplicate cell or out-of-order delivery. For this purpose, each OR keeps a receive-window of CIRCUIT_MAX_OLD_CELL_WINDOW size (configurable parameter) for each circuit. It also saves the sequence number of last cell received in the circuit. As shown in figure 3.2 each window is a bit vector with bits mapping to sequence number received earlier. When a new cell is received, a decision has to be made whether to drop the cell (for example, a duplicate packet) or process it. Each received cell's sequence number is mapped to its corresponding bit in receive-window. If the bit is set or if the sequence number of new cell is same as the last sequence number received, it is

a duplicate cell and necessary action needs to be taken. If the sequence number maps to a number beyond the window size, the cell is silently dropped. Out-of-order cell is processed if its sequence number maps within the receive-window. 1 shows an algorithm to check the validity of received cell based on sequence number.

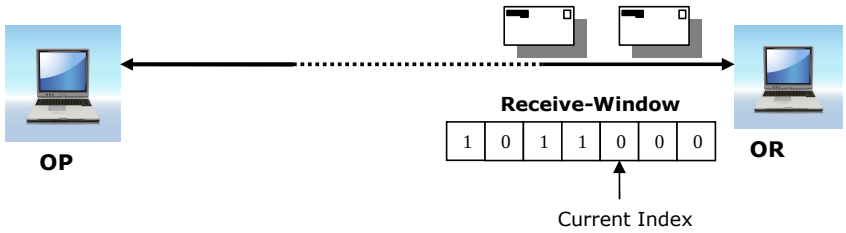

Figure 3.2 Window maintained by OR to keep track of old cells

Input: seq number of new cell(*new_seq*), seq number of last cell(*old_seq*),
receive-window(*recv_win*)

Output: Cell Validity (true/false)

$diff$ = *new_seq* - *old_seq*;

if $diff$ = *0* **then**
| Return false;

end

if $diff > 0$ **then**
| **if** *Full(recv_win)* **then**
| | ShiftRight(*recv_win*, $diff$);
| **end**
| **else**
| | Increment(*current_win_index*, $diff$);
| **end**
| Mark(*recv_win*, *current_win_index*);
| Increment(*aes_ctr*, $diff$);
| Return *true*;

end

else
| *recv_win_indx* = Map(*current_win_index*, $diff$); ;
| **if** *recv_win_indx* > $CIRCUIT\_MAX\_OLD\_CELL\_WINDOW$ **then**
| | Return $false$;
| **end**
| **else**
| | Mark(*recv_win*, $diff$);
| | Decrement(*aes_ctr*, $diff$);
| | Return *true*;
| **end**

end

Algorithm 1: Algorithm to check validity of newly received cell.

3.3 End-to-end integrity checking

As mentioned earlier, Tor provides a mechanism to check end-to-end itegrity. When OR sets up a connection with any hop, each of them also initialize SHA-1 digest with derivate of key negotiated between them. Then as they transmit the relay cells, they incrementally update the SHA-1 digest with the contents of all relay cells and include first four bytes of the current digest in the *Digest* field of each relay cell. They also keep a SHA-1 digest of data received to verify the received hashes [4].

Again due to unreliable nature of UDP, we might face discrepancies in digest sent and digest calculated if we even have a loss of single packet. Hence it is not feasible to keep a running digest of all the cells sent or received to verify end-to-end integrity. So we will set the *Digest* field to the SHA-1 digest of only the current *RELAY* cell payload (with the digest field set to zero). With a blanket of security provided by underlying DTLS protocol and given the fact that SHA-1 is seeded with derivate of keys only the two end-points know, we get acceptable per-cell integrity check with this method. However we admit it is not as strong as original Tor version and search for better way to handle this situation has been postponed for future work.

3.4 Application Interface

The only way user application can speak with Tor is through SOCKS protocol [40]. Application establishes TCP connection to SOCKS server (generally via SOCKS proxy like Privoxy, Dante, SOcat) running in OP requesting for a connection to remote address and port. OP will then attempt to create a Tor circuit to client's requested address and returns a result (success or failure) to the client. Subsequent communication then takes

place using normal TCP stream. Figure 3.3 depicts a typical connection setup between user application like web browser with OP using SOCKS proxy like Privoxy.

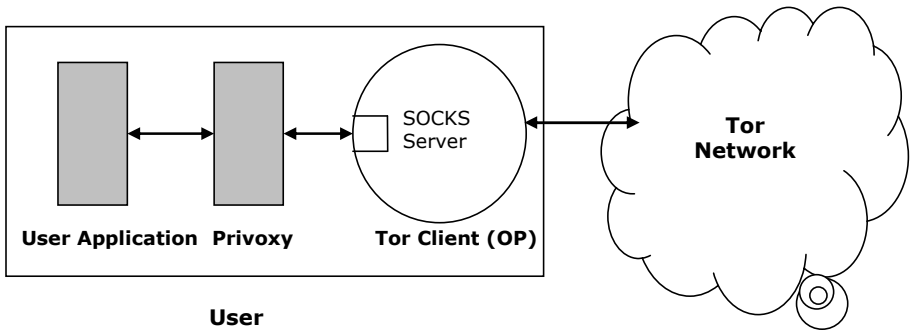

Figure 3.3 Relation between User Application, OP and Tor Network

Tor supports all the versions of SOCKS protocol with few of its own extensions. SOCKS 5 has a provision to handle UDP datagrams using *UDP_ASSOCIATE* command [20]. When a client wants to connect with remote server using UDP, it sends a UDP_ASSOCIATE request to SOCKS server (via TCP) with address and port of a destination. The SOCKS server, after evaluating the request, will establish UDP connection with requested server and sends back new address and port where client should send UDP datagrams in order to be relayed. If the original TCP connection with SOCKS server terminates, associated UDP connection also terminates.

Tor currently does not support UDP_ASSOCIATE request. We are also currently deferring it as a future work because handling two separate streams (TCP and UDP) for same connection may add some complication which we want to avoid at this phase of our design. Instead we have implemented custom UDP version of original TCP based SOCKS mechanism. A client asks SOCKS server (listening on UDP port instead of TCP) to establish a connection with destination by sending a *CONNECT* request. SOCKS

server will then create/select a Tor circuit to an appropriate exit node and send *RELAY_BEGIN* cell requesting to create a UDP connection with destination. If it receives a *RELAY_CONNECTED* cell back, it sends success to client otherwise failure with appropriate error code. Our request/response message has following format which is similar to SOCKS 5 request with IPv4 address type:

Field	*Size*
Version	[1 byte]
Reserved	[1 byte]
Address Type	[1 byte]
Address	[4 bytes]
Port	[4 bytes]

Here, we are using 5 as *Version* and IPv4 (value:1) as *Address Type*. In case of CONNECT request, Address and port contain the IP-address and port number of destination where as response contains address and port of SOCKS server to which client should send UDP datagrams.

3.5 Pre-connection setup

The way multiple clients are handled in UDP and TCP protocols are little different. TCP is a connection-oriented protocol where each client has to establish separate connection with a server through 3-way handshake before sending data where as UDP client can immediately start sending packets once socket is created. As stated earlier OR generally listens on its OR port for any incoming connection request. When it receives a connection request from a client, it creates a new socket (and corresponding data structure for new connection) and handles subsequent connection with client through that

socket. However, UDP server receives packets from multiple clients through the same socket and hence has to rely on remote address returned from *recvfrom* function call to identify which client has sent that particular packet. This thing gets little complicated with an introduction of DTLS in the picture. DTLS, as implemented in OpenSSL, reads a packet from the socket using *SSL_read* (similar to TLS). So we do not have any means to identify the sender as *SSL_read* does not return address of the sender. For example, in figure 3.4, OR-2 receives packet from both OR-1 and Bob. Without any means to check the network address of sender of recently received packet, we cannot process the packet further. As per our understanding current DTLS implementation in OpenSSL does not seem to provide any mechanism to handle multiple DTLS clients (at least we are assuming so as there is clearly a lack of proper documentation on OpenSSL DTLS implementation).

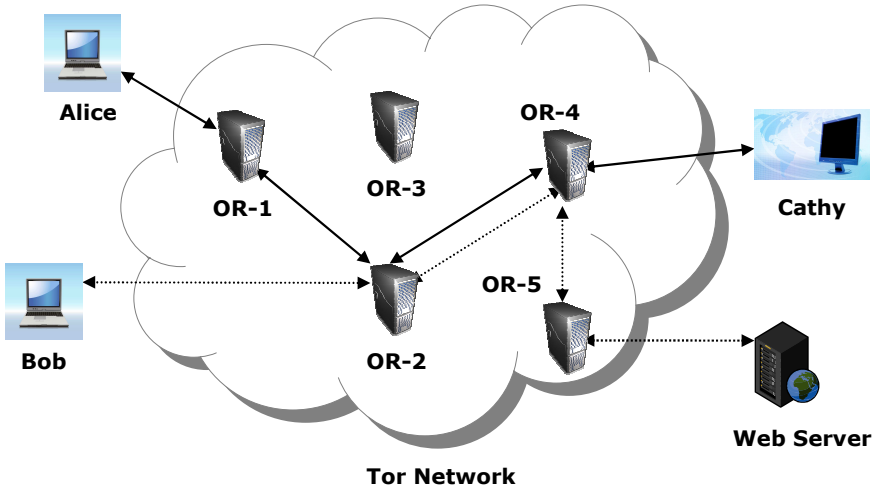

Figure 3.4 Tor network with multiple connection to same OR

In order to handle multiple clients with DTLS/UDP protocol, we are going to imitate the connection mechanism of Trivial File Transfer Protocol (TFTP) [41]. TFTP is

a simple watered-down version of FTP based on UDP. It is mostly used when speed and size are of more concern than data integrity. In TFTP, an initiator sends a *RRQ* (Read Request) or *WRQ* (Write Request) to a TFTP server listening on a well-known port 69. The server then replies back with appropriate response sent through newly allocated ephemeral port. Subsequent connection with the client takes place through that port while it listens for new connections on its original port.

Similarly, when OP/OR wants to setup a connection, it sends a request to a listening UDP socket of next hop. The receiving OR then checks whether the connection has already been established for that OP/OR or not. If not, it creates a new socket with new ephemeral port. It then initializes the necessary data structure to handle the connection, sends an acknowledgment back to the sender through the new port and keeps on listening on the original OR port for new connection. The client, when it receives an acknowledgment, will update its address-book with new address and starts DTLS handshake with new Server socket. This two-way handshake has been shown in the figure 3.5

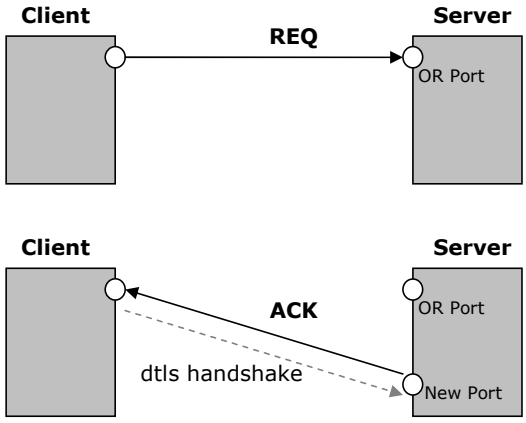

Figure 3.5 Two-way handhsake during pre-connection setup

Currently the format of request-response message is:

Field	*Size*
OpCode	[1 byte]
Sequence Number	[2 bytes]
Server Port	[2 bytes]
Reserved	[2 bytes]

The *OpCode* field can be any of the following:

OpCode	*Value*
CONN_REQUEST	0
CONN_ACK	1
CONN_ERROR	2

Sequence number is a 2 bytes randomly generated number that is used to identify duplicate request/acknowledgment. Server Port field is set to null in CONN_REQUEST where as it holds newly allocated ephemeral port number of server in CONN_ACK message.

Due to unreliable nature of UDP communication, we also need to set timeout and retransmission mechanism as discussed earlier. Figure 3.6 shows different possibilities that can occur in our two-way handshake protocol (three-way if we consider client's DTLS handshake message as an acknowledgment to server's ACK). Case (a) depicts a typical successful handshake. In case (b) request is lost; after retransmission timer times out, client will again send a REQ message. Case (c) shows a scenario where a timeout occurs even before client receives a server's ACK message. So a server will get a duplicate request. In that case server will just send another ACK message (in case the first ACK message might have been lost). Client will ignore the second ACK message (if it received

the first one) as it might have already started DTLS handshake with server. Figure (d) shows similar case when server's ACK message is lost. Client will send another request after its RTO times out.

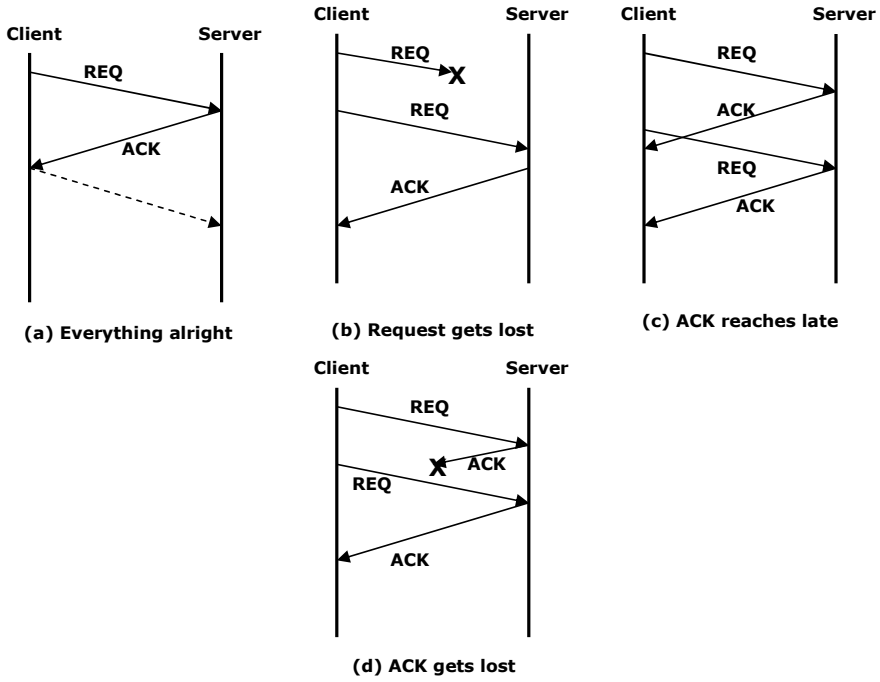

Figure 3.6 Different retransmission scenarios in 2-way handshake

Adding extra steps to set up a connection using two-way handshake will surely introduce some delay in a process. However this delay occurs during connection setup and not while sending data when any delay is of utmost importance. Generally Tor always creates a number of circuits in advance and hence user is not exposed to connection delay most of the time. And also since we are not exposing anything more than that can be retrieved from packet headers, we believe we are not introducing any kind of vulnerability by introducing TFTP like mechanism to establish connection.

3.6 Multiplexed Circuits

Another issue that cropped up while designing our UDP Tor is related to circuit multiplexing. Instead of creating multiple connections to same OR for different circuits, Tor multiplexes all the circuits destined for single hop. As can be seen from figure 3.7, OP-1 has two circuits that connect it to remote servers. Both the circuits have OR-1 as a first hop and therefore have them multiplexed to OR-1. Similarly, two different circuits from different initiators are also multiplexed in a connection between OR-2 and OR-4.

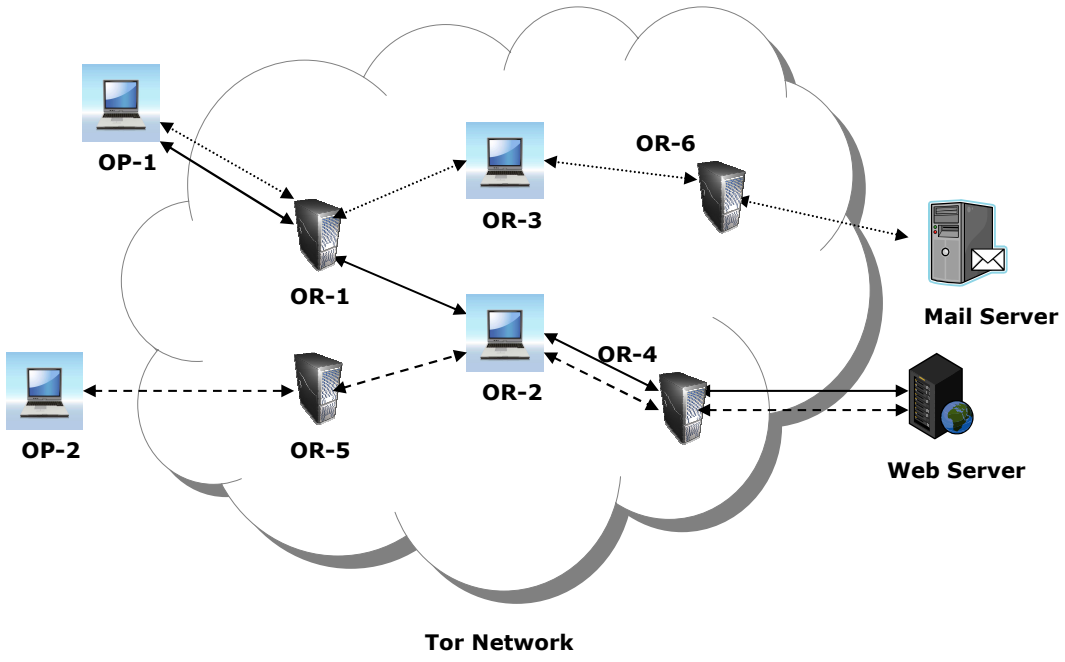

Figure 3.7 Circuits Multiplexing in Tor connections

The problem with multiplexing multiple circuits comes up while handling timeout on socket descriptor. As we know Tor running on TCP does not need to have timeout mechanism as it is handled internally by TCP protocol. But in case of UDP, we need to have explicit timeout and retransmission mechanism to guarantee connection setup

and circuit establishment. Whenever a timer times out a pre-registered event handler is invoked. However, if a connection has multiple circuits, how do we know that this timeout is related to which circuit? For example, in the figure above 3.7 if connection at OR-2 (connecting to OR-4) times out, we will not be able to relate this timeout with any of the two circuit with certainty. Even before that, as event handler is associated with each socket/connection, it is not possible to set different timer values for multiple circuits utilizing the same connection.

To solve this issue, we have added additional element to data structure maintained by Tor for each connection. Now each connection will hold a list of circuits currently multiplexed by it. There is a provision to dynamically add/remove the circuits from the list too. Apart from that, as each connection cannot have multiple timeout values, we have set the timer to go out every second. Each circuit will internally hold its timeout value and a flag to indicate whether it is expecting any response or not. So every second when the timer times out in any connection, we will traverse through the list of its multiplexed circuits and check:

- Is this circuit expecting a response?
- If yes, has it timed out?

If we have a timeout, we will handle retransmission as discussed earlier. Figure 3.8 shows a sample connection between two ORs with three multiplexed circuits. The flag *resp_expected* is true if we are expecting any response; for example when we have just sent *CREATE*, *CREATE_FAST*, *RELAY_BEGIN* cells and do not proceed further without getting related response. So every second, event handler for particular connection at OR-1 will check all three circuits for possible timeout (only if they are expecting any response). If the original circuit is torn down, it is also removed from the list.

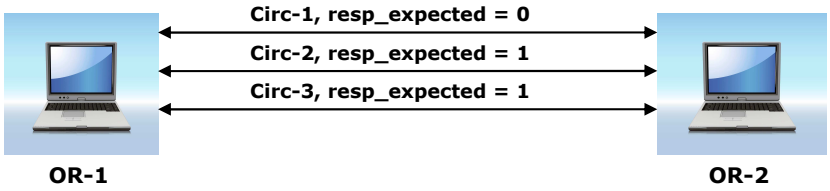

Figure 3.8 Handling timeout in multiplexed circuit

Setting up timer to time out every second definitely adds extra latency due to delays associated with invoking call-back function and traversing through the circuit-lists. However, currently we do not see any other efficient solution to handle this kind of problem. We will however try to optimize the present implementation or look for more efficient solution in our next phase.

3.7 What we have left untouched?

There are few parts of Tor which we have not modified in our design. Some of them are good as they are and some we have deferred as future work. Few of the things we have left untouched are:

- support for rendezvous points and hidden services.
- tunneling of Directory Service through Tor network.
- resolving address using *RELAY_RESOLVE* command through SOCKS proxy
- support of UDP datagram handling using SOCKS 5 protocol
- controlling Tor process using Tor Control protocol

Connection between directory server and OR/OP is still handled using TCP as we need a guaranteed and in-order delivery of network status report.

CHAPTER 4

EXPERIMENT SETUP

In this section we will describe our experiment setup to test UDP Tor. Figure 4.1 shows a diagram of our basic test setup. We have a test application that tries to setup a connection with OP using custom SOCKS protocol as discussed earlier. It is analogous to dialing a receiver's number requesting an operator to establish a connection. OP will then create a circuit to the destination and send an acknowledgment back. Once we receive *SUCCESS* from OP we will start reading data from raw audio file, encode them to compressed bitstream and transmit them frame-by-frame to the destination through UDP Tor network. At the receiver end we will read UDP packets, decode them and save them as a raw audio file.

Next we will examine individual components of our test module in little detail.

4.1 Private Tor on PlanetLab

To test our design, first and foremost, we need to set up a private Tor network. It is possible to arrange few systems and setup private Tor nodes locally. But as it will not be able to represent a true network characteristic of the Internet, we decided to run our experiment on PlanetLab testbed. We can select nodes globally from different geographic regions and transmit packets over the real Internet. To setup private Tor network we need to have our own authoritative directory servers, tor routers and clients. We also need to configure each of them through tor configuration file (*torrc*) so that clients and servers will connect with our private directory servers rather than default public ones.

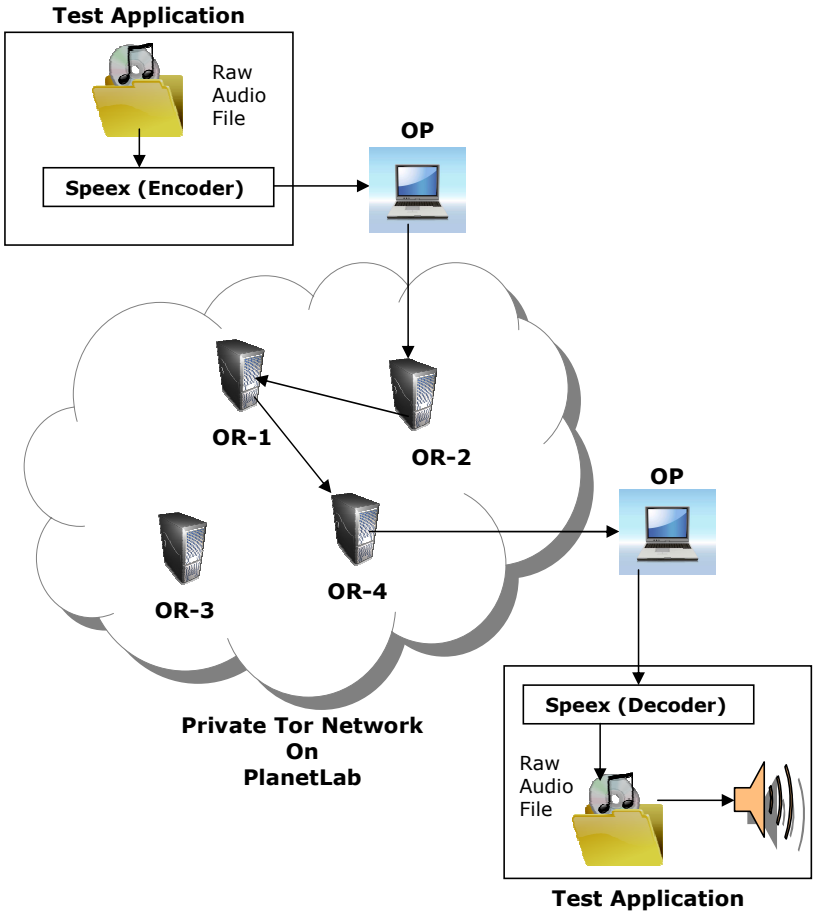

Figure 4.1 Our test setup with Private Tor deployed on PlanetLab

We have provided sample *torrc* configuration files for directory server, router and client in Appendix 1 A. As per recommendation, we need to have at least two directory servers and three routers to run a private Tor network.

We need to have a good and reliable nodes to run as the directory servers. Since some of the PlanetLab nodes tend to be erratic and unreliable, we need a way to sift through the selected nodes and choose the good ones. To solve this problem we have written a python script that tries to setup connection with each of the nodes in our slice

and separates good active nodes from inactive ones. After running the script for number of iterations we were able to collect list of few good and reliable nodes spread across the world. We also have scripts to automate the process of setting up and updating nodes of our Tor network. They will read nodes from the provided node-list, upload required packages, install dependent standard libraries and setup/run Tor nodes - directory servers, routers and clients (in that order). All the tor nodes are run as daemons with startup scripts set to handle automatic restart in case of system reboot.

4.2 Test Application

There are large number of VoIP phones available in the market today. [42] has an extensive listing of both hard and soft phones. However for our initial testing, we have decided to design our own custom application that will encode a voice sample into bitstream and send it over the network using UDP protocol. One of the main reason on doing so is due to non-standard interface we currently have to connect datagram-based application with Tor client. As discussed earlier, user application will now have to use our custom UDP SOCKS protocol to setup a connection with onion proxy. So writing a separate SOCKS proxy that mediates a connection between existing VoIP applications and our Tor client would be an extra work which we have deferred for now. Another reason for using our own test application is due to flexibility and convenience we get in modifying various test parameters like bit rate, sampling frequency, frame size of voice sample to be transmitted over the network. Unless we extend Tor to support UDP using SOCKS 5 protocol so that we can use existing SOCKS proxy, our simple test application would give us a flexibility and freedom to test UDP Tor with different codec parameters not possible with existing ready-to-eat VoIP applications.

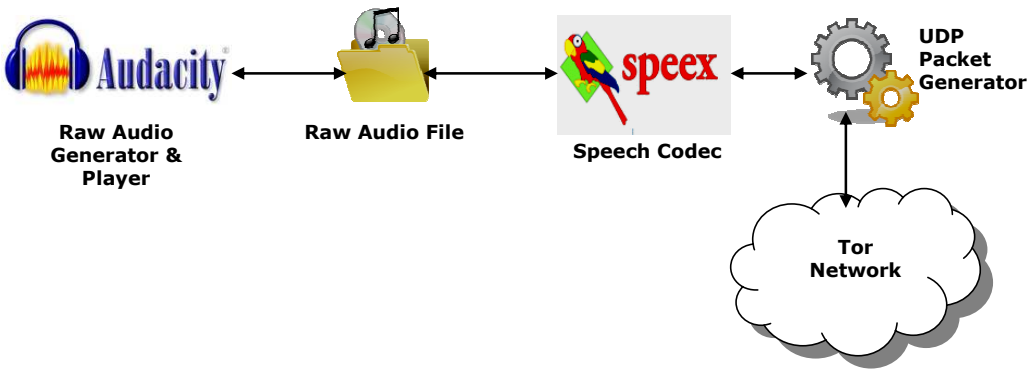

Figure 4.2 Internal components of test application

Figure 4.2 shows internal components of our test application. Let us zoom into them and briefly talk about each component.

4.2.1 Raw Voice Generator and Player

As of now, we have not implemented a mechanism to record and transmit a voice in real time as done by normal VoIP application. Even though it sure is a nice feature to have, we are sacrificing it in favor of simplicity as it demands a lot of extra time to be invested on designing (or searching) a A/D converter. Instead of that we are generating a raw voice file and feed it to a codec. For this, we are using an open source audio recorder and editor called Audacity [43]. We will record (or feed existing audio file in different format) to *Audacity* and get a 16-bit PCM raw audio file as an output which will then become an input to Voice encoder. Similarly on a receiving side, we will import decoded raw audio file as an output and play it in *Audacity* to check the quality of received voice sample. Audacity also has a provision to perform side by side comparison of two audio samples.

4.2.2 Speech Codec

Speech codec is used to compress a digital audio signal and generate compact bitstream. We are using *Speex* [31] as our speech codec. It is a free open-source audio compression format designed especially for voice applications. It is based on Code-excited Linear Prediction (CELP) [44] algorithm and supports narrowband (8KHz sampling rate), wideband (16KHz sampling rate) and ultra-wideband (32KHz sampling rate) compression with bitrates ranging form 2-44Kbps. It has also provided *libspeex* API that can be used to design our own encoder and decoder with set of parameters specific to our need. For example, at encoder side, we can use 20 ms frame in narrowband mode (giving us frame-size of 160 samples) and quality set to 8 in Constant Bit Rate (equivalent to a bitrate of 15Kbps) giving us encoded audio bitstream of 37.5 bytes. Similarly we can vary the voice payload by experimenting with differnt modes (narrowband/wideband/ultra-wideband) with varying bitrates resulting in a different payload size. Currently we are encapsulating one frame in one UDP packet only. However we can increase the number of frames in each packet to get better performance. At decoder side, we will receive audio bitstream that we decode and save it as a raw audio file.

4.2.3 Packet Generator

Once we get a compressed bitstream from an encoder, we need to send it through the network. Most of the VoIP services use RTP to transmit voice packets over the network. RTP along with RTP Control Protocol (RTCP) facilitate applications to handle out-of-order delivered packets and supply information to implement some kind of flow control mechanism which UDP alone cannot provide. As we are not implementing a full-fledged VoIP application, we have kept the things simple and send encoded data as simple UDP packets.

4.3 Where are we now?

We have completed the design and implementation of the aVoIP. We have taken a stable Tor source code (*tor-0.2.1.16*) and built our design on top of it. Currently we are at the end of our debugging phase. We are now able to build a full circuit (three hops) and are working on debugging end-to-end connection. We have a fully functional python scripts to automate the task of setting up private Tor network on PlanetLab nodes. We have tested the scripts by setting up original private Tor network and performing web related activities. We have also fully tested our voice application locally with different audio samples and codec parameters.

4.3.1 Debugging Issues

Tor is a complex system and debugging has been quite challenging. We mention here two most significant issues encountered during debugging session - both related to DTLS. DTLS is comparatively newer addition to OpenSSL library and hence lacks a proper documentation and support. To solve these issues, we had to spend considerable time hacking though OpenSSL code apart from Tor.

- Tor uses sockets in non-blocking mode. Hence during DTLS handshake, a *read* on socket(internal to *SSL_accept/SSL_connect*) returns
  1. number of bytes read (success)
  2. no data to read
  3. timeout (timer maintained internally by DTLS during handshake)
  4. other errors

  As per OpenSSL documentation, we are supposed to receive separate return values for 2 and 3. But we were getting same error for DTLS handshake timeout and non-blocking return. Since the action to be taken is different in both the cases (closing

the connection on timeout and retry on non-blocking return) our connection was getting closed for no fault.

- *SSL_read* after DTLS handshake was crashing the system. We were able to read from the same socket using *recvfrom()* system call, but trying to use *SSL_read* crashed (inside *SSL_read*) without any error. After much debugging, we found that if we ignore few of the packets received just after handshake, *SSL_read* then works fine. So in absence of correct solution, we have employed a temporary hack. After DTLS handshake is complete, OR sends pre-determined *start_code* to the other OR and vice versa. Until OR receives a *start_code* it will read from the socket using *recvfrom()* and discard the received packet. After receiving a *start_code* it will then read packets embedded in DTLS frames using *SSL_read*. Due to unreliability of UDP, we need to add retransmission mechanism in this case also.

CHAPTER 5

CONCLUSION

With growing popularity, VoIP demands better user privacy. To provide anonymity to VoIP users, we may need to tunnel voice packets through anonymity system, which in turn will introduce extra delays. Hence supporting anonymous VoIP over the Internet might add some more challenges in terms of QoS. In order to study the feasibility of aVoIP over current Internet infrastructure, we seek to implement aVoIP. We choose Tor as a base architecture for our aVoIP system design. Tor has originally been designed to provide low-latency anonymity services to TCP based applications. In order to transmit voice packets over the packet switched network with acceptable QoS we need to rely on unreliable but fast and efficient UDP protocol. So the main objective of our work has been to understand the Tor in depth, analyze the complications raised in replacing TCP with UDP, find simple and efficient solution and implement it.

We took a stable version of Tor source code and tried to comprehend the Tor architecture in detail. One of the major issue that crops up while introducing UDP is related to unreliable nature of UDP protocol. We do not need guaranteed in-order delivery of voice packets but we need to provide that for Tor circuit establishment and management. We have implemented various features like timeout and retransmission with exponential backoff, replay-detection utilizing windows mechanism, explicit sequence number to handle layered encryption of relay cell using AES in CTR mode, pre-connection setup to support multiple clients in DTLS. We have also developed a testing environment to test our system design. We have written scripts to automate the setup of Private Tor Net-

work on PlanetLab nodes. We also have flexible test application to test our design with different codec parameters. With design and implementation of aVoIP system already complete and with simple, efficient and flexible testing environment in place, we have completed our first milestone towards an implementation of anonymous Voice-over-IP.

5.1 Future Work

We are currently at the end of our debugging phase. Hence our imminent goal is to run our system on PlanetLab successfully and test with our custom voice application. We then need a suitable VoIP application to test real-time performance. Most of the VoIP applications available today use SIP as a signalling protocol. Even though RTP-over-UDP is a protocol of choice for real-time application like VoIP, some VoIP applications may use TCP tunneling for NAT and firewall traversal. As we do not support TCP in our aVoIP at all (except for connection with Directory Servers), we need to make sure the VoIP application we choose do not rely on TCP for any signalling and call-handling. Other option would be to implement a simple VoIP application ourselves. With most of the VoIP components freely available as open source, we may only need to implement a simple user interface and integrate the other components together. Once we have all the setup ready, we like to run the aVoIP on PlanetLab as long-running service where people will be able to make calls. In this way we can also get a good measure of efficiency, reliability, quality and user satisfaction of our system design. We will also be releasing our source code under /emphopen source license.

APPENDIX A

SAMPLE TORRC CONFIGURATION FILES

torrc 1 Sample Tor Configuration File for Client

```
## Replace this with "SocksPort 0" if you plan to run Tor only as a
## server, and not make any local application connections yourself.
SocksPort 9050 # what port to open for local application connections
SocksListenAddress 127.0.0.1 # accept connections only from localhost

## Logs go to stdout at level "notice" unless redirected by something
## else, like one of the below lines.
## Send all messages of level 'notice' or higher to
##/usr/local/var/log/tor/notices.log
#Log notice file /usr/local/var/log/tor/notices.log
## Send every possible message to /usr/local/var/log/tor/debug.log
#Log debug file /usr/local/var/log/tor/debug.log
## Use the system log instead of Tor's logfiles
#Log notice syslog
## To send all messages to stderr:
#Log debug stderr
Log info file /usr/local/var/log/tor/tor.log

## Uncomment this to start the process in the background... or use
## --runasdaemon 1 on the command line.
RunAsDaemon 1

## The directory for keeping all the keys/etc. By default, we store
## things in $HOME/.tor on Unix, and in Application Data\tor on Windows.
DataDirectory /usr/local/var/lib/tor
PidFile /usr/local/var/run/tor/tor.pid

## The IP or FQDN of client.
Address planetlab2.uta.edu

# Run Tor as client only. Tor will under no circumstances run as a server.
ClientOnly 1
EnforceDistinctSubnets 0

## Option intended for setting up a private Tor network with its
## own directory authorities.
## List the authoritative directory servers
DirServer MyDirSer02 v1 128.42.6.144:9030 6B36 252F 210A 272E 2AB1 820E
DD9C 39F3 8ACA BD83
DirServer MyDirSer03 v1 143.215.128.194:9030 0DB1 30E1 4437 48F6 0C61 F274
E82F F22F 41EE AE5E
```

torrc 2 Sample Tor Configuration File for Onion Router

```
## Replace this with "SocksPort 0" if you plan to run Tor only as a
## server, and not make any local application connections yourself.
SocksPort 0
SocksListenAddress 127.0.0.1 # accept connections only from localhost

## Logs go to stdout at level "notice" unless redirected by something
## Send all messages of level 'notice' or higher to
#/usr/local/var/log/tor/notices.log
#Log notice file /usr/local/var/log/tor/notices.log
## Send every possible message to /usr/local/var/log/tor/debug.log
#Log debug file /usr/local/var/log/tor/debug.log
## Use the system log instead of Tor's logfiles
#Log notice syslog
## To send all messages to stderr:
#Log debug stderr
Log info file /usr/local/var/log/tor/tor.log

## Uncomment this to start the process in the background
RunAsDaemon 1

## The directory for keeping all the keys/etc.
DataDirectory /usr/local/var/lib/tor
PidFile /usr/local/var/run/tor/tor.pid

## Required: A unique handle for your server.
Nickname Ser128x252x19x21

## The IP or FQDN for your server. Leave commented out and Tor will guess.
Address vn2.cs.wustl.edu

## Contact info to be published in the directory
ContactInfo Bikas Raj Gurung <gbikas@hotmail.com>

## Required: what port to advertise for Tor connections.
ORPort 9001
EnforceDistinctSubnets 0
AssumeReachable 1

## List the authoritative directory servers
#DirServer nickname fingerprint
DirServer MyDirSer00 v1 129.107.35.131:9030 96DF 3FB9 40B3 A3AF 45C4 ED65
5A16 E1EE AA7C CEE8
DirServer MyDirSer01 v1 129.107.35.132:9030 CDE5 23DD A9D2 9802 CA62 3142
0474 A3E7 491F 8D90
```

torrc 3 Sample Tor Configuration File for Directory Server

```
## Replace this with "SocksPort 0" if you plan to run Tor only as a
## server, and not make any local application connections yourself.
SocksPort 0 # what port to open for local application connections
SocksListenAddress 127.0.0.1 # accept connections only from localhost

## Logs go to stdout at level "notice" unless redirected by sth else
## Send all messages of level 'notice' or higher to
## /usr/local/var/log/tor/notices.log
#Log notice file /usr/local/var/log/tor/notices.log
## Send every possible message to /usr/local/var/log/tor/debug.log
#Log debug file /usr/local/var/log/tor/debug.log
## Use the system log instead of Tor's logfiles
#Log notice syslog
Log info file /usr/local/var/log/tor/tor.log
## To send all messages to stderr:
#Log debug stderr

## Uncomment this to start the process in the background
RunAsDaemon 1
PidFile /usr/local/var/run/tor/tor.pid

## The directory for keeping all the keys/etc.
DataDirectory /usr/local/var/lib/tor
## Required: A unique handle for your server.
Nickname MyDirSer03

## The IP or FQDN for your server. Leave commented out and Tor will guess.
Address planet.cc.gt.atl.ga.us

## Contact info to be published in the directory
ContactInfo Bikas Raj Gurung <gbikas@hotmail.com>

## Required: what port to advertise for Tor connections.
ORPort 9001

## Uncomment this to mirror the directory for others.
DirPort 9030 # what port to advertise for directory connections
## Uncomment this if you run more than one Tor server
#MyFamily nickname1,nickname2,...

## bypasses the reachability detection and lets the network bootstrap
AssumeReachable 1
## become an authoritative Directory
AuthoritativeDirectory 1

#dummy DirServer fingerprint, so as to calculate correct fingerprint
#need to be replaced with actual directory server fingerprints
DirServer MyDirSer v1 127.0.0.1:9030 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000
```

REFERENCES

[1] "Anonymizer web site."

Available at http://www.anonymizer.com.

[2] "Lucent Personalized Web Assistant,"

Available at http://www.bell-labs.com/projects/lpwa.

[3] "FindNot,"

Available at http://www.findnot.com.

[4] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[5] O. Berthold, H. Federrath, and S. Köpsell, "Web MIXes: A system for anonymous and unobservable Internet access," in *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[6] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for Web Transactions," *ACM Transactions on Information and System Security*, vol. 1, no. 1, pp. 66–92, Nov 1998.

[7] A. Back, I. Goldberg, and A. Shostack, "Freedom 2.0 security issues and analysis," Zero-Knowledge Systems, Inc. white paper, Nov 2000.

[8] M. Freedman and R. Morris, "Tarzan: A peer-to-peer anonymizing network layer," in *Proc. ACM Conference on Computer and Communications Security*, Nov 2002.

[9] M. Rennhard and B. Plattner, "Practical anonymity for the masses with morphmix," in *Proc. Financial Cryptography (FC '04)*, February 2004.

[10] B. Levine and C. Shields, "Hordes: A protocol for anonymous communication over the Internet," in *ACM Journal of Computer Security*, vol. 10, no. 3, 2002, pp. 213–240.

[11] H. Federrath, "JAP: A Tool for Privacy in the Internet," http://anon.inf.tu-dresden.de/index_en.html.

[12] D. M. Goldschlag, M. G. Reed, and P. F. Syverson, "Hiding Routing Information," in *Proceedings of Information Hiding: First International Workshop*, R. Anderson, Ed. Springer-Verlag, LNCS 1174, May 1996, pp. 137–150.

[13] P. F. Syverson, D. M. Goldschlag, and M. G. Reed, "Anonymous connections and onion routing," in *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1997, p. 44.

[14] P. Syverson, M. Reed, and D. Goldschlag, "Onion Routing access configurations," in *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, vol. 1. IEEE CS Press, 2000, pp. 34–40.

[15] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr, "Towards an Analysis of Onion Routing Security," in *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, H. Federrath, Ed. Springer-Verlag, LNCS 2009, July 2000, pp. 96–114.

[16] D. Koblas and M. R. Koblas, "Socks," in *Unix Security III Symposium*, 1992.

[17] T. Dierks and C. Allen, "The tls protocol - version 1.0," in *IETF RFC 2246*, January 1999.

[18] T. Dierks and E. Rescorla, "The tls protocol - version 1.1," in *IETF RFC 4346*, April 2006.

[19] ——, "The tls protocol - version 1.2," in *IETF RFC 5246*, August 2008.

[20] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "Socks protocol version 5," in *IETF RFC 1928*, March 1996.

[21] "Privoxy." [Online]. Available: http://www.privoxy.org

[22] "Planetlab." [Online]. Available: http://www.planet-lab.org

[23] "Comon – a monitoring infrastructure for planetlab." [Online]. Available: http://comon.cs.princeton.edu

[24] "Codeen – a content distribution network for planetlab." [Online]. Available: http://codeen.cs.princeton.edu

[25] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman, "Stork: Package management for distributed vm environments," in *Proceedings of the 21st Large Installation System Administration Conference - LISA '07*, November 2007.

[26] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "Sip:session initiation protocol," in *IETF RFC 3261*, June 2002.

[27] "H.323: Packet-based multimedia communication systems,"
Available at http://www.itu.int/rec/T-REC-H.323-200606-I/en/.

[28] "Skype,"
Available at http://www.skype.com.

[29] S. A. Baset and H. Schulzrinne, ""An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol"," Columbia University, Technical Report CUCS–039–04, December 2004.

[30] P. Mehta and S. Udani, ""Overview of Voice over IP"," University of Pennsylvania, Technical Report MS-CIS-01-31, February 2001.

[31] "Speex: A Free Codec for Free Speech," http://www.speex.org.

[32] *One-way transmission time*, ITU-T Recommendation G.114, May 2003.

[33] D. R. Kuhn, T. J. Walsh, and S. Fries, *Security Considerations for Voice Over IP Systems : Recommendation of the National Institue of Standards and Technology*, 800th ed., National Institue of Standards and Technology, January 2005.

[34] N. Modadugu and E. Rescorla, "The design and implementation of datagram tls," in *Proceedings of NDSS 2004*, 2004.

[35] E. Rescorla and N. Modadugu, "Datagram transport layer security," in *IETF RFC 4347*, April 2006.

[36] R. Dingledine and N. Mathewson, "Tor control protocol,"
Available at https://svn.torproject.org/svn/tor/trunk/doc/spec/control-spec.txt.

[37] W. R. Stevens, *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI.* Prentice Hall, 1998, vol. 1.

[38] R. Dingledine and N. Mathewson, "Tor protocol specification,"
Available at https://svn.torproject.org/svn/tor/trunk/doc/spec/tor-spec.txt.

[39] R. Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)," in *IETF RFC 3686*, January 2004.

[40] R. Dingledine and N. Mathewson, "Tor's extensions to the socks protocol,"
Available at https://svn.torproject.org/svn/tor/trunk/doc/spec/socks-extensions.txt.

[41] K. Sollins, "The tftp protocol (revision 2)," in *RFC 1350*, July 1992.

[42] "VoIP Phones,"
Available at http://www.voip-info.org/wiki-VOIP+Phones.

[43] "Audacity: Free Audio Editor and Linker,"
Available at http://audacity.sourceforge.net/.

[44] M. R. Schroeder and B. S. Atal, "Code-excited linear prediction (celp): high-quality speech at very low bit rates," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 1985.

BIOGRAPHICAL STATEMENT

Bikas Raj Gurung was born in Kathmandu, Nepal, in 1979. He received his Bachelor of Engineering in Electronics Engineering from S. V. Regional Engineering College, India in 2002 and his Masters of Science degree in Computer Science and Engineering from The University of Texas at Arlington in 2008. His research interests include Network Security, Anonymity and Privacy on the Internet.