Securing Voice over IP Conferencing with Decentralized Group Encryption

by

Steven Kannan
S.B. Computer Science, MIT, June 2006
S.B. Mathematics, MIT, June 2007

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 4, 2007

Author____
Steven Kannan
Department of Electrical Engineering and Computer Science
September 4, 2007

Certified by____
Roger I. Khazan
Research Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

This research was sponsored by the United States Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are not necessarily endorsed by the US Government.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
04 SEP 2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	
Securing Voice over IP Conferencing with Decentralized Group Encryption	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Massachusetts Institute of Technology,Lincoln Laboratory,244 Wood St,Lexington,MA,02140**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**110**	

Securing Voice over IP Conferencing with Decentralized Group Encryption

by

Steven Kannan

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis addresses the development of an end-to-end secure Voice over IP (VoIP) conference system. We are particularly interested in challenges associated with deploying such a system in ad-hoc networks containing low bandwidth and/or high latency data links. End-to-end security is handled by the decentralized Public Key Group Encryption library (PKGE) developed at Lincoln Laboratory; PKGE allows real-time keying of conference users without an on-line central keying authority.

We present a system design and its prototype implementation in accordance with a set of appropriate design goals. The final product demonstrates the feasibility of using PKGE in the demanding conditions of VoIP conferencing. The system development sheds light on a number of issues and engineering challenges that ultimately affect call quality, functionality, security, and usability, motivating our recommendations for the next generation system.

Thesis Supervisor: Dr. Roger Khazan
Title: Research Staff
MIT Lincoln Laboratory

Acknowledgements

I would like to thank Roger Khazan for his insight and intelligence, and his willingness to apply these qualities so liberally to the project contained in these pages. His advice and knowledge were used for every stage of this thesis. I am further indebted to him for his commitment to my own education and development. I would also like to thank Galen Pickard and Joe Cooley. Galen put considerable work into the extension of PKGE and used his knowledge of the library to help me in implementation and the unpleasantness of debugging. Joe helped with all aspects of the project from day one, conveying technical know-how via Jabber on a moment's notice.

I would also like to thank my family and friends for moral support during the late nights of writing, coding, etc. In short, a great many people helped me complete this project.

Contents

Figures

Tables

Code

1 Introduction

This thesis explores the feasibility of developing an end-to-end secure Voice over Internet Protocol (VoIP) conferencing system using the Public Key Group Encryption library (PKGE) [40] implemented at MIT Lincoln Laboratory. It is part of a larger effort to secure dynamic, ad-hoc group communication in *disadvantaged networks* containing a mix of low latency, high bandwidth links and high latency, low bandwidth links; thus, designing a system that is optimized for such networks is also considered.

As a primary goal, we aim to design and implement a working proof-of-concept system that will shed light on the challenges and potential solutions in VoIP engineering. To this end, this thesis explains the development of a VoIP system, including the design decisions, the implementation details, and a barrage of tests for performance, reliability, and functionality. As a final deliverable, we reflect on the success of the system and suggest improvements in a second phase development.

We now outline our motivation, and summarize design, implementation, and results.

1.1 Overview and Motivation

One of the tenets of the Department of Defense (DoD) vision of Network Centric Operations and Warfare (NCO/W) [36] is that improved information sharing and collaboration ultimately results in improved mission effectiveness. A major part of this information exchange is envisioned to be done by dynamic groups of participants, so-called “Communities of Interests” (CoIs) [18].

Ideally, CoIs could assemble and communicate on a moment’s notice, without any significant overhead. They would be able to do this with minimal network requirements. Information exchange would occur securely, and in real-time through a variety of communication modes including text, voice, and video. One mission for this thesis is to contribute to that vision.

Specifically, we design and develop a secure VoIP conferencing proof-of-concept system. A VoIP conference involves multiple users speaking to one another in real-time, and can include concurrent speech from multiple sources that must be combined before played back. For security, this project uses PKGE, which is a decentralized group encryption and authentication library that allows users to secure communication in real-time without an online keying authority. It also allows users to secure communications without the excessive message-size overhead implied by other security solutions, such as S/MIME [28] and PGP [8]. In fact, message size during keying is virtually independent of group size [41]. Previous efforts in [40] have successfully used PKGE in a text chat prototype. We will test PKGE further, by putting the system to use under the more demanding conditions of VoIP.

Security is not new to VoIP, but our specific solutions address the DoD communication vision by optimizing for disadvantaged networks, which we define as two or more well-connected subnets linked with high latency, low bandwidth connections. Use of PKGE for security in such a situation is logical because its low message size overhead should not place high requirements on bandwidth. Of course, this is only helpful if the processing cost of PKGE is not prohibitive for high quality voice chat. Part of our motivation is to investigate whether this is the case.

Moreover, our solutions add a number of features that secure commercial VoIP systems lack, including end-to-end confidentiality and voice authenticity, and the ability to remain available even after the conference creator exits the conference. Thus, while we are contributing to the DoD communication vision, we also make novel extensions to securing VoIP technology itself.

1.2 Summary of Contributions

Work on this thesis started out with the idea of securing VoIP and a tool for doing it: PKGE. With those two seedlings, our project evolved through numerous designs and redesigns, addressing the challenge of engineering a VoIP system around the cryptographic backbone of PKGE. Through it, there have been a few advances that we have contributed.

Novel VoIP System Design

The system specified in this document makes two interesting departures from typical VoIP systems. First, it transmits data in a novel way to minimize the impact of low bandwidth, high latency links in the network it uses. While most VoIP programs transmit data either directly from peer-to-peer or indirectly between client and server, our system blends both concepts for the advantages of each. The second departure is that it compartmentalizes the various functional units, separating them along clearly defined interfaces. Separation allows simple customization for users whose needs differ from those that we outline here. Different functional units are easily swapped in and out, provided that they use the appropriate interfaces.

Proof-of-concept Implementation

We have assembled a working system that demonstrates the feasibility of the design we describe. Using Yet Another Telephony Engine (YATE) [17], Pidgin [9], and PKGE, the prototype is a useful starting point for future researchers who want to pursue similar goals. It is an extensible implementation that can be easily used by developers to experiment with their own designs for VoIP.

PKGE Evaluation and Next Generation Suggestions

Efforts in [40] prove the usability of PKGE in quasi-real-time text based communications. We extend this by using PKGE in a high intensity real-time voice communication system. The difference is subtle, but a text chat system is much slower paced than a VoIP system, based on the differences in data rate. Chapter 5 shows the data processing rate to be in the range of 64 kbps for PKGE in a VoIP system with approximately 50 PKGE operations per second. This is a much more stressful test than a text chat system that requires the processing of a few ASCII characters every few seconds, limited by the typing rate of its human users. In short then, we use this project to empirically show that PKGE is robust and usable in real-time.

Finally, we suggest how ongoing improvements in the decentralized cryptographic library can be used in a next generation VoIP conferencing system. Specifically, we indicate how the additional features of the successor to PKGE, GROK, can be used. We also use the evaluation and lessons learned from this proof-of-concept to make design guidelines for the next generation system.

1.3 Summary of Results and Recommendations

The main focus of this project was the development of a proof-of-concept VoIP application, which is currently in a source repository at Lincoln Lab. It was successful in that the system met design goals for call quality, usability, and security as outlined in chapter 2, using the test hardware described in chapter 5. Notably, call quality performance metrics are met despite the overhead of adding PKGE to real-time voice. While the system did not meet the functional goal of optimization for disadvantaged network, a design was specified, and the results from the proof-of-concept suggest that it too could meet performance metrics depending on the type of low bandwidth, high latency data link used. It could achieve total call quality with a TCDL [12] link, and tolerable call quality with Inmarsat [4] or Connexion [2] satellite links. Actual development of the disadvantaged network design is required to verify its usability. For links that cannot achieve call quality standards reasonable for typical conversations, users can compensate by using a conversation protocol much like that used over high latency radio links.

Another important result is that the processing latency from using PKGE, both during key distribution and during conversation, is not a serious bottleneck compared to the latency over the aforementioned weak links. PKGE adds on the order of 100 μs of processing and latency to send a 20 ms voice message, and about the same amount of time to receive one. This is compared to the 325 ms time needed to transmit data across an Inmarsat or Connexion link. PKGE adds roughly 30-35 ms to package and send a message with a new group key. This emphasizes the importance of implementing the disadvantaged network design. Along the same lines, the message size overhead during key distribution is almost independent of group size, allowing scalability, as we expected.

An important lesson shown by processing latency figures is that concurrent speech from multiple users during a conference increases the delay by a factor equal to the number of speakers. That is, the cost of sending or receiving a voice message (with a predetermined key) when there are three concurrent speakers is roughly 300 μs. Our design and implementation assume that such a scenario is rare, minimizing the impact of the overall quality of the conference. If this is not rare, this is a scalability limitation for future designers to account for.

Several more improvements are possible, particularly in the way of reliability and usability. The proof-of-concept system uses a Pidgin plug-in as the interface through which users join and leave conferences. However, our implementation provides no way to notify users when other users are running the plug-in, or are available for voice chat. Moreover, the system does not prevent users who are not running the voice chat plug-in from joining the conference. It also makes sense that all components of the system are started simultaneously, thereby avoiding these usability problems in most cases. Thus, this is an area where more development will lead to a more complete, robust application.

Similarly, it makes sense to allow multiple applications to use PKGE simultaneously; for example, text and voice chat. Concurrently with this project, Lincoln Lab has been developing an improved service, called GROK, which addresses this. GROK allows applications to access a single database of group keys that is consistent across applications. A future implementation could integrate text and voice into one seamless application. Users could join multimedia conferences, and then choose to use text, voice, or both. Building on the usability suggestions, other users could see who had voice functionality running. Users could also invite one another to turn their voice system on.

Because multiple applications can use the GROK simultaneously, sharing keys for the same groups, it may be possible to improve key distribution. Our system distributes keys as part of the voice stream, which uses UDP. In order to add reliability to this process, a VoIP application using GROK might distribute needed keys using Pidgin's TCP based communication. This is especially important in the disadvantaged networks where bandwidth restrictions cause packet loss. This optimization is thus left for the future developer who also implements the disadvantaged network design.

GROK also allows group keys to be stored and persist in a database even after applications terminate. This eliminates the distribution overhead when an application run at a later time requires a key for a group that has been used before.

1.4 Thesis Roadmap

This thesis begins with a problem definition, walks through a solution, details its implementation, and thoroughly evaluates it, assessing where further exploration is warranted. Technical details, along with some background reading, are included in the appendixes. Readers unfamiliar with VoIP and the specific technologies used should start there. Similarly, system users should peruse the installation guide.

Chapter 2 defines the problem we aim to solve with this project. We review motivation and form design goals on this basis. Chapter 3 uses the design goals to sketch out the high-level architecture for the system, analyzing what functional units are needed.

In chapter 4 we get into the nitty-gritty details. The exact implementation is described, including the specific technologies used and even some relevant code segments that shed light on how the system was put together. We show exactly how each function was implemented.

Chapter 5 puts the implementation on trial, measuring it against each design goal set in chapters 2 and 3. We outline a methodology for the performance tests and assess each dimension. The results in part motivate chapter 6, which discusses areas for further exploration and recommendations for the next generation system. The results also motivate our final thoughts in chapter 7, the conclusion.

2 Problem Statement

This section continues the discussion of using PKGE to extend the DoD communication vision in a VoIP system. We motivate and define the exact goals we have, based on the constraints of using PKGE and deploying the VoIP system in disadvantaged networks. Of course, we also consider the basic functional requirements for VoIP and the IP telephony community's standards for call quality. These goals will form the basis of the design we adopt and the steps taken to implement it. At the end of this thesis, we will then evaluate the system against these goals, and apply the results to a next generation system based on that evaluation.

2.1 Considerations and Constraints

Here we examine the main considerations before solidifying design goals. Considerations include the environment of deployment and the details of PKGE.

2.1.1 Deployment Environment

Because our system should advance the DoD communication capabilities and its vision for collaboration in ad hoc networks, we should maximize the range of networks we support. A system that requires a 100 Mbit/s Ethernet would not be suitable, for example. Rather, it is important that the system can perform as well as possible on Wide-Area Networks (WAN) including those that contain both low-latency, high bandwidth data links and high latency, low bandwidth data links.

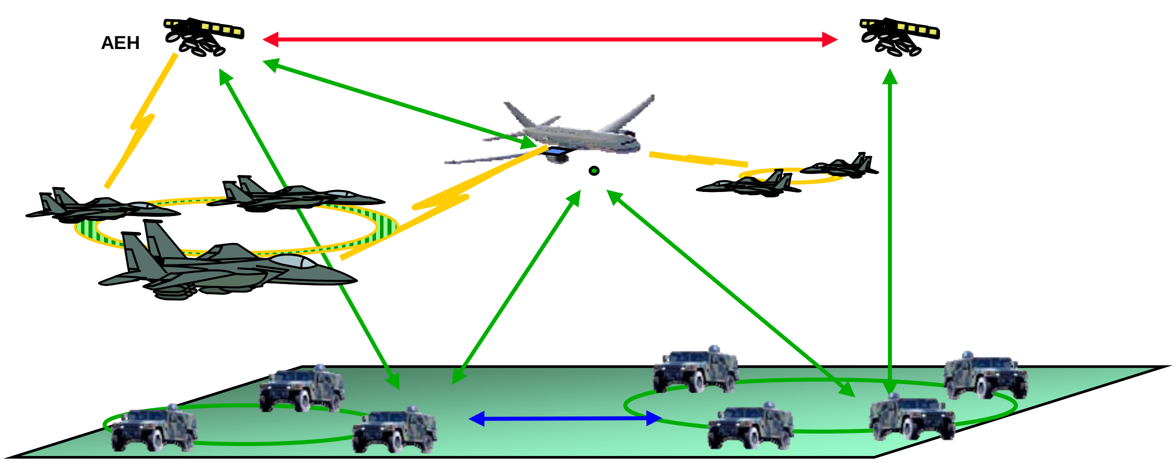

Figure 2.1 A Disadvantaged Tactical Network.

Consider for example a scenario involving a number of airborne parties collaborating via satellite with mission control on the ground, much like in Figure 2.1. The ground system may be very well connected and the airborne users may be very well connected, but the satellite between the two networks may be have a 2.4 kbit per second maximum bandwidth, such as an Iridium Satellite connection [5]. These links also have a high transmission latency of 2 seconds. Such a disadvantaged network containing two subnets connected with a weak data link will form the basis of our design. It may be impossible to get the same performance metrics on a geographically distributed disadvantaged network as on an Ethernet, but versatility mandates that we optimize for both situations. This will play into the design goals.

Just as we consider a range of network options, we also consider a range of deployment scenarios. That is, the system should be usable both in highly equipped state-of-the-art facilities such as research labs, and in reduced-resource situations such as isolated soldiers with only as much computational power as they can carry. The system should minimize hardware requirements, and maximize ease of use.

Thus, in our design we should consider how to create a VoIP system that can adapt to disadvantaged situations, allowing the maximum scope of deployability. Similarly, we want to minimize the system requirements in terms of hardware.

2.1.2 PKGE Details

In order to test its usability in a VoIP system, PKGE will provide the cryptography for our system. This includes encrypting/decrypting messages, providing message authenticity, and ensuring data integrity. Encryption is keeping message contents confidential, authentication is providing proof that the sender is among the group members, and integrity refers to ensuring that message contents have not changed between sender and receiver. Here we examine the technical details of PKGE and the various modes of operation it supports so that we can use it properly in the VoIP system.

PKGE is a decentralized dynamic-group based cryptosystem, in that it allows users to encrypt and sign messages for distribution to a group of users without relying on a real-time key distribution authority. That is, with PKGE, symmetric keys are securely distributed by the users themselves. The symmetric keys are then used to encrypt and sign messages during a conversation. When the group changes, a new key is distributed securely to the new group using a modified Boneh-Gentry-Waters (BGW) [20] broadcast encryption system and digital signatures. This key depends on the *security epoch*, which is the duration of time that a key is valid. After an epoch ends, the keys associated with it expire and are deleted from the system.

Existing approaches for decentralized encryption exist, such as S/MIME for secure email. In this approach, a user encrypts a random symmetric key using the private key of the intended recipient and encrypts the message with the symmetric key. The enciphered key is attached to the message ciphertext. For multiple recipients, this process is performed multiple times, making the message space linear with the number of recipients. [41] reports that a message sent to 50 users using 64-byte public keys would require 3200 bytes of space. This might stress the bandwidth limitations of our deployment environment. Other approaches for real-time decentralized keying also exist. In short, such schemes generally involve multiple communication rounds during the keying process, which is undesirable over a high latency link. A link speed ~100 ms is slow enough as it is without having to use it more than once during key distribution.

PKGE does not have these weaknesses. Instead, users distribute new keys as the group changes with a single constant-size message. What's more, this message is small, having only 200-300 bytes of overhead because it uses the BGW. The Boneh scheme is entirely stateless, meaning that each message includes headers for determining the symmetric key used to encrypt the message.

PKGE uses this scheme for distribution of 256-bit AES [38] keys whenever group membership changes. These keys are used in Cipher Block Chaining (CBC) mode to encrypt message payloads. After a group key is distributed, PKGE can operate in one of several stateful modes to reduce message overhead. Subsequent messages may not contain the headers necessary to obtain the symmetric key, on the assumption that users have stored in locally. Specifically, PKGE has four modes of operation, with different optimizations.

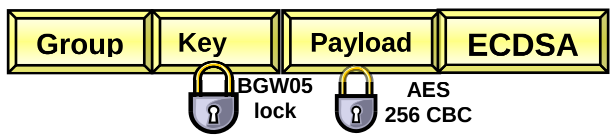

Figure 2.2 PKGE-Sealed Message, Stateless Protocol

- The *Stateless* protocol does not save any information in between messages. Thus, each message includes the group it is intended for, a key encapsulation using BGW, the payload encrypted with AES, and a 224-bit ECDSA digital signature for authenticity and integrity. The full message picture is shown in Figure 2.2.

- The *Optimistic* protocol is a stateful protocol that saves a copy of the symmetric key defined in the first message sent to a group. All subsequent communications to this group use the same symmetric key, without including the headers that allow computation of the key. This works because the receivers of the first message store a copy of the key as well. This way, computation time on the sending and receiving end is reduced because neither sender nor receiver has to recalculate the key. Also, message sizes decrease, because the headers are not included in messages. Messages are authenticated with the same digital signature as in the Stateless protocol. Thus, after the first, messages look like Figure 2.3, including session key identification information, the encrypted payload, and the signature.

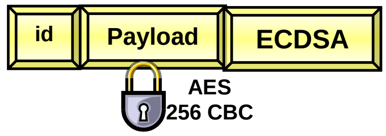

Figure 2.3 PKGE-Sealed Message, Optimistic Protocol

- The *Sessions* or *Pessimistic* protocol is the same as the Optimistic protocol, except the sender includes key defining headers in each message until she/he has observed each receiver use the key, thereby confirming its receipt. In this case, message size and computation time decrease only once the headers are omitted.

- The *Lean* protocol is similar to the Optimistic, but uses a keyed SHA1 hash of the message instead of a digital signature. This is called an HMAC. This reduces the computation time, because the HMAC can be computed more quickly than the ECDSA. Also, the HMAC is limited to the first four bytes of the hash, reducing message size.

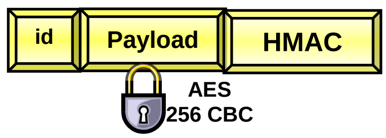

Figure 2.4 PKGE-Sealed Message, Lean Protocol

We will refer to the process of encrypting a message and adding a signature or HMAC as *sealing* the message. Decrypting and verifying the message will be referred to as *unsealing*. These terms are derived from the PKGE API, which encapsulates their functionality into `seal` and `unseal` methods.

All of the protocols except Stateless enable unique symmetric keys to be used for each group that exists throughout the duration of a conversation. This enables forward/backward secrecy, because as the group changes, so does the key. In the Stateless protocol, forward/backward secrecy is achieved because the key changes after every message. Similarly, they all account for authentication and integrity verification, either with digital signatures or HMACs. Note that forward/backward secrecy means that as users join and leave the group, they are not privy to communications from before joining or after leaving.

However, these protocols have different strengths and weaknesses. The Stateless protocol has the most fault tolerance, followed by Pessimistic. This is because Stateless does not assume anything, save for the public key infrastructure. The Pessimistic scheme is fault tolerant against messages lost in transition; however, user malfunctions that result in lost keys are not handled. Optimistic and Lean do not tolerate key loss in transition, because they only distribute the key once. However, they are the fastest and have the smallest message size. Lean in particular should be the fastest and smallest.

In fact, in [40] some performance metrics were taken. PKGE *seal* (encrypt and sign) costs roughly 10-20 ms after keys are distributed in the Optimistic and Sessions protocol, and seal before the keys are placed costs roughly 80-100 ms. Thus, the Optimistic scheme will consistently take 10-20 ms for seal after the first message, while the Stateless protocol will consistently take 80-100 ms. The unseal operation (decrypt and verify) was equally costly[1]. These measurements were based on 550 byte messages and used the following reasonable hardware: Powerbook G4, 1GHz Power PC, 1GB SDRAM.

The author did not test the Lean protocol, which should be even faster because it avoids a digital signature in favor of the HMAC. The digital signature is the clear bottleneck for the Optimistic protocol, because AES in CBC mode can encrypt at roughly 46 megabytes per second on similar hardware, based on OpenSSL[14] figures. In fact, OpenSSL speed tests show that the digital signature takes several orders of magnitude longer. On the other hand, HMAC can be computed at a rate of 16.7 megabytes per second on similar hardware. Our test results in chapter 5 will verify that this is the fastest protocol.

The message size advantages of using PKGE in a real-time group conversation suggest that it is useful in disadvantaged networks. However, it's not clear whether computation time will prove too much for VoIP quality. This depends on the design goals that we set in the next section. These performance metrics and estimations will suggest whether the system will succeed.

2.2 Design Goals

Here, we take into account the considerations of the previous chapters and define the goals for our VoIP system. We define what a conference system is, and describe our goals for functionality, security, and performance. In Chapters 3 and 4 we discuss a design and a proof-of-concept implementation of a secure VoIP conferencing system, and in Chapter 5 evaluate how well this implementation achieves the goals set forth in this chapter.

2.2.1 Conference Features

A *conference* is an audio conversation among a group of participants. This is not the same as a set of separate phone calls for each pair of participants in a group. Conferences are different because all concurrent audio is mixed into a single voice stream and played back to users, mimicking a real-life conversation among multiple people.

Additional conference features include the ability for participants to join and leave the conference at any time; this should not significantly degrade the conversation for remaining users. The conference should be consistently available, in that its persistence

---

[1] During our tests in chapter five, a bug was discovered in PKGE that caused the computation time for both seal and unseal to be roughly 5 ms longer than necessary.

is not dependent on any specific user being part of the conference. A conference should be scalable in that call quality should not degrade significantly as more users join. As a standard, a conference should support at least six users. This goal is somewhat arbitrary, but is motivated by the idea that a network must contain at least three nodes to be considered a network. Thus, if we want to support deployment on a WAN consisting of three different interconnected subunits, each of which contains at least two users, then we have to support at least six users.

Continuing the idea of WAN deployment, the conference system should be versatile enough that it can be deployed in a disadvantaged network containing high latency and/or low bandwidth links. In such a situation the design should minimize dependence on the weak link to best achieve the remaining design goals. It is possible that certain weak links make it impossible to achieve quality goals (section 2.2.3), however, the design should be adaptable to get as close as possible.

2.2.2 End-to-End Security

We define security in the context of PKGE and VoIP as:

> *Confidentiality* - Parties outside the conference cannot comprehend the audio data passed between current conference participants.
>
> *Forward/Backward Secrecy* – A user cannot understand the communications that continue after she/he leaves the conference, nor can she/he understand communications intercepted before joining. This is implied by confidentiality, but makes explicit what it means to be part of the conference group.
>
> *Perfect Forward Secrecy* – If long-term secrets, such as public/private key pairs, are compromised outside of the security epoch, the prior conference communications should not be compromised.
>
> *Group Authentication and Integrity* – Participants can verify that communication came from within the conference group and has not changed from what the sender originally intended. Sender authentication is not a goal because voice transmissions are inherently authenticated by the unique voice prints of the users.

Note that we do not include admission *authorization* as part of the security definition. This project is an experiment in providing cryptographically secure communications and showing the usefulness of PKGE. Because authorization is an entirely different project than simply securing communications, it is not considered. We do, however, leave the system customizable enough that future developers can add this feature.

Security functions can be added to any layer of a software system, or even to multiple layers. For example, a system may use IPsec [29] for network layer security or Transport Layer Security (TLS) [27] for transport layer. We adopt *end-to-end* data security, meaning that encrypt/decrypt operations are added to the end user applications only. This excludes models in which data is decrypted and re-encrypted by a server sitting between endpoints.

Such strictness is motivated by the trust model, which includes only the conference participants and the key creation authority for the pre-placed PKI used by PKGE. If the trust model were extended to include external VoIP processing servers, the data would be vulnerable to an extra point of attack for hackers. The extended trust model would also assume users trust the server administrators with the conference data. It makes sense to minimize points of attack and restrict trust as much as possible. The decentralized nature of PKGE lends itself to use in end-to-end secure systems, thus allowing PKGE to integrate well with this VoIP application.

End-to-End security also prevents data from being exposed to other layers of the system that the application runs on. Thus other applications should not be able to eavesdrop on an end-to-end secure VoIP application running in parallel. This is particularly important for shared computers.

Thus, in order to restrict trust to the minimum possible set of users and limit the extent of data exposure on a system, we adopt end-to-end security as a design goal.

2.2.3 Call Quality and Performance

Informally, we want clear, understandable voice data to facilitate a normal conversation interaction. For our project, this translates to reducing time between voice production and playback (delay or lag), and reducing packet loss on the network. For each of these metrics, it is more important to optimize quality during the call itself than to optimize the call setup or optimize the joining and leaving of users. This is because, most likely, the call setup and the joins and leaves of users will account for only a fraction of the call duration. Also, users are more likely to tolerate a one time delay during call setup than a consistent lag during conversation.

The main determinants of VoIP call quality, according to[43], are speech coding distortion, packet loss, packet delay, loudness, and echo. Because this thesis is concerned with adding security to calls, we will focus our evaluation on the qualities that our system is likely to affect. That is, it is pointless to assess our system on qualities that are out of scope, such as loudness and echo. Similarly, speech coding is out of scope, so we will rely on the quality of speech coding for the Public Switch Telephone Network (PSTN) by using the same audio codec. Specifically, we use the mu-law companding algorithm specified by International Telecommunications Union (ITU) recommendation G.711 [25]. By using G.711, the extent of call quality determined by the codec is guaranteed to be consistent with the PSTN. Our evaluation will thus concentrate on packet loss and delay.

Packet delay is caused by a number of factors including transmission latency, encryption cost, and audio compression and decompression. [22] specifies basic audio delay guidelines, measured from end-to-end or “mouth-to-ear”. Delays beyond 400 ms are considered unacceptable for most applications, while delays below 150 ms experience "transparent interactivity." However, applications that involve a significant deal of interaction between users may be affected by delays of 100 ms. Because the system's intended use is not specified at this point, we want to enable the greatest range of applications, including those with high interactivity. Thus, we ambitiously adopt 100 ms of audio delay as an upper bound for *total quality*, and 400 ms as an upper bound for *tolerable quality*.

Packet loss, which is caused by a number of factors including bandwidth restrictions and transmission errors, is generally very detrimental to VoIP. With the codec we have adopted, G.711, packet loss of even 1% can significantly degrade call quality [21]. Thus we adopt 1% as the bound for tolerable quality. Because of the sensitivity to packet loss with this codec, we adopt 0% as the standard for total quality. The acceptable packet loss levels depend on the codec used, with lossy codecs tolerating even less.

Quality	Max Packet Loss	Max Packet Delay
Total	~0%	100 ms
Tolerable	1 %	400 ms

Table 2.1 Call Quality Standards

Other call quality standards can be used, such as the Mean Opinion Score (MOS) [24] and the E-Model [23]. MOS is purely subjective rating based on the experiences of system testers. While user experience is highly appropriate for testing a phone system, it is unattractive here for several two main reasons. First, we lack the resources and time to recruit and train an adequate sample size of testers and subsequently run tests that will allow us to gauge success or failure. Second, the rating is subjective and its results are difficult to put in absolute terms for comparison with other systems, such as the PSTN.

E-Model is a rating system adapted specifically for VoIP systems and includes a mapping to MOS ratings. Luckily, E-Model relies primarily on an objective formula that uses the various factors of call degradation in VoIP, including packet loss and packet delay. E-Model is comprehensive enough that it breaks each factor down into subcategories, such as packet loss in transit, packet loss in queues, packet loss in encryption buffer, and packet loss to due to bit errors.

Because my objective is to determine call quality in a variety of scenarios for the system we have designed and implemented, the exact nature of call quality degradation is irrelevant. Thus we choose to evaluate quality based on the simple two factors outlined above. Based on these standards, tests of system overhead will be used to determine the minimum performance requirements for the network used.

2.2.4 Usability

The system should be robust and usable. It should be easy to install and use, with an intuitive interface. It should require few clicks to join or leave a conference. Any status or errors should be conveniently conveyed to the user. Graphic displays of membership and call information are preferred. Usability will be evaluated subjectively.

In addition to usable interfaces, the system should be usable based on the resources required to run it. As a goal, we strive to maximize the range of hardware on which the performance and functional goals are achievable.

2.3 Summary

In summary, we have made the following decisions for our VoIP system:

- Disadvantaged networks containing two or more well-connected subnets joined by weak data links are our deployment environment.

- PKGE is used for securing the voice communications of the conference.

- Conference goals include functionality, end-to-end security, call quality, and usability.

The next section will put these ideas into a cohesive conference system design.

3 Design

This section details the design of a secure VoIP conference system. The system will satisfy a number of design goals for security, functionality, performance, and usability. As an additional constraint, security features use the Public Key Group Encryption (PKGE) service developed at MIT Lincoln Lab. This project tests the usefulness of PKGE in the context of real-time security needs, specifically, VoIP.

A summary of the previous sections design goals follows:

Conference Features: We define conferences as groups of users communicating amongst themselves with real-time voice. Users can come, can go, and the conference availability does depend on any specific user. The conference should also be scalable and adaptable to disadvantaged networks.

End-to-End Security: The security goals are confidentiality, forward/backward secrecy, perfect forward secrecy, authenticity, and integrity.

Call Quality: We aim for total quality, defined as audio lag less than 100 ms and packet loss close to 0%. Tolerable quality is considered audio lag of less than 400 ms and packet loss of less than 1%. When total quality cannot be achieved due to physical limitations, tolerable quality becomes the goal. These goals are consistent with [21] and [24].

Usability: Goals include reducing resource requirements, using graphic interfaces where possible, and reducing installation and usage complexity.

As an overarching design principle to achieve these goals, we adopt a “divide and conquer” strategy to the different functional units of our system. By building separate modules for different functions and connecting them to meet specific design goals, the system can support customization. Functional separation ensures that achieving the design goals of one module does not interfere with achieving the design goals of another. Furthermore, by connecting modules along well-defined interfaces, we can easily swap in different implementations for a module if the need arises; for example, to optimize different performance metrics or to meet revised system requirements for functionality as well.

Thus we have divided the system into three functional units. Modules exist for: voice data transmission and processing, conference membership management, and security. This section deals with defining the function and interaction of these three components, and making clear references to how they achieve the design goals. The final high-level design looks like Figure 3.1.

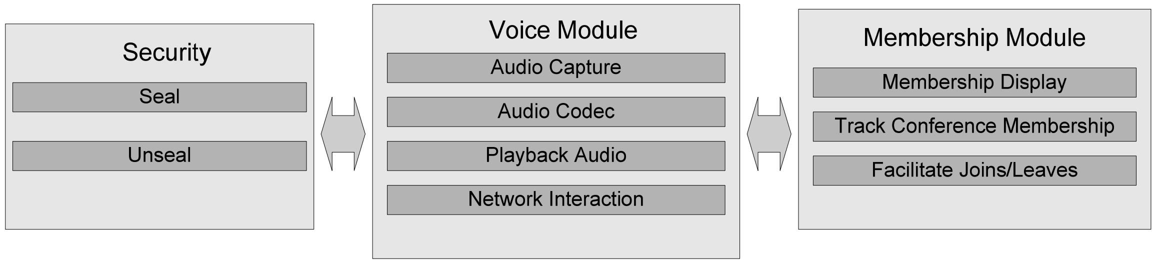

Figure 3.1 System Component Functional Separation

The security module provides a service that allows data to be encrypted/decrypted and verified. The voice module handles audio compression and decompression, maintaining IP address info of conference participants, creating connections between users, passing data onto the network, as well as capturing and playing back audio. Membership management involves maintaining the list of usernames in the conference, displaying this information to users themselves, tracking the mapping of usernames to IP addresses, and allowing users to join and leave. Naturally the membership module and the voice module must communicate so the voice module knows where to transmit data. Similarly the voice module and the security module must interact to secure the voice data. The voice module passes raw audio data to the security module, which seals it and passes it back. On the receiving end, the security module unseals the data.

We could split the software along different functional lines, but these three partitions are the most logical. Voice data and membership data are separated because the security goals for the two are different. That is, the goals outlined for end-to-end security in 2.2.2 apply to voice communications, not membership information. Membership information is not considered confidential in our project, so it is nonsensical to treat it the same way as we treat voice data. Security is separate because we want to use PKGE. By keeping it as a separate module invoked through function calls, the system can be easily modified to use a different implementation for security. In fact, the next generation system will swap out PKGE for GROK to take advantage of its improved features. Because GROK has the same interface, this transition will be trivial.

The voice data module could have been split up further; for example it could be split into voice routing module and a voice compression module. However, there are no obvious gains from this because we have no desire to experiment with changes in audio codecs. Our focus is the feasibility of adding encryption and authentication to VoIP conferences. Also, further division would complicate implementation and add to latency from inter process communication.

3.1 Transmitting Voice Data

In this section, we present our model for processing voice data and transmitting it between the multiple participants of a conference call. We focus on providing a design that is versatile enough to achieve our design goals, even in disadvantaged networks containing low bandwidth and high latency data links.

Thus, we present a dual-mode system that can take advantage of high bandwidth networks, but is also adaptable to low bandwidth situations. In either case, it attempts to reduce processing and transmission latency by minimizing the number of costly seal and unseal operations, and minimizing the network congestion. Network congestion minimization also reduces dropped packets, the second measure of call quality.

3.1.1 The Dual Mode Model

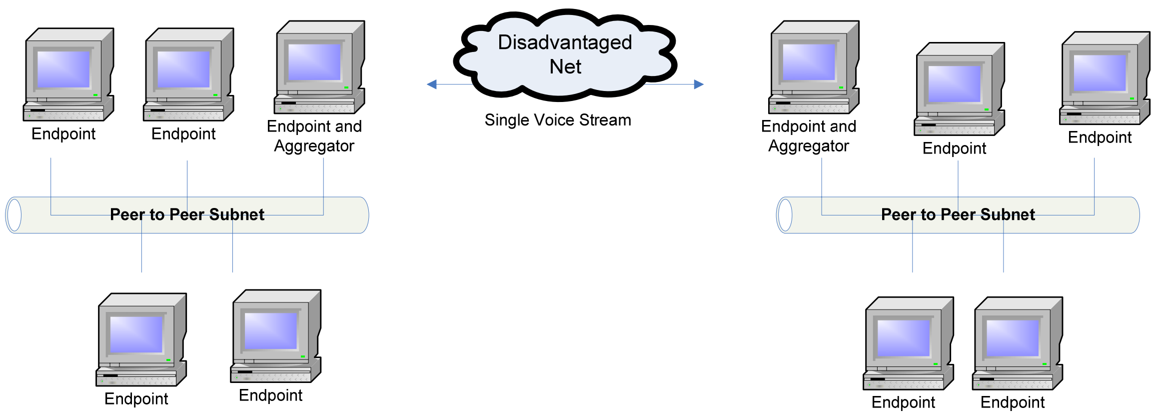

Figure 3.2 P2P Subnets Connected by Low Bandwidth Links

Our conference system is characterized by self-sufficient endpoint software that handles all aspects of the voice module. This includes the audio playback/capture, audio codecs, and voice transmission directly between peers. The endpoints do not rely on servers for anything voice related, but rather use strictly peer-to-peer voice transmission when deployed in well-connected networks. However, the system is configurable to networks containing low bandwidth, high latency data links. The system is optimized for such a network by minimizing the amount of voice traffic across such as link.

When weak links exist in the system, the network is separated into well-connected subnets on either side of the weak links. Such a scenario is exemplified by a conference call between several parties in an airplane and several parties in a terrestrial network. The airborne endpoints are connected amongst themselves in a high performance local network as is the ground crew. However, the two subnets are connected via a low bandwidth satellite link. We call this the *two-tiered* model.

Traffic over the link is reduced by ensuring that only a single voice stream is passed across at any given time. This is accomplished by electing an *aggregator* node in each subnet. While the rest of the subnet is communicating in a purely peer-to-peer way, the aggregator nodes act as intermediate hops between nodes on opposite sides of the weak link. An aggregator will mix the voice data produced by all concurrent speakers on its subnet and transmit the combined stream across the weak link to the aggregator on the other subnet. The aggregator on the other subnet will transmit this voice stream to each local endpoint. Each endpoint then mixes this voice stream with any voice streams coming from within the subnet. The final product is played back to the local user.

In this fashion, we prevent the scenario in which multiple speakers in one subnet all attempt to send data across the low bandwidth link. The required bandwidth would increase by a factor equal to the number of speakers. If the bandwidth demands exceeded the bandwidth available on the weak link, there would be reduced performance in the due to lost packets.

In a typical voice conference, only one source is transmitting data at any given time, reducing the function of the aggregator to blindly forwarding data to the aggregator on the other subnet, which in turn forwards it on to the endpoints on its subnet. No intermediate encryption or decryption need take place. In the rare case that there are multiple speakers at the same time on a subnet, the aggregator decrypts, mixes, and encrypts the signal before routing it to the next aggregator. In this case, there is an increased cost of extra seal/unseal operations, but this is traded off for the possible dropped packets and increased latency that would have been introduced into the system from the additional traffic across the weak link.

The only constant cost introduced by the aggregators is that of the extra node through which data passes even when aggregation does not occur. That is, even when there is only one speaker, the data is still routed to the aggregator before crossing the weak link, which is an extra hop. To reduce this cost, the processing that occurs at this step is very low.

The aggregator acts as a finite state machine with two states: the aggregation state and the forwarding state. Starting in the forwarding state, the aggregator automatically forwards data to other subnets before processing. If it begins to receive data from multiple sources, it switches to the second state, aggregation. In this state, it decrypts data it receives from each source, mixes it, encrypts the new signal and only then forwards a single stream to outside subnets. If it detects that there is only a single voice stream, it reverts back to the forwarding state.

Despite having aggregators that are necessary for communicating to outside subnets, proper implementation allows this system to remain headless. Even if an aggregator leaves the conference, the conference persists because the endpoints in a subnet can simply elect a new aggregator. This selection process would be deterministic based on conference membership to avoid extra communication.

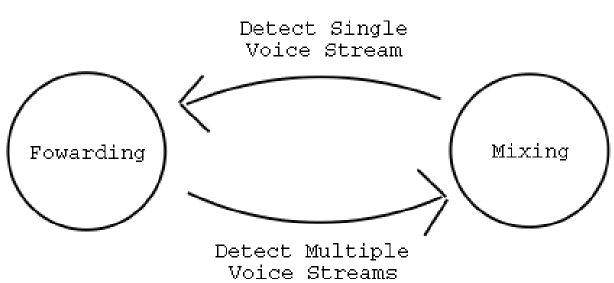

Figure 3.3 Aggregator State Diagram

When the weak link is not involved, voice data is sent directly between peers. This offers a number of advantages over a client/server routing model. P2P routing reduces the number of hops in a packet's path between endpoints. Each buffer that the packet enters will introduce additional latency that decreases call quality. Moreover, by routing data between peers instead of through a sever, there is no single point of failure. One of the design function requirements was for headlessness. With P2P routing, the conference does not depend on any particular participant.

Additionally, P2P routing makes end-to-end encryption simple to implement. Because data goes directly between users, the VoIP application merely encrypts at one end and decrypts at the other, thereby achieving end-to-end encryption. The system could have used a server to do some of the data processing, such as audio mixing. However, the server would have to use plaintext voice for the mixing, and thus would have to be able to decrypt the audio. Unless the server is a conference participant itself, this would violate our trust model and the end-to-end principle. The aggregator nodes in the two-tier model are conference participants and thus do not violate the trust model.

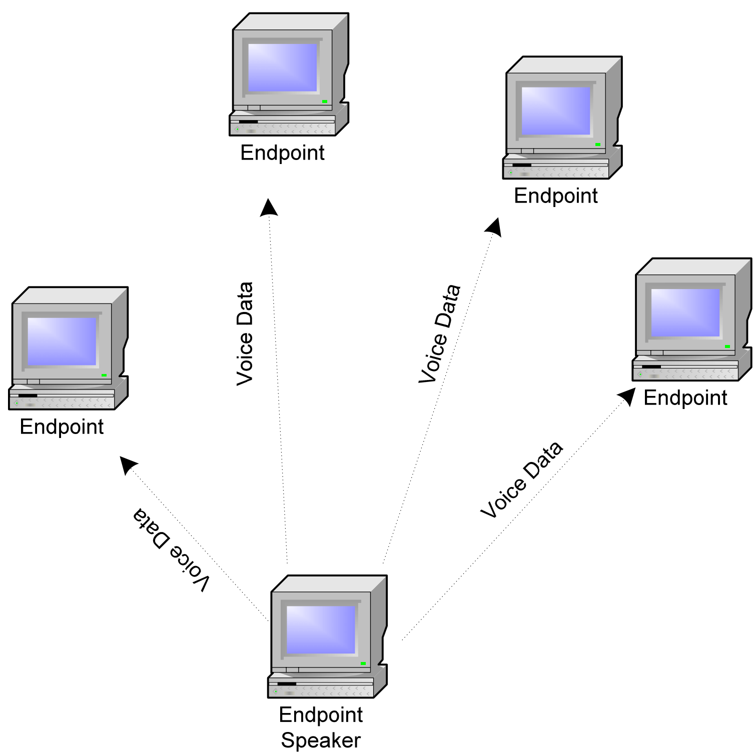

Figure 3.4 Peer-to-Peer Routing in a Well-Connected Subnet

P2P within the subnet offers another advantage, reduction of the number of encryption and decryption operations. When transmitting data, each user transmits n-1 streams of voice data for a conference of n users, unless UDP multicast is used. If multicast is used, users only need transmit a single stream, which is routed to end users appropriately by the multicast implementation. Regardless of the transmission model, users receive a stream of data for each conference member who is speaking. Despite the number of voice streams, because a shared key is used, the speaker need only run one encrypt operation. She/he stores the encrypted data and forwards it to the other users.

If there were a server processing data, the server would decrypt and re-encrypt data, introducing more computational latency. Worse yet, the server would have to run a different encryption operation for each user, because users each receive a different voice stream. The voice streams are different because a speaker should not hear her/his own voice streams.

Thus, this conference system offers a number of advantages. When transmitting data from peer-to-peer, transmission latency is decreased by minimizing the number of encryption operations and minimizing the number of hops in a packet’s path. P2P transmission also trivializes implementation of end-to-end security. In the presence of a weak link, the system is adaptable to maintaining these advantages and adhering to design goals as much as possible. Performance degradation due to low bandwidth is reduced by minimizing traffic across the link with aggregators. The aggregators themselves reduce their own latency by forwarding traffic with minimal processing when there is only a single voice stream. It is only when multiple voice streams threaten to overload the network that the aggregators trade off extra encryption and mixing latency for reduced network congestion.

3.2 Membership Management

The membership management module contrasts the voice module by using a client/server architecture. A central server maintains a list of usernames and IP addresses in a conference. When a user wants to join the conference, she/he sends a request to the server, which admits her/him to the conference, and informs the existing users of the new member and her/his IP. The server also relays the current conference membership to the joiner. The same process takes place when users leave the conference.

Client software handles sending and receiving the join requests and updated conference membership information from the server. It also displays this information to the end user, and communicates group membership and member locations to the voice module. The client software should have a well-defined interface for voice module communication, ensuring that the voice module need not change if a different membership module is implemented. The interface between the two is defined in the implementation section.

The client/server model is acceptable in this case because membership information does not require end-to-end confidentiality in the design goals. Thus using a server does not violate any constraints. However, it makes sense to restrict access, because the membership of the conference can be considered valuable in some cases. Thus, our system could allow for confidentiality in an extended trust model that includes the server. In short, the security goals for membership information are relaxed. If a future designer adopts goals that include securing membership information from end-to-end, the designer should re-implement the module under a different paradigm. By using the same interface with the voice module, no changes need to be made outside of group management.

Using a server for membership management has several advantages over using a peer-to-peer model as the voice module does. First, the server makes joining and leaving conference simple. Users need only know the location of the server and issue a single request to join. If the joiner had to send requests itself to all participants, it would need to acquire a list of users first, which would add overhead to the join request. This simplicity keeps overhead low and implementation easy.

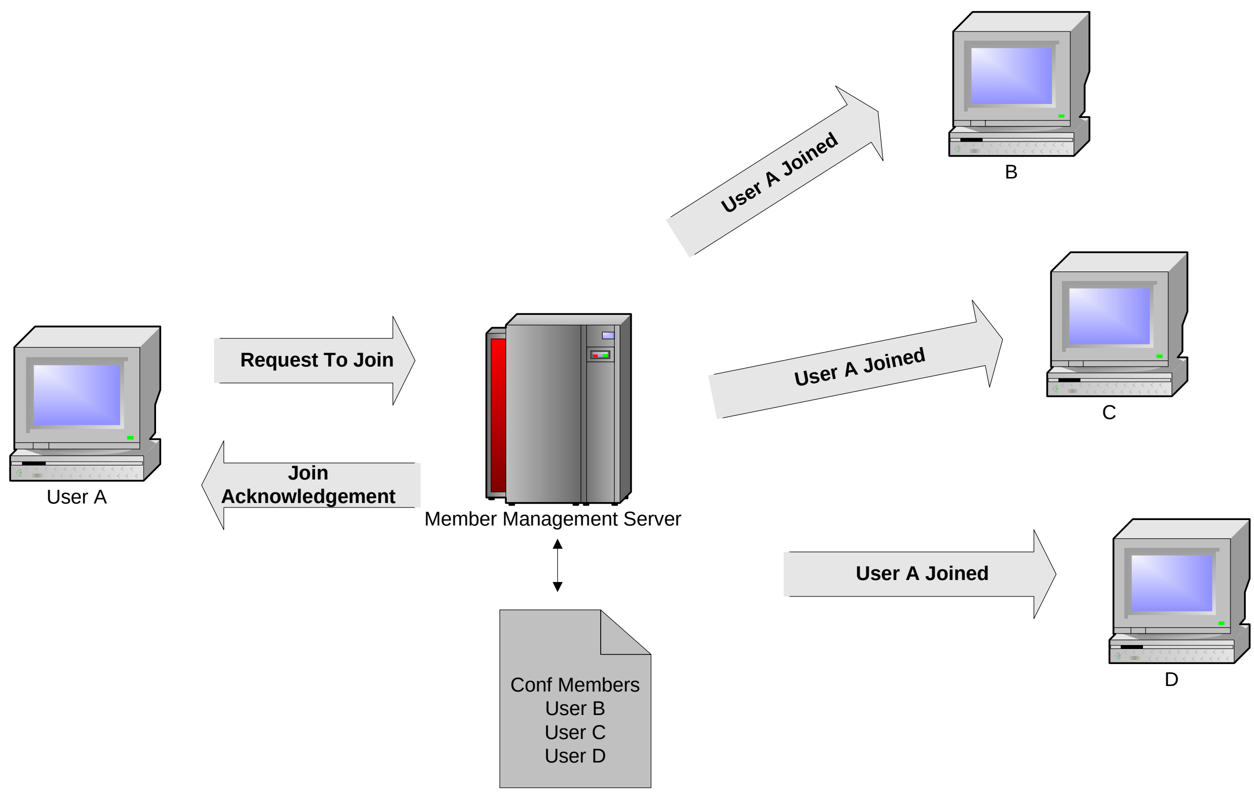

Figure 3.5 Membership Module Design

Furthermore, if there are admission requirements, use of a server allows a single authority to make a decision about whether to allow a user to join. Without it, the existing users would need some protocol for negotiating join requests. This protocol would add traffic to the network and latency to the join process. This unnecessary overhead could erode call quality.

However, a central server for membership management does introduce a single point of failure. If the manager goes offline, we merely lose the capability to change the conference membership; the existing connections for voice transmission handled by the other module remain active. This is another advantage of separating the system components. Note that even if this server is subverted, or is not a trusted party, the most harm it can cause is denial of service. It cannot view encrypted communications, nor can it allow others to view encrypted communications because it does not deal with encryption keys. Thus it remains consistent with design goals.

The advantages of using a server reduce the overhead required for joining a conference, thereby improving call quality. Thus, the use of a client server membership management module helps achieve design goals.

3.3 Security

Part of the mission for this thesis is to use PKGE in a real-time VoIP application. Thus our security module *is* PKGE. However, it is important that PKGE fits into the design goals, and is properly integrated with the system as a whole.

PKGE achieves the security goals handily, having been designed with end-to-end security in mind. PKGE is a cryptographic library, and by invoking it from the voice transmission module, we ensure that it is used at the application layer. Moreover, [40] claims fulfillment of confidentiality, forward/backward secrecy, perfect forward secrecy, and authentication/integrity. Confidentiality is derived from use of AES symmetric encryption on all messages sealed. Authentication and integrity come from either digital signatures of HMACs, depending on the protocol. Forward/backward secrecy is achieved by using unique keys for unique groups, and perfect forward secrecy is achieved by re-keying after epochs expire. Old keys are destroyed at their expiration, making saved communication impossible to decrypt outside of its epoch.

Furthermore, PKGE is ideal for usage in group conversations because of the group oriented nature of the protocol. It allows real-time group keying, thereby allowing users to join and leave at will, as per design goals. Moreover, because message size during keying is independent of group size, PKGE enables scalability without degrading the conference quality, even in low bandwidth situations.

Conference quality is an issue on its own, and PKGE's performance metrics in [40] suggest that call quality goals are achievable, although this is network dependent. Our own measurements in chapter 2 suggest that the overhead of PKGE after keys are distributed should be on the order of microseconds when using the Lean protocol, which is negligent compared to the design goal for total quality (100 ms total lag). Keying, however, should cause slowdowns on the order of 80-100 ms on the test hardware of [40].

The API for PKGE is simple and useful. When we want to seal messages, the calling program simply invokes the `seal` method, and invokes `unseal` for unsealing. This makes the integration with the rest of the system very simple. Moreover, it fulfills our need for simple interfaces that allow future system to use different software components. In this case, a future library implementing this interface can be seamlessly swapped in.

For these reasons, PKGE will provide a suitable cryptographic backbone for the VoIP system. Of course, because of the component separation principle, we could use a different security module with the same interface in future systems if design goals for security or performance change. For this system though, it is appropriate to use PKGE in order to test is viability in VoIP.

3.4 Design Summary

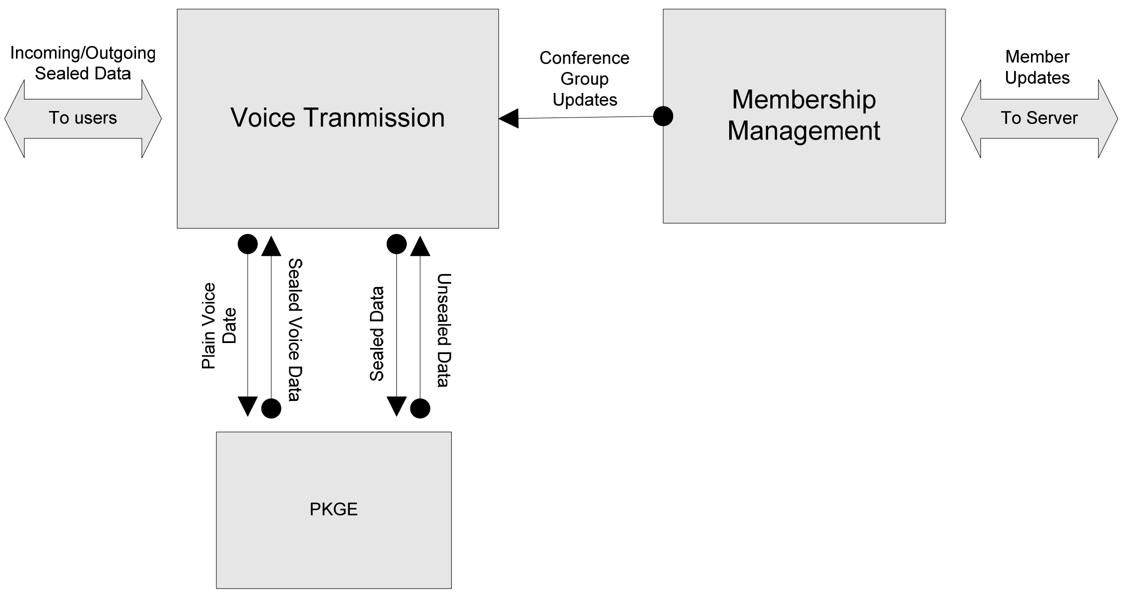

Figure 3.6 Block Diagram for End User System w/ Interconnections and Data Flow

In sum then, we choose to divide the VoIP system into three interconnected modules. There is a voice data module which will handle call setup and voice data transmission in a strictly peer-to-peer manner in order to facilitate end-to-end encryption. This module is optimized into a two-tiered network in the presence of high latency, low bandwidth data links. We use a client/server member management module to reduce network congestion during member joins and leaves. We use PKGE for the performance benefits of shared symmetric key cryptography and key exchanges of fixed message size overhead. PKGE also allows realization of the security design goals.

4 Implementation

Recall that the design specifies three major functional components: membership management, voice data transmission and handling, and security. The voice data module includes call connection, data transmission, mixing, and audio playback. Membership tracks the changes in the conference group, and voice encryption and authentication is implemented by PKGE. This section outlines how we chose to implement and connect each of these in such a way that would remain faithful to the design goals.

4.1 Architecture and Components

For the encrypted conference system, we choose to use the open source VoIP engine Yet Another Telephony Engine (YATE) to handle the voice data transmission, compression/decompression, and mixing. YATE has a SIP stack which we use to implement a call connection script to meet the data transmission design specifications. It contains a conference module that allows realization of the conference functionality we specified. It also has a GTK based audio playback client, but we choose to use a different freeware VoIP client call X-Lite because of its usability. Public Key Group Encryption (PKGE) developed at Lincoln Lab is used for encrypting and authenticating the voice data that YATE handles. The open source chat protocol Jabber[6] is used to maintain and communicate conference membership information, which the user accesses with a plug-in developed for the open source Jabber client Pidgin.

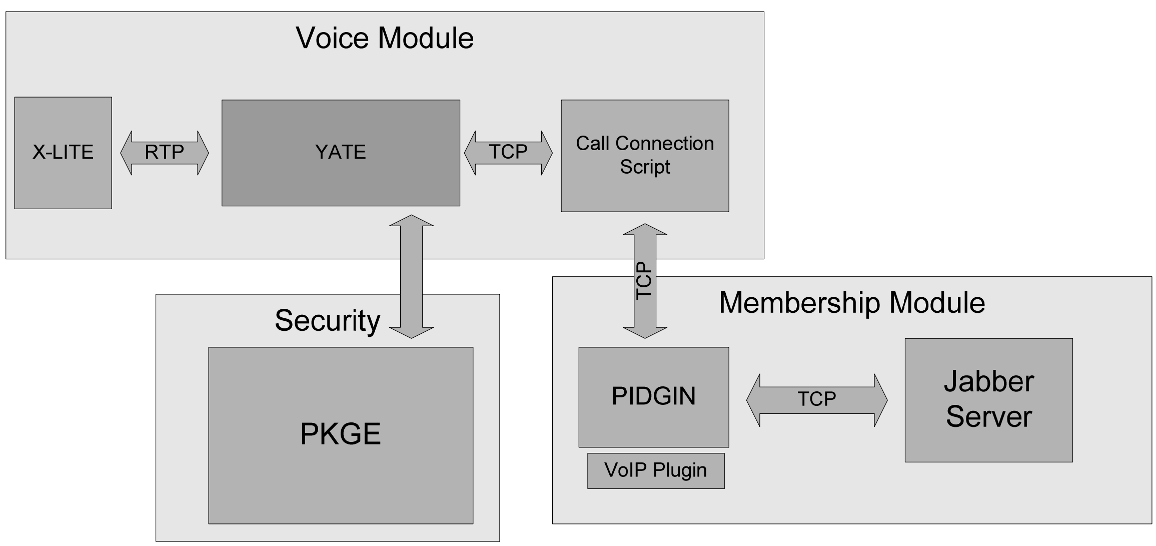

Figure 4.1 System Architecture

When necessary, most communications among the different modules use TCP sockets with predefined protocols. PKGE, however, is packaged as a library that can be invoked from YATE with function calls. The protocols for communication allow any implementation for each module to seamlessly integrate into the system, provided that it can understand and use the protocol.

4.1.1 Voice Data with YATE

This conceptual module includes call connection, voice routing, audio mixing, audio playback, and audio compression/decompression. Including all of this in one section may seem to contradict modular design goals. However, we combine them because audio mixing, compression, and playback are beyond the scope of this project and thus their implementation details are irrelevant to us. While call connection and voice routing implementation is important, they are combined because their design goals are the same. Moreover, most software telephony engines already combine all five of these together. Thus, our logical choice is to use an open source engine that includes all of these functions and is extensible enough that we can implement the call connection and voice routing that our design calls for.

This does not mean that all of these pieces *must* be implemented with the same software. It means that we do not require well-defined interfaces and ease of swapping other implementations in. Thus we have free reign to reuse as much code as possible, while modifying it to our needs.

The YATE software PBX is a suitable choice. Like many telephony engines, such as SipX and Asterisk, YATE meets the requirements for functionality (compression, conference mixing, etc) and is available on multiple platforms. However, YATE stands out because the code is highly extensible. For a detailed description, see Appendix A. Importantly, the system allows developers to write external Python scripts that control call connections by passing messages of a pre-defined protocol to the main system via TCP socket. Moreover, the core system itself is extensible enough that we can change how data is routed from a client/server design to a peer-to-peer design. Also, we can insert encryption calls into the system.

4.1.1.1 From Client/Server to P2P

YATE uses a client/server paradigm for conference calls. Our design mandates that data is routed directly between peers and encrypted at the endpoints. Morphing the client/server PBX into endpoint software seems like a daunting task, but traces through YATE code and careful design analysis lend insight on how to do this.

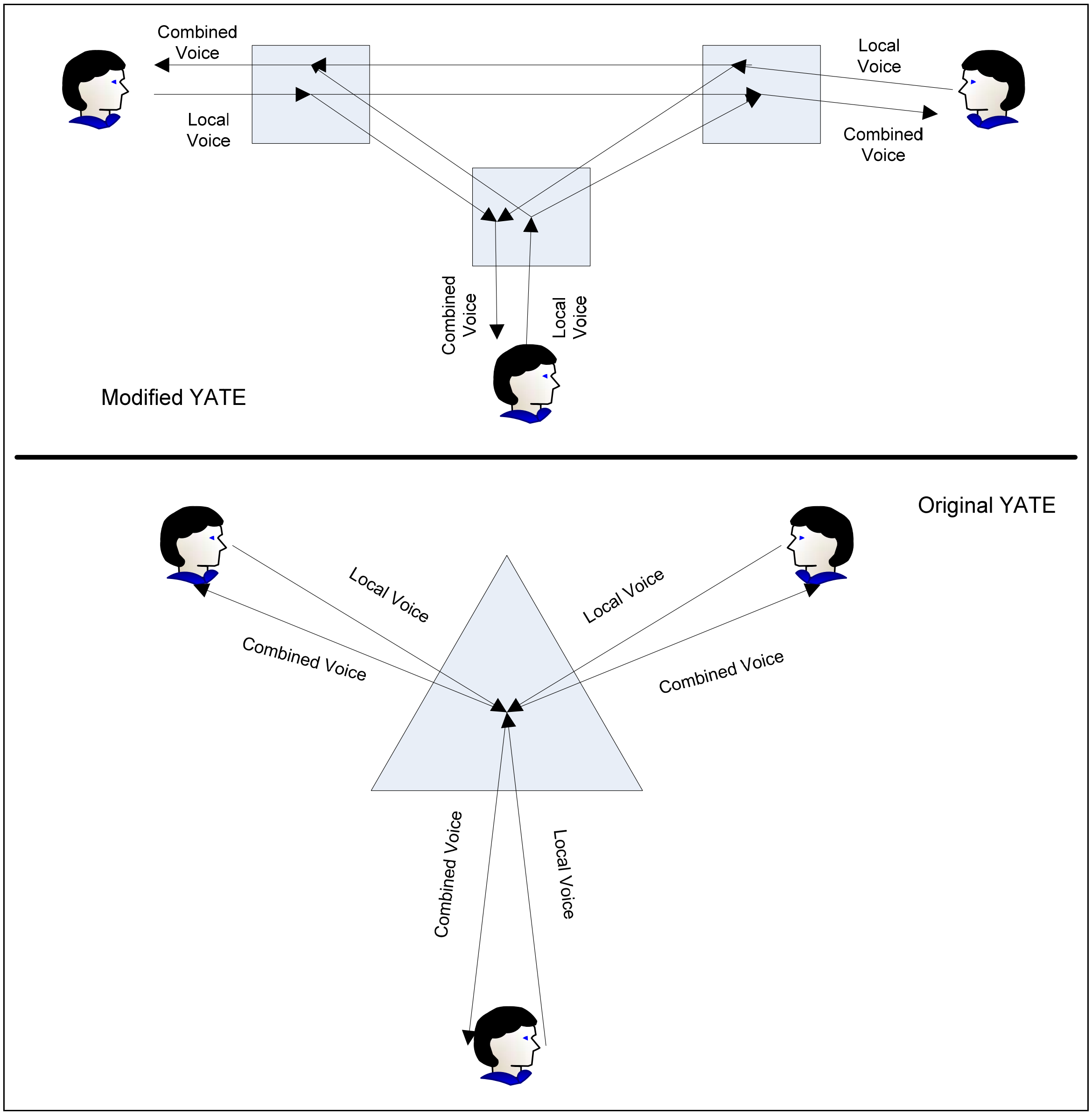

Figure 4.2 Comparison of YATE Conference Configurations

In a typical YATE conference call, multiple clients communicate voice data generated by their users to a single YATE server, which mixes the data if necessary, and redistributes the combined stream to every end user in the conference. Our system requires two differences. First, each client runs its own instance of YATE. In fact, we redefine client to include an instance of YATE. YATE still receives voice data from each party in the conference call. However, the second difference is that the combined stream is only transmitted to its local client. All other clients are passed the single stream from the local client.

To understand why this works, realize that each end-user is receiving the combined voice stream, because each user runs its own instance of YATE. Because each instance of

YATE passes the data generated by its local client to each other instance of YATE, each instance of YATE receives all the data necessary to transmit to its local user. The diagram in Figure 4.2 should clarify this. It shows that in the original version of YATE, each user communicates with the same instance of YATE, passing its local data and receiving the combined stream. In the modified version, each user communicates with its own instance of YATE, which passes locally produced data to other users' instances in exchange for their locally produced data. Each user's instance of YATE produces the combined stream for the user to playback.

Based on this it should become clear how simple it is to change the code of the system for this behavior. In the new system, YATE is transmitting and receiving the same streams of data as in the client server model, the only thing that changes is who receives what and from whom.

In the code shown below (Code Chunk 4.1), the YATE conference module passes the data buffer `buf` to a `ConfConsumer co`, which is encapsulated in a `ConfChan` channel object. `buf` contains the combined voice stream for all the users in the conference and each conference channel is the data line to a different end user. The `co` for the channel object passes the data through different parts of the system including audio compression and an RTP formatting module. Eventually the data reaches the wire, where it is transmitted to the end user that `co` encapsulates. Also, `len` specifies how many audio samples are in the buffer, so `consumed` can subtract the right amount from the user's incoming voice buffer and send the right amount over the outgoing UDP socket. This code is called for every channel attached to the conference (every end user). Thus, every end user receives the combined voice stream from this conference code. Because we only want the combined stream to reach the local user, we only call this line if this is the local user, which is always stored first in the list of channels.

```
for (l = m_chans.skipNull(); l; l = l->skipNext()) {
    ConfChan* ch = static_cast<ConfChan*>(l->get());
    ConfConsumer* co = static_cast<ConfConsumer*>(ch->getConsumer());
    if (co) {
          co->consumed(buf,len);
    }
}
```

Code Chunk 4.1 Routing Mixed Data to All End Users Attached to the Conference

Thus we make the following change:

```
int i = 0;

for (l = m_chans.skipNull(); l; l = l->skipNext()) {
    ConfChan* ch = static_cast<ConfChan*>(l->get());
    ConfConsumer* co = static_cast<ConfConsumer*>(ch->getConsumer());
    if (co) {
          if (i==0) {
                co->consumed(buf,len);
          }
```

```
            else {
                  co->consumed(buf2, len, hostlen);
            }
      }
      i++;
   }
```

Code Chunk 4.2 Modified YATE Routes Mixed Data to the Local User Only

This section of code now transmits `buf` only to the local user. Other users are passed `buf2`. Before this section of code is run, we cache the local user's voice data in `buf2` in the same section as the mixing of all voice streams into `buf`. `hostlen` contains the number of audio samples in `buf2`. We also override the `consumed` function so that we can pass both `len` and `hostlen`; this way, the correct number of samples (`len`) is subtracted from the incoming buffer, and the correct number of samples (`hostlen`) is sent over the wire. In the previous case, the two numbers were the same, which is why we had to override the `consumed` function in our modifications.

This change, along with the overloaded `consumed` function, is all we need to make the YATE conference module route calls for the P2P design. However, without changing how calls are connected, the conference code is not complete. We need to modify the system to allow separate instances of YATE to connect and begin RTP sessions. Normally, several users all call in to the same instance of YATE. We want users to connect with their local instance of YATE, which will automatically create outgoing connections to other YATE instances, according to the conference membership list.

4.1.1.2 Connecting Calls

In order to manage these incoming and outgoing connections, we take advantage of YATE's external scripting interface. This allows developers to write Python [10] scripts using a library of functions that communicate with the YATE engine using its message passing system. See Appendix for more details on this. For our system, all call channels are connected to the conference module of YATE, because this module mixes audio data. The following code illustrates how an outgoing connection from a conference is created by the YATE script.

```
yate.msg("call.execute", {"callto" : "dumb/", "target" :
         YATE_CONNECTION_TYPE + "/" + YATE_CONNECTION_CHANNEL + "@" +
         users[stringAddr],"caller" : "Roger", "callername" : "Roger"
         }).dispatch()

yield yate.onwatch("call.answered")

answered = getResult()

yate.msg("chan.masquerade", {"message" : "call.execute",
         "callto" : "conf/1", "id" : answered["id"]}).dispatch()
```
Code Chunk 4.3 Adding an Outgoing Conference Connection

YATE conferences cannot create outgoing calls by themselves, so we start by initiating a call from a *dumb* channel. The first yate.msg statement accomplishes this by passing a `call.execute` message to the telephony engine. This message, which signals the creation of a call, is handled by the dumb channel, as designated by the "callto" parameter. Other parameters designate the connection type (SIP), and the designated conference connection channel. `users` contains a mapping of usernames to IP addresses; thus `users[stringAddr]` designates the IP address of a user stored in `stringAddr`.

When the outgoing call is answered by the receiver's conference module, we connect the dumb channel to the local YATE conference module, as shown in the second yate.msg statement. The yield statement ensures that the flow of execution does not continue until the call is answered. This completes the connection between the two instances of YATE.

Once this connection is complete, we store the channel ID for the connection in a *dictionary* structure, indexed by username of the user at the other end of the connection (not shown). This way, we can easily close connections by sending "call.drop" messages to the telephony engine, passing the appropriate channel ID as a parameter.

It is important for the telephony engine to keep track of the group of users in the conference, because it will ultimately have to encrypt data. However, we chose not to overload the "call.execute" message with this information. That is, we could have modified the call.execute message handler in the conference modules by allowing it to receive the username as a parameter. However, "call.execute" is used by many modules throughout the system for every type of call, so we decided not to make any changes here. Instead, we added three new messages to the system: "yate.add", "yate.remove", and "yate.self." Additionally we wrote handlers for the messages which add/remove usernames from the encryption group or add the local user to a new encryption group.

With these new messages, the script can send messages to add and remove users, as shown in this line of code:

```
yate.msg("yate.add", {"callto" : "conf/1", "user" : member}).dispatch()
```

Because the call connections are based on the conference membership, it is here that we create the communication interface between membership management and voice data.

4.1.1.3 YATE Call Connection Interface

To keep with our design goals, we require a complete and concise interface between the membership module and the call connection module. It should expose all the functionality of the call connection script, while hiding its implementation. It thus allows re-implementation of either module without affecting the other.

The call connection module has the following capability. It can create outgoing connections to users, remove outgoing connections to users, shutdown all outgoing connections, add members to an encryption group, remove members from an encryption group, and create encryption groups. It also receives IP address information about users in question.

We thus define a simple communication protocol between the modules that allows the membership manager to issue one of seven commands as shown in Table 4.1. The entire command and its parameters are combined into a single ASCII string terminated by a newline. Each command is exactly four characters. By fixing the length to four characters and making clear command delineation with the newline character, we simplify the parsing of commands and identifying errors.

Command	Parameters
Join	Username
Leav	Username
Clos	*None*
Add-	Username
Rem-	Username
Bind	Username + “TO” + IP addr
Self	Username

Table 4.1 Call Connection Interface

Commands are transmitted over a predefined local TCP port (6060) by the membership module. The call connection module acts as the server in the TCP connection, listening for the commands.

The current implementation of this interface lives in the call connection script. It sets up a socket listener, accepts an incoming connection and reads commands one at a time. If there are errors, the script discards the command and waits on the next one, without passing feedback to the command issuer. Code for this is shown in Code Chunk 4.4.

```
data = conn.recv(1024, socket.MSG_PEEK)
            if not data:
                    break
                else:
                    delimiter_index = data.find("\n")
                    data = conn.recv(delimiter_index + 1)

                if (len(data) > 3):
                  command = data[0:4]
             else:
                  command = "error"
```

Code Chunk 4.4 Call Connection Command Receiver

This section is followed by a series of IF statements, which appropriately dispatch execution based on the command. A command of “error” does nothing. Note that if the data received over `conn` is not properly formatted, `delimiter_index` will be -1, and thus 0 characters will be received.

4.1.1.4 Two-Tiered Design

The design of chapter 3 specified that the implementation of the voice transmission module would be so flexible that it could be configured into a two-tiered hierarchy for improved performance deployment in disadvantaged networks. While the details of this are spelled out clearly in the design section, the implementation did not come to fruition as of the writing of this thesis.

This is the primary area for future work that we leave open. YATE is flexible enough that implementation is feasible. The required additions include a configuration parameter that allow nodes to function as aggregators and an algorithm that will allow deterministic aggregator selection within subnets. After this, the aggregator functionality is needed, which is simply a different set of voice streams than P2P nodes use.

Specifically, P2P nodes route two types of voice streams: the mixed stream, which is transmitted to the local user, and the locally produced stream, which is transmitted to all its peers. The aggregator has a small difference. It transmits the following streams:

- The combined stream to its local user.
- Its own stream mixed with the other aggregator to its subnet peers
- The mix from itself and its subnet peers to the other aggregator.

With these modifications, the implementation would be complete and versatility. However, this is left incomplete for now.

4.1.2 Membership Management with Jabber and Pidgin

Having defined a way to communicate with the call connection module, we can now implement a membership management module that communicates effectively along this interface.

Design goals for the membership management module of our system are simple: maintain the set of users in the conference group and communicate changes in this information to the voice data module using the interface provided. The set of users includes the usernames and IP addresses of each conference participant. For a prototype implementation such as this one, it makes sense to piggy-back on existing software infrastructure that implements this functionality but can be modified to use the interface defined in the previous section.

Jabber [6] is an open-source protocol for instant messaging and presence information exchange that is primarily used in text chat applications. Consequently, the protocol supports exchange of group membership information, which is used to manage group chat sessions. It is a client-server based technology in that end nodes communicate with a server rather than each other as they exchange information. Communication between client and server can be configured to use Transport Layer Security (TLS) [27] for confidentiality.

Because Jabber is an open protocol, our system could use the protocol to implement its own server and client software. However, open source server software already exists, as does highly extensible client software. Thus, to reduce implementation time, we choose to use the open source Jabber client Pidgin to communicate membership information among the conference members. We implemented a Pidgin plug-in that will transmit this information to the YATE module. Pidgin's intended use of the Jabber protocol is instant messaging.

It is important to note that including a client-server membership module does not compromise the design goals of the system, which call for end-to-end encryption and peer-to-peer voice data routing. This module deals only with membership information, which abides by relaxed goals for confidentiality; thus, such information can be transmitted through a server which is not part of the trusted group. By using Jabber's TLS capabilities, the information remains confidential to an expanded group, including the conference membership and the server.

In a system with strict end-to-end confidentiality requirements for membership, the designer could implement a module that transfers this information directly between peers. Providing this flexibility is the reason why modules are decoupled from one another.

4.1.2.1 Membership Information Exchange with Jabber Protocol

Conveniently, the Jabber protocol provides support for conference membership management. Jabber clients can send join or leave messages to conferences hosted by Jabber servers. The server maintains this information, and distributes any changes to clients that have joined the group. The clients can receive changes in presence information for the conference group [46].

The only additional functionality needed is communication between the client side software, and the voice data module. Obviously, this functionality is not pre-loaded in Pidgin, so our system includes a plug-in that implements it.

4.1.2.2 Implementation of the Pidgin Plug-in

Pidgin is highly extensible in that it allows developers to code C plug-ins spanning a wide range of uses by using its callback system. Loaded plug-ins have the ability to register callbacks with the Pidgin system that are triggered whenever a specific event occurs. The triggering of a plug-in results in a function call. Possible callbacks include such instant message and chat session related events as: `sending-im-msg`, `received-chat-msg`, and `conversation-created`. A complete list can be found at http://developer.pidgin.im.

For membership management, the only relevant events occur when a user joins or leaves the conference. We use Pidgin's group chat sessions as conference membership lists. In order to join or leave the group for the voice conference, users must join or leave a pre-designated group chat. The plug-in acts on these events using callbacks, and notifies

YATE of the changes.

It must also notify YATE of the IP address of the users involved, so that YATE can create the data connections. For this purpose we overload the group chat message sending functionality. Because each end user is unaware of the location of the other end-users (as this is a client/server system), users must overtly inform other users of their locations. The plug-in accomplishes this by sending a string containing its local IP address as a group chat instant message. Code to determine the local IP address is shown in Code Chunk 4.5. This code is for windows version, and uses functions `gethostname` and `gethostbyname` from the winsock library.

```
int find_local_ip(char * ip)
{
    char ac[MAXHOSTNAMELEN];
      if (gethostname(ac, sizeof(ac)) == SOCKET_ERROR) {
        return 1;
    }

    struct hostent *phe = gethostbyname(ac);
    if (phe == 0) {
        return 1;
    }

    int i = 0;
    while (phe->h_addr_list[i] != 0){
        struct in_addr addr;
        memcpy(&addr, phe->h_addr_list[i], sizeof(struct in_addr));
        sprintf(ip, inet_ntoa(addr));

      i++;
    }

    return 0;
}
```

Code Chunk 4.5 Code for Finding Local IP Address

The string ultimately sent to the group contains tags that earmark it as an IP address. The plug-in must appropriately handle such earmarked messages from other users by storing the IP address and preventing the message from being displayed. If someone joins the conference without running the plug-in, her/his chat window will display plaintext messages containing the other members' IP address information. We consider this a harmless side effect for this prototype. In this way we distribute the appropriate information, while making the system relatively robust. However, a more graceful solution exists, devised by Joe Cooley at Lincoln Lab[2].

---

[2] IP address information could be distributed in the *id* attribute of a Jabber *span* tag, in which the actual message enclosed by the tags is "VoIP Info." The plug-in could handle such messages by extracting the relevant information and concealing the message from the user's conversation window. Users without the plug-in will only print "VoIP Info" instead of all the information.

Furthermore, it is logical to authenticate IP address information in a future system. Even if membership information is not considered confidential, the IP addresses used are important for keeping the voice data confidential. Thus, users should only accept IP to username bindings from trusted users. Otherwise, users could be tricked into transmitting voice data to the wrong source. A system under development at Lincoln Lab, as part of the same overall project, allows authenticated IP address dissemination. The extensibility of our prototype and simplicity of component interfaces will allow this system to be integrated easily upon its completion.

With this in mind, we are interested in 5 specific callbacks: `chat-buddy-joined`, `chat-buddy-left`, `chat-joined`, `chat-left`, and `receiving-chat-msg`. Using the protocol defined in the interface for call connection module, the basic algorithms for each call back are as follows:

```
receiving-chat-msg
     if msg marked as IP address of sender
           if message is new
               send Bind sender To IP address to YATE
               send Join sender to YATE
     Do not display in chat session window
     Else display in chat session window

chat-joined
     send YATE: Self username msg
     send to group chat: tagged IP address
         for each username in group
             send Add username to YATE encryption group

chat-buddy-joined
     send to YATE: Add username to YATE encryption group

chat-buddy-left
     send Rem- username to YATE encryption group
     send Leav username to YATE

chat-left
     send Clos to YATE
```

Code Chunk 4.6 Pidgin Callback Pseudocode

The pseudocode is not much different than the actual code. Code Chunk 4.7 shows the chat-buddy-left callback implemented.

```
static void chat_buddy_left(PurpleConversation *conv, const char *
name, const char *reason, int sock) {
  char buf[strlen(LEAVE_MSG)+strlen(name)+1];
```

```
  char buddy_left_msg[strlen(REMOVE_MSG)+strlen(name)+1];

  sprintf(buf, "%s%s%s", LEAVE_MSG, name, "\n");
  sprintf(buddy_left_msg, "%s%s%s", REMOVE_MSG, name, "\n");

  send(sock, buf , strlen(buf), 0);
  send(sock, buddy_left_msg, strlen(buddy_left_msg), 0);
}
```

Code Chunk 4.7 Implementation of Chat-Buddy-Left Callback

Note that when a user joins a group chat, she/he receives from the server the entire transcript thus far of the group chat, assuming the chat history option is enabled. Thus, the receiving-chat-msg callback is triggered once for each message that has been sent to the group. Consequently, the function call for this trigger exits without communicating anything to YATE when the message in question is old, i.e. from the conference history. Otherwise, each time a user joins, everyone in the conference would attempt to create a connection with her/him, while she/he would create a separate connection with each other user. Because connections are two-way automatically, an extra connection creates a redundancy which will overload the system and create loopback effects. We solve this problem by only allowing existing users to connect to new users. The user that creates the connection also maintains it and is responsible for tearing it down when one of the parties leave.

Even though only one user maintains a given connection, all users must maintain an accurate list of current group members. Thus, when a user joins, the Pidgin plug-in tells YATE to add all of the group's usernames to the list. Whenever new users come or go, the plug-in updates YATE.

As far as voice connections are concerned, Pidgin tells YATE to create a new connection when a user joins, and remove the connection when the user leaves. It is also important that an exiting user tells YATE to close all of its connections. This is because the YATE script only manages the connections that is has created itself. Thus, if user A joins before user B, user A will create the outgoing connection to user B. If user A also leaves before user B, Pidgin on user B will tell YATE to remove the connection, but YATE will be unable because it did not create the connection. Thus, user A must tear down all connections it has created.

Theoretically, a malicious user could continue to maintain its connections after it has left the conference in an attempt to eavesdrop without others knowing. This is why other conference participants also send a `remove` message to YATE, telling it to remove the user from the encryption group when she/he leaves. Thus, the subversive user will receive encrypted voice data, for which she/he does not have the decryption key. Consider the following exchange, in which Mallory tries to eavesdrop on a sensitive conversation between Alice and Bob.

Alice Connects to Mallory

Alice: Hello, Mallory
Mallory: Do you have any insider stock tips for me?
Alice: You know I can't give you that

Bob joins conference group in pidgin
Alice Connects to Bob
Mallory Connects to Bob
New Key issued for new group

Mallory: Hi Bob, planning any major acquisitions?
Bob: I can't disclose that information in front of you Mal, maybe you should go.
Mallory: Ok, bye

Mallory leaves pidgin conference group, secretly maintains voice connection to Bob
New Key issued for new group

Bob: Ok let's talk about secret wall street stuff.
Alice: Ok good, Mallory can't understand us even though he's still listening, because we switched keys!

Mallory is foiled

As this conversation goes, the encryption module protects against eavesdroppers. Now all we have to do is implement it.

4.1.3 Security with PKGE

Now that we've got the vital links set up in our system, we proceed to adding what we came here for: the encryption. Carrying over from the design section, we have elected to use the Public Key Group Encryption (PKGE) libraries from Lincoln Lab. For our purposes, there are only a few PKGE functions and data types that are relevant.

`pkge_seal` encrypts and signs a data buffer and includes any necessary headers that the protocol calls for. `pkge_unseal` does the opposite, decrypting and signing a buffer that has been sealed. We also deal with `group_t` objects, which contain lists of usernames, that are passed to the seal and unseal functions so that they use appropriate keys. `pkge_group_add` and `pkge_group_remove` modify the group objects.

Logically, the seal call must take place after any lossy audio compression, otherwise, the sealed message would be corrupted and unrecoverable. Similarly, it is easier to place the seal call before any RTP headers are added to the data buffer, so that these headers are recoverable by the lower level RTP receivers on the other end. Naturally then, we choose to seal outgoing messages in the RTP channel module just before the RTP headers are added.

```
if (!m_wrap->rtp()->local() && DataConsumer::m_have_yate_group) {
  if (DataConsumer::s_use_previous_packet) {

      m_wrap->rtp()->rtpSendData(false,tStamp,(char *)
                                 DataConsumer::s_previous_block.data,
                                 DataConsumer::s_previous_block.len);
  }

  else {
      data_t original_msg;
      data_t sealed_data;
      original_msg.data = (byte_t*) data.data();
      original_msg.len = sz;
      sealed_data = data_copy(original_msg);
      int r = pkge_seal(DataConsumer::m_yate_group, &sealed_data);
      DataConsumer::s_previous_block = data_copy(sealed_data);
      m_wrap->rtp()->rtpSendData(false,tStamp,(char *)
                                 sealed_data.data,sealed_data.len);
      free_data(&sealed_data);
  }
}
```

Code Chunk 4.8 Sealing Outgoing Audio

In Code Chunk 4.4, the else statement contains code to copy over `data` into a data structure supported by PKGE. This aptly named parameter, `sealed_data`, is passed to `pkge_seal` before being sent over the wire, then freed. Note also, that this case is only accessed when `m_wrap->rtp()->local()` is false. This is simply an indicator we added that returns true when we are working with the local end user (to whom we send plaintext data).

Furthermore, there are references to `DataConsumer` static `s_previous_block` and `s_use_previous_block` variables. `s_use_previous_block` indicates whether or not the necessary sealed, packetized data has already been computed, and if it has, `s_previous_block` maintain the data itself. Such a situation occurs when there are 3 or more confernece participants and this code segment is executed three times. The data produced by the local endpoint should be passed to the other two end users in identical form. Thus, this data is saved in `s_previous_block` after it is computed for the first time, and its presence is flagged in `s_use_previous_block`. We set the values for the flag in the conference module when the data is being computed, and free the data when the flag changes from true to false. The point of these variables is to take advantage of the fact that messages sent to different parties in the conference use the same keys. This efficiency is one of the reasons we adopted the design in section 3.

Assignment and updating these variables would have been in Code Chunk 4.2, which precedes the seal calls in the execution stack. However, we omit these few lines of code due to lack of relevance in that section.

Note also the reference to `m_yate_group`. For our system, we require a group object that is referenced whenever data is encrypted. It is this `group_t` object that is modified as the telephony engine receives add and remove user messages from the script described in section 4.1.1.2. The group is created when the control script passes a yate.self message to the engine. The following code in Code Chunk 4.3 is run. The `s_pin` variable stores the users pin number, and is accessed in a configuration file. The yate.self message is passed with the local username as a parameter, `self`, which is used to create `m_yate_group`, after any old data is freed. The PKGE protocol that is used is specified in the call to `pkge_init`.

```
void ConfRoom::addSelf(String &self) {
      pin_t * pin = new pin_t;
      pin->data = (byte_t *) strdup(s_pin.c_str());
      pin->len = strlen(s_pin.c_str());
      user_t* user = new user_t;

      if (DataConsumer::m_have_yate_group)
          pkge_group_free(DataConsumer::m_yate_group);

      pkge_user_load(&DataConsumer::m_yate_self, s_system.c_str(),
                     self.c_str(), pin);

      pkge_init(s_system.c_str(), PKGE_PROTOCOL_LEAN);

      DataConsumer::m_yate_group =
                    pkge_group_new(DataConsumer::m_yate_self);
      DataConsumer::m_have_yate_group = true;

      delete pin;
}
```

Code Chunk 4.9 Creating an Encryption Group

The code for adding and removing members is omitted due to its simplicity and similarity to Code Chunk 4.9

Based on the position of the seal calls, unseal calls must be placed as soon as the RTP headers are stripped from incoming data. As always, this does not apply to the local user. As Code Chunk 4.10 shows, the manner of unsealing the data is similar to the mannner of sealing the data.

```
if (!m_local && DataConsumer::m_have_yate_group) {
      DataBlock unsealed_block;
      data_t sealed_data;
      sealed_data.data = (byte_t *) data;
      sealed_data.len = len;
      data_t unsealed_data = data_copy(sealed_data);

      s_cert_Mutex.lock(-1);

      group_t decrypt_group =
              pkge_group_new(DataConsumer::m_yate_self);

      int r = pkge_unseal(decrypt_group, &unsealed_data);
```

```
      pkge_group_free(decrypt_group);

      s_cert_Mutex.unlock();

      unsealed_block.assign((void*)
      unsealed_data.data,unsealed_data.len, false);
      source->Forward(unsealed_block, timestamp);
      unsealed_block.clear(false);
    }
else source->Forward(block,timestamp);
```

Code Chunk 4.10 Code to Unseal Incoming Data

An important subteley to notice is the use of `s_cert_Mutex`. This static variable refers to a mutex which serializes decryption calls. YATE launches a new thread whenever a packet is received on the RTP port. Consequently, it is possible for multiple concurrent threads to run the decryption Code Chunk 4.10. The mutex ensures that PKGE methods are not invoked by separate threads at the same time. Because PKGE does not protect data for multithreading, multiple calls to the same PKGE method can cause unexpected behavior. Appropriate data protection is a notable feature for consideration in the next generation system.

Similarly, the decryption group `decrypt_group` should not be accessed by two threads at the same time, otherwise one will see garbage data. Thus, we statically store only the username needed to define the group (which for decryption is a single group member, not the entire group) and create a new group every time the code is invoked. We free the group, unlock the mutex, and forward the freshly decrypted data on its merry way.

4.1.4 Audio Playback with X-Lite

The role of the audio playback software is to establish a connection with the voice data module and playback audio to the end user. The physical analog of the playback module is a telephone. Because this module is not directly related to the most important design goals of the system (encryption, peer-to-peer audio transmission, etc), this module should be chosen first based on how well it integrates with the modules that do achieve these goals and second based on how usable it is.

Specifically, the audio playback module must communicate with the voice data module effectively. YATE is chosen as the voice data module for the above reasons. YATE can initiate conversations with a soft-phone using any open call protocol including SIP and H.323, and can transmit the voice data using RTP. Thus the playback software must have support for these protocols. It must also run on the required platforms: Windows, Mac OS, and Linux.

We use X-Lite 0, a freeware soft-phone product for Windows, Linux, and Mac OS that provides robust, reliable SIP calling and RTP conversations. It plays back audio transmitted in a variety of formats, including the mu-law compressed data that is delivered by YATE.

Because there are many free soft-phones capable of meeting these preliminary requirements, usability was also an important selection consideration. In fact, the obvious choice for this function would have been YATE's own client, but a cursory examination found its usability to be lacking. As the market's leading free SIP based soft-phone 0, X-Lite's popularity among consumers suggests its ease of use. More concretely, it can be quickly downloaded, configured and installed in 5 steps that do not take more than 5 minutes on modern machine with a broadband connection (see Appendix C). It takes exactly two clicks for a properly configured version of X-Lite to interface properly with the YATE call handling module. On a more subjective note, it has an aesthetically pleasing UI (see Figure 4.3).

Figure 4.3 X-Lite User Interface

Moreover, the minimum system requirements for X-Lite are 256 Mb of main memory on a Pentium III with clock speed of 700 MHz. This is years behind the current upper bound on computational power and suggests that X-Lite will not hog machine resources that are needed by the more laborious processes of the system.

Several other soft-phones were considered, but were rejected for reasons including platform dependence and configuration difficulty. Such phones included KPhone, which only runs on Linux and Cisco IP Communicator, which doesn't support open protocols such as SIP. It is important to note that the list of soft-phone clients is long and growing, making it difficult or impossible to evaluate all of them. It is likely that there are other phone clients that would have been as effective as or more effective than X-Lite at meeting our goals. However, X-Lite met requirements better than anything else examined.

4.2 Summary

In this section, we have reviewed the low level details of our secure conference implementation. All three major modules (voice, membership, security) were completed. The only exception is the omission of the two-tiered design for voice transmission. Otherwise, the voice module handles call connection and voice transmission in a peer-to-peer fashion, as well as audio playback and compression from off the shelf components. Membership management was implemented with a Pidgin plug-in that uses the existing Jabber protocol infrastructure. Of course, we make use of PKGE for security, as it achieves the design goals and demonstrates its usability in a real-time communication system.

In the next section, we will examine exactly how well the design goals are met. Where possible, quantitative evaluation is employed.

5 Evaluation & Testing

Now that development of the system is complete, we proceed to create a testing framework. We will use this to evaluate the system against design goals set in section 3. The points of examination are functionality, security, quality, and usability.

Evaluation and tests will show that most major goals for functionality, security, call quality, and usability have been achieved. The overhead added by PKGE is small for packet latency, indicating that the system's ability to meet quality standards is largely dependent on the network of deployment. Similarly, byte overhead does not contribute significantly to packet loss. However, system functionality lacks implementation of a versatile two tier design, while usability lacks simple installation and startup. Design goals that are not achieved are left for future work.

5.1 Functionality

The functional requirements for the system included creating conference calls that could support multiple speakers, occasional membership changes, consistent availability, and at least six participants. In short, each of these has been achieved.

√ The design is such that each endpoint transmits its voice data to each other endpoint, which mixes all data into a single stream. This is fully implemented. This distinguishes our system as a single conference, rather than multiple phone calls. In the two-tiered model design, aggregators perform some of this mixing.

√ Membership management with Pidgin and Jabber allows membership changes to be disseminated to each endpoint so that users can join and leave the conference. As membership changes, all users are kept up to date on the conference group so that the appropriate encryption keys are used.

√ Because each of the endpoints is self-sufficient, conferences do not depend on any one user for persistence. If the membership server goes down, join and leave functionality is lost, but the current members can continue conversing.

√ Ability to maintain six participants depends on the links used for voice transmission. However, chapter 5.3 provides evidence that the system scales well with additional users. Conferences of six are possible if the network has enough bandwidth, as prescribed in 5.3.

The one functional requirement omitted from implementation is the two-tiered model for deployment in disadvantaged networks. We have, however, designed the solution in chapter three, and specified how exactly it should be implemented.

5.2 End-to-End Security

In this section we evaluate the security design goals. Specifically, we sought to provide confidentiality, authentication and integrity end-to-end. The security is dependent on the security strength of PKGE, while the achieving it from end-to-end depends on the implementation. The peer-to-peer model and two-tiered model are both end-to-end secure.

Confidentiality of messages during voice conferences depends on the algorithm used for encryption and the method of distributing keys. In short, AES in CBC mode with 256 bit keys is used, which is the standard for symmetric encryption adopted by the U.S. Government and the National Institute of Standards and Technology (NIST). These keys are unique for each set of participants, providing forward/backward secrecy for the conference. Perfect forward secrecy results from reissuing keys for each epoch. However, these communications are only really secret if the key distribution is secure. The key distribution is done using BGW Broadcast Encryption. Thus, the secrecy of messages reduces to the strength of the BGW scheme.

Authentication and integrity verification are also required by design, and achieved by PKGE. For the Lean protocol, this is achieved through a keyed SHA1 hash, or an HMAC. Authentication is thus dependent on the strength of the SHA1 hash, developed by the National Security Agency (NSA) and considered a standard by the NIST. Other protocols, as well as key distribution, use an elliptic curve digital signature algorithm (ECDSA).

As for ensuring that the security is implemented from end-to-end, we can conclude that this is the case in the peer-to-peer model. The P2P model was designed specifically for end-to-end security. The model invokes seal immediately after voice production and compression, before transmission. Unseal is called right before decompression at the end of the wire.

In the two-tiered model, performance necessities force us to unseal the messages at the aggregator nodes when there are multiple concurrent speakers. However, the aggregators that unseal and reseal the messages in the middle of transmission are trusted conference participants, thus part of the trust model. Thus, this does not constitute a departure from end-to-end security.

Thus the system meets the security goals of 2.2.2.

5.3 Call Quality

This section details a number of rigorous tests to determine the call quality on the test-bed, and extrapolate the call quality in several disadvantaged networks. Using the fastest, leanest protocol, total quality performance can be achieved in Ethernet connected subnets and across TCDL links. Tolerable quality is achieved with Inmarsat and Connexion satellite links, while Iridium satellite links cannot achieve any reasonable quality.

Recall the standards for call quality we set are based on packet loss and packet delay, both measured from end-to-end. We choose to evaluate quality based on those simple two factors outline in 2.2.3. Specifically, total quality is considered less than 100 ms of end-to-end delay with 0% packet loss, while tolerable quality is up to 400 ms of delay with up to 1% packet loss.

5.3.1 Call parameters

I will calculate packet loss and packet delay to determine call quality while varying several factors to determine the scope of appropriate use for the system. The parameters to vary are:

1. *Size of Conference* – As more users join the conference, we expect the degradation in call quality to be minimal while there is adequate bandwidth. Based on the PKGE protocols, adding additional users does not significantly increase computational overhead. Because messages are transmitted to each individual user, required bandwidth increases linearly with each additional participant.

2. *Packet Size* – YATE packetizes audio data into 160 byte chunks by default. We will compare results for 80 byte packets as well to determine if packet size has any effect. The difference in computation overhead should be negligible for all protocols except Lean, because the bottleneck operations in PKGE run in constant time with respect to message size, except during keying. Specifically, the ECDSA is performed on a constant size message hash. When the HMAC is used instead of the ECDSA, the operation runs in linear time.

3. *PKGE Protocol* – It should be clear that the Lean protocol will have the lowest time cost and smallest packet size. However, for the sake of comparison we include some measurements from the other protocols as well. The increased latency in the other protocols pays off in unreliable data networks because neither the Stateless protocol nor Sessions protocol requires guaranteed packet delivery. Thus it is important to experiment with them and determine their usability. Nevertheless, we expect the Lean protocol to be significantly faster due the decrease in packet size overhead, and the difference in computation time between a message authentication code and a digital signature. Moreover, we expect Stateless to be the slowest due to larger packet sizes, and increased computation for each packet, as keys are computed every single time.

4. *Membership Changes* – While members are joining and leaving, we expect significant degradation in quality both from packet loss and latency. Increased latency will occur as new keys are computed and distributed. Packet loss will increase because packets distributed at this time will have

higher latency (may be dropped by buffer) and larger size. Also, in order for packets to be properly signed, encrypted, and decrypted, the conference membership changes must be synchronized at different endpoints. For example, when a new user joins, if the current speaker transmits voice data but has not yet updated the group membership variable, the new user will have to drop the packet. As this is unlikely, a few packets will probably be dropped.

5.3.2 Network Parameters

Tests were carried out on the Lincoln Lab "netlab" test-bed consisting of 36 Intel Xeon 2.4GHz quad core 2GB machines running Debian Linux. The computers are connected with a 100 Mbit/s Ethernet. Four of the machines are used for the tests.

5.3.3 Methodology

To measure packet latencies, we calculate two quantities. First, we calculate the time between audio production and audio transmission. This includes all computation on the speaker side of a conversation, notably sealing the message. Everything else the VoIP software does, such as audio compression is also included. Secondly, we measure the latency on the other side of the wire. That is, the time elapsed between packet receipt and audio playback. This includes unsealing the data[3].

I chose to measure these quantities instead of directly measuring end-to-end latency because end-to-end latency depends heavily on the network on which the system is deployed. By measuring the latency on the speaker side and on the receiver side, we can determine the network requirements for the various levels of call quality.

I took measurements by inserting calls to the system clock directly in the YATE code. We invoked the `now()` method of the YATE time library, which is equivalent to calling `gettimeofday()` on a Linux machine. This returns the time in microseconds, although the precision is on the order of 10 microseconds.

Packet loss is measured in a similar fashion. Because the network test-bed used was connected with at 100 Mbits/s Ethernet, the rate of packet loss on the network was negligible. Thus, we could measure end-to-end and know that any loss was based on the system, and not the network. The same backwards computation for acceptable networks can then be applied. Specifically, we insert a counter at both the sending and receiving points in the YATE code. This way, during a test call, every party counts every packet sent out, and every party counts every packet played back. The difference between the two is the packet loss. For example, see Code Chunk 5.1, which shows how simple it is

[3] The version tested included a bug in the PKGE code that caused additional computational latency, which averaged 5201 μs. In order to avoid repeating the extensive testing, the numbers reported in this section were calculated by subtracting this additional latency from the actual time measured.

to count received packets. The counter is placed after the unseal call. This is the last point the packet passes through at which it is reasonable for the packet to be dropped.

```
int r = pkge_unseal(decrypt_group, &unsealed_data);
if (!r) read_counter+=1;

source->Forward(unsealed_data, timestamp);
```
Code Chunk 5.1 Inserting Packet Counter in Receiver

Using this test-ready code, we carried out a number of experiments to examine the quantities mentioned in the previous section. Let's see which predictions held up on the test-bed.

5.3.4 The Verdict

The following tests show very promising results for PKGE, as we learn that the Lean protocol adds very little latency to the VoIP system and adds very little byte overhead. The entire send side latency is on the order of 100 microseconds, and 66 bytes of overhead is added with Lean PGKE.

5.3.4.1 Protocol Latency Comparison

Because of the performance superiority of the Lean protocol, it is used for most of the tests. However, as a quick sanity check, we compare the sender side latency in a two-party conversation using 160 byte packets. Figure 5.1 shows the latency for the first 15 messages in the conversation for each of three protocols: Stateless, Sessions (Pessimistic), and Lean. Without any mathematical analysis, it is clear that Lean blows the competition out of the water, having negligible seal latency after the first message.

The nature of the protocols is such that we expect the first message in the Lean protocol to be as costly as the messages in the Stateless protocol, the message requires key generation and includes all headers. Similarly, the Sessions protocol will behave the same as Stateless until the key is verified, which occurs after three messages in this case. Otherwise, Lean is the fastest protocol.

5.3.4.2 Marginal Participant Latency

With that out of the way, let's examine something more interesting: the effect of additional parties in the conference call. Because the system is for conferencing, scalability to more than two partiers is essential. Scalability to larger groups augments the scope of applications, making the system more practical.

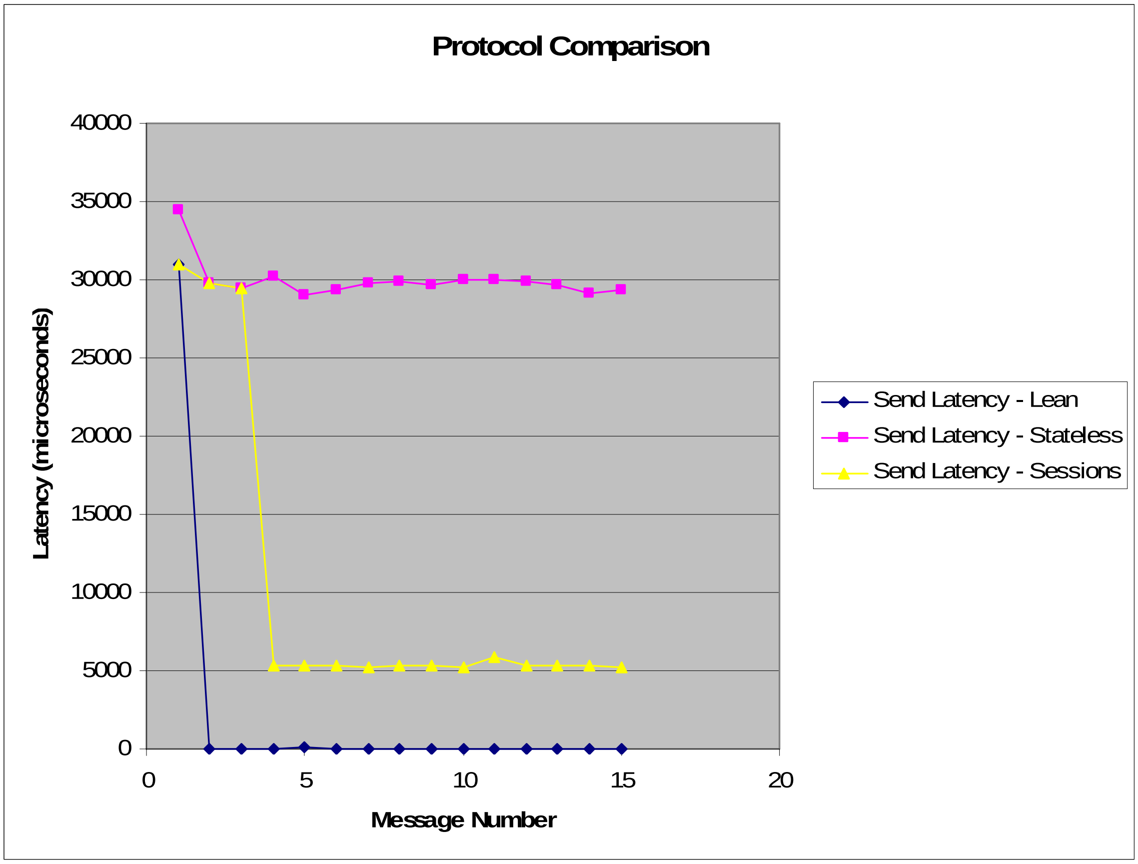

Figure 5.1 PKGE Protocol Comparison

I test this property by comparing sender side latency as the conference grows from two to four members and extrapolating to even larger groups. We use the Lean protocol with 160 byte messages. Recall from the implementation that the speaker only calls `seal` once, regardless of the number of users in the conference. The same sealed data can be transmitted to each conference user because we use group encryption. Thus, we expect the additional latency to be low; the computation necessary involves moving some data around the YATE system and back out onto the wire.

Figure 5.2 shows the time elapsed between audio production and transmission for each of the three recipients in a four party call. Note that each line does not represent additional latency, it represents total latency. As an example, for the 15^{th} message, audio is produced at time = 0, data is transmitted to the first recipient at time = 165 μs, and data is transmitted to the third recipient at time = 367 μs . The average additional latency for the second recipient is 96 microseconds, and the average additional latency for the third recipient is 91 microseconds. On this basis alone, the conference is highly scalable, confirming the predictions. Of course, we will also look at packet sizes and network requirements later on.

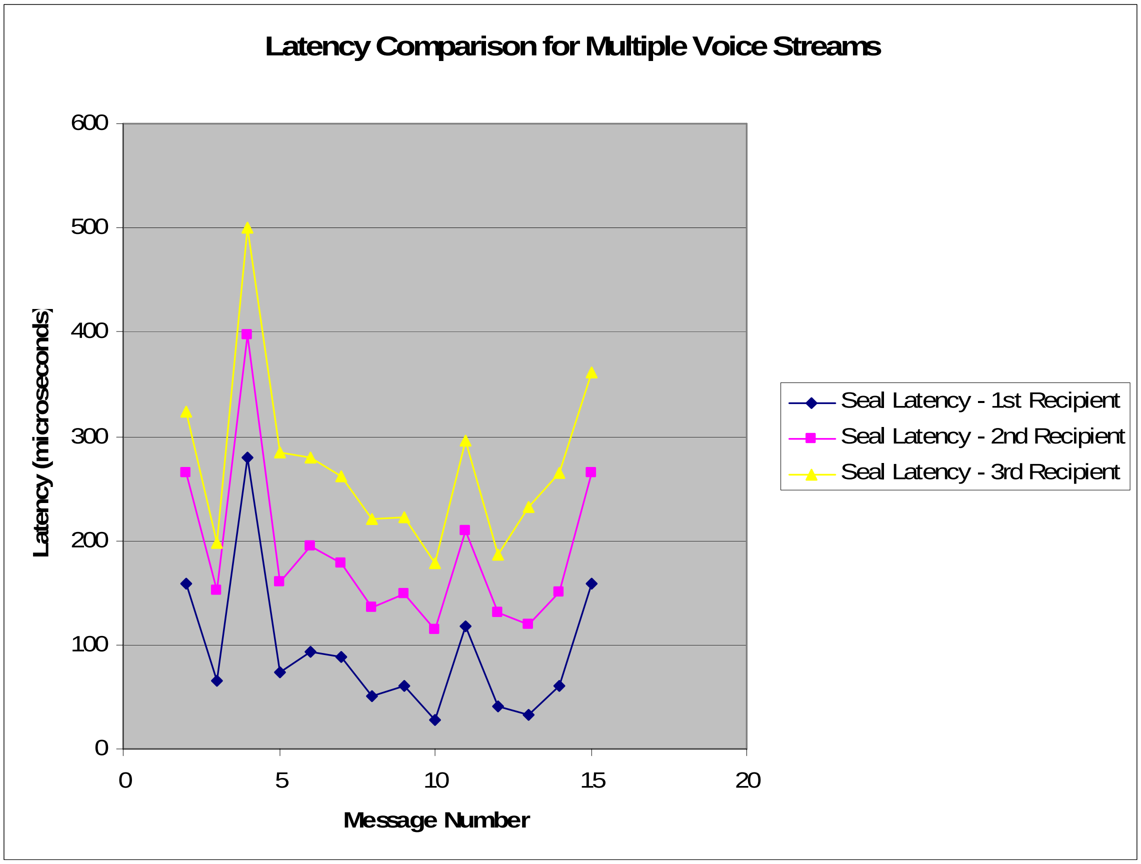

Figure 5.2 Latency for Multiple Voice Streams

5.3.4.3 Receive Side Latency

So far I've only shown results for send-side latency. We do this on the assumption that the receive-side latency is about the same, because the operations are relatively symmetric. That is, on the send side, the bottleneck is the seal and the only other significant operation is the compression. On the receive side, the bottleneck is unseal, and the significant operations are decompression and audio mixing. Audio mixing is uncommon, so its effect is small and less important than other processes. Before jumping to any wild conclusions, we test the receiver side latency for the Lean protocol, with 160 byte messages. Figure 5.3 shows that receiver side time varies from about 40 µs to 180 µs for the 15 messages measured. This is close enough to the send side latencies of 20-280 µs that we can assume that they will be similar in future cases. This graph leaves out the initial message in a conversation, which carries an encapsulation of a group key to the new group; this message is larger and takes longer to transmit than subsequent messages.

5.3.4.4 Join Message Latency

The initial messages in the conversation, as well as messages that coincide with the joining of a new group member will necessarily be much larger and slower for the Lean protocol. In fact, they should be equivalent to the messages sent during the course of a

Stateless protocol conversation. This is because the first message after a new group member joins must contain the headers necessary for re-calculation of the group encryption key. When the message is sealed, the sender spends time calculating that key and the headers. Thus, we expect ‘join’ messages to have a latency in the 30 ms range, as the Stateless protocol does in Figure 5.1. The receiver of the join message also endures additional latency by computing the computing the decryption key from the headers and verifying authenticity and integrity. This process is roughly equivalent in time used, and requires 30-40 ms.

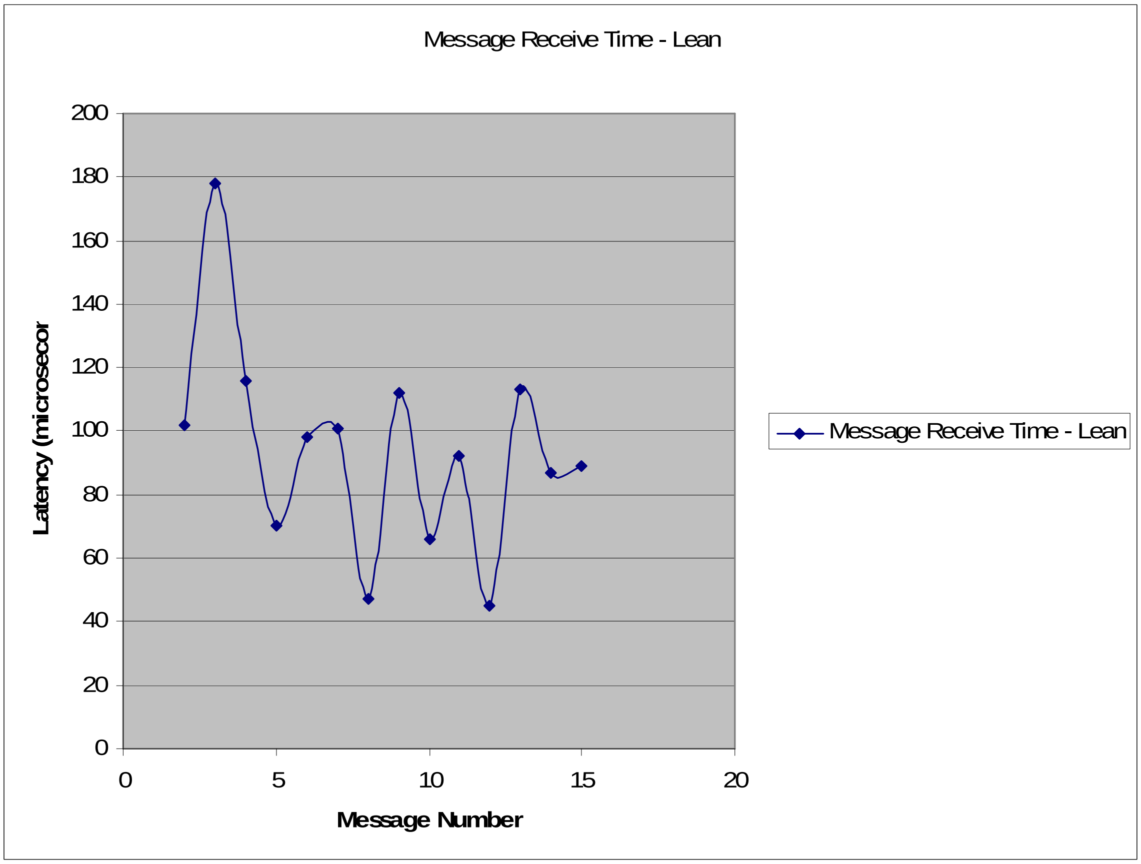

Figure 5.3 Receive-Side Latency

Figure 5.4 shows the latencies during a Lean protocol conversation using 160-byte messages. Messages 1, 16, and 31 are ‘join’ messages. This conversation data corroborates the predictions, as the ‘join’ messages take about 30 ms to compute before sending. Note also that the ‘join’ messages appear to be slower with each additional party, starting at 31.1 ms and going to 32.8 ms for the last one. This trend is reasonable because each additional party increases the computation cost for the headers that are transmitted with the message.

5.3.4.5 Concurrent Speech Cost

Thus far, we have only tested the common communication case in which only a single user is speaking at a time. It is rare that there are multiple concurrent speakers, because this makes conversation difficult. However, it is conceivable that certain scenarios (such as a debate) will involve numerous users trying to squeeze in their two cents all at the same time. To determine how this affects packet latencies, we tested a three person conference, using the Stateless protocol in which all three endpoints continuously transmitted audio data to one another. The results are shown in 5.5.

Interestingly enough, the send side latency appears to triple compared to the single speaker model in 5.1. This is surprising at first, because having extra speakers should not affect the encryption at all. However, some debugging revealed that the audio-sending thread is interleaved with the audio-receiving thread. Because there are three uninterrupted speakers, each user receives two packets for every packet it sends. Consequently, the audio-receiving thread, which is about as costly as the transmission thread, is launched twice while the transmitting thread is running. This causes the slowdowns. Note also that there are a few data points that are significantly faster (by 30 ms). These occur when only a single packet is received during the run of the transmitting thread. This is possible because the different speakers are not perfectly synchronized and the network does not necessarily deliver every packet with the same speed. In other words, there is some random variation between the different users.

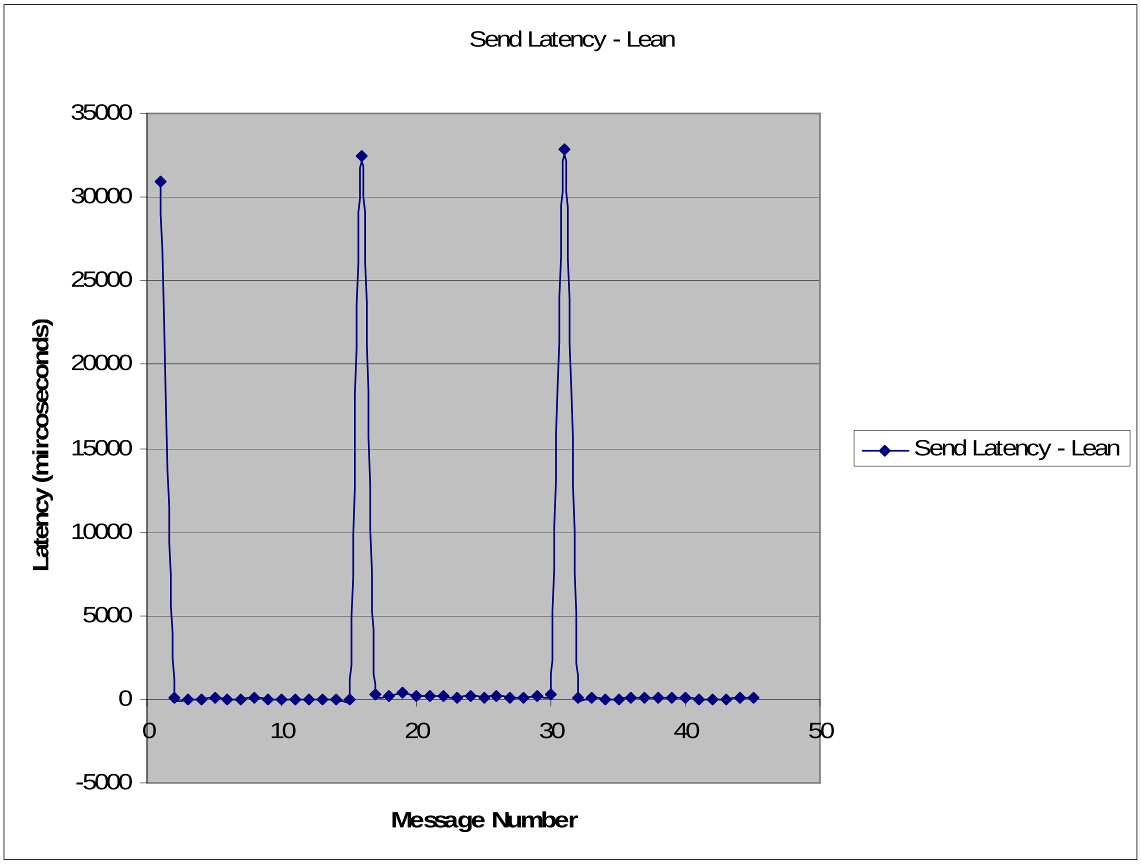

Figure 5.4 Send-Side Latency with Join Messages

5.3.4.6 Effect of Message Size on Latency

There is one more comparison dimension left: message size. For the Lean protocol, messages during the conversation are encrypted using AES in CBC mode. The authentication algorithm in this protocol is a keyed SHA1 hash. Both of these algorithms are lightning quick, and should take a few microseconds to compute (see 3.2.1). The time complexity of both algorithms is linear, but computation time is on the order of microseconds, so we expect only small differences in the send latency tests. This is confirmed in Figure 5.6, which shows the comparison in computation time for 80 byte messages and 160 byte messages. 160 is consistently slower by about 50 microseconds, most likely because of the slight slow down from the two linear time algorithms.

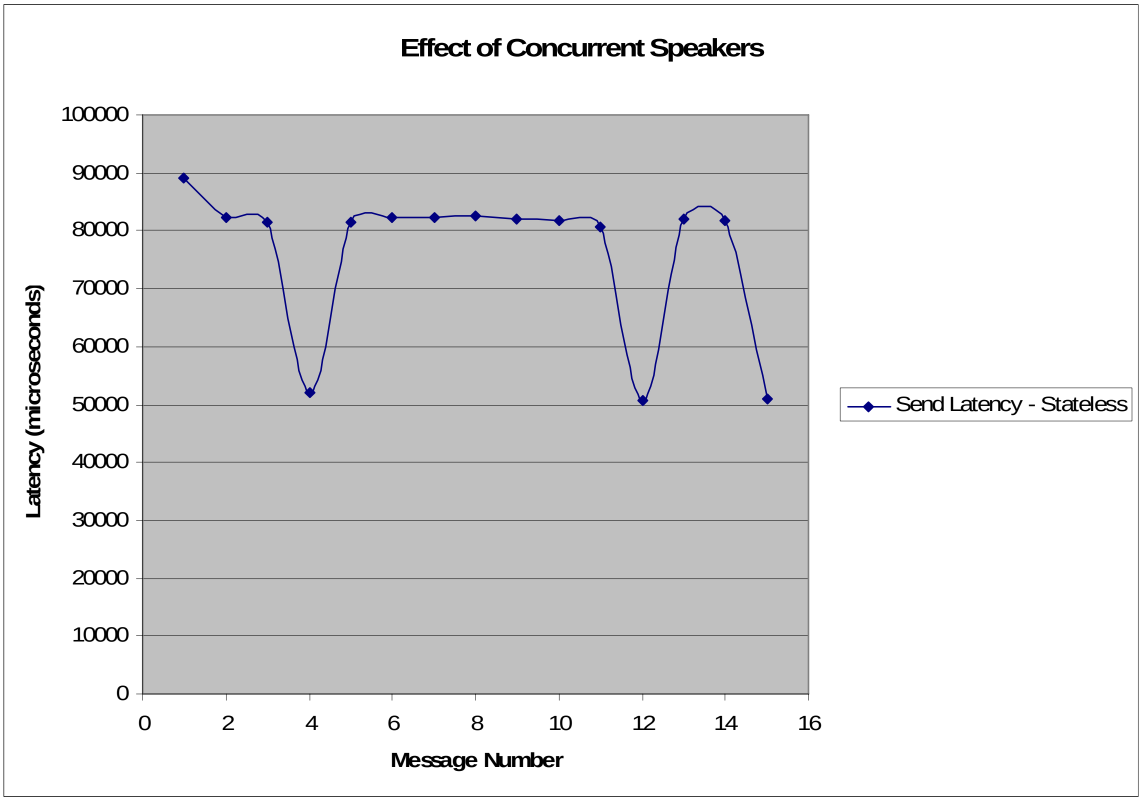

Figure 5.5 Effect of Concurrent Speakers

Along the same dimension as message size, we also look at the actual size of the sealed message when it is sent out onto the wire. This is important for networks that have stricter bandwidth constraints than our test-bed. Table 5.1 details this information, showing the size of messages transmitted for each protocol, for both original message sizes, and for both 'join' messages and 'during' messages.

Not surprisingly, the Lean protocol is the leanest. The authenticating HMAC is smaller than the digital signature used in all the other protocols. Actually, the protocol uses only four bytes of the HMAC, while the digital signature adds about 60 bytes. The Stateless protocol comes in a solid last place for the during conversation messages, as it includes key calculation headers every time. However, because the Stateless protocol does not need to include add supplemental session-related information, it saves a about 40 bytes over the other three protocols when they send out 'join' messages. Concretely, state information coupled with the long digital signatures cause Sessions and Optimistic protocol 'join' messages to be the longest overall. Similarly, even though Lean 'join' messages save about 55 bytes by using HMACs for authentication, they are only about 20 bytes shorter than Stateless 'joins' because of the state information included.

This message size information will prove useful in calculating the necessary bandwidth in various usage cases.

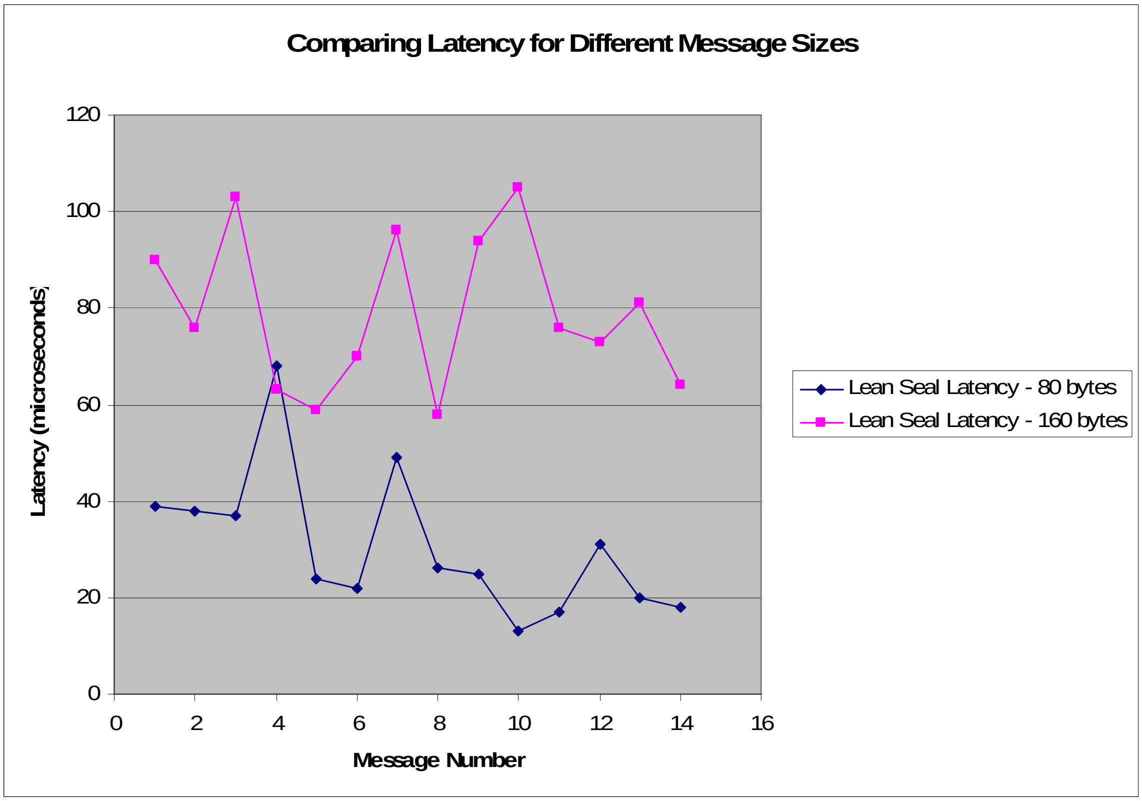

Figure 5.6 Computation Latency for Different Message Sizes

5.3.4.7 Packet Loss

Our last set of tests for comparison deals with packet loss from end-to-end. Packet loss comes from a variety of sources, including transmission errors, network congestion, buffer overruns, and encryption errors. Based on the impressive parameters of the network used, we do not expect any packet drop or bit errors in transmission. The packet loss tests confirm this, showing 0% packet loss during the course of a conference call.

What we do see, however, is that when a user joins a conference, the first few packets received are not decipherable. This is because the membership management module and voice transmission module are not completely synchronized. This causes the speaker to create a connection with the new user and transmit data before the new user's username is added to the group variable. Thus, the audio that the new user receives is not encrypted such that she/he can decipher it. We can consider this as packet loss, or we can simply define the ‘join’ to occur once the new user is added to the group.

Existing users in the conference, when a new user joins, do not miss a beat. The increased latency exists (as shown in Figure 5.4) from the added seal cost, however, no packets are lost. Packets are not dropped on the network, nor are there unseal errors, because the existing users are already part of the group.

These tests show a number of important points that we will use to make recommendations for the system’s use. First, the tests confirm the fastest protocol and the most reliable protocol are Lean and Stateless, respectively. The tests indicate what the latency costs are on the send side and receive side during conversation and during membership changes. This, along with packet size tests, allows us to calculate network requirements.

Packet Sizes For 3 Party Conference

Protocol	Data Size	Message Type	Byte Overhead
Lean	80	Join	282
	80	During	66
	160	Join	286
	160	During	66
Optimistic	80	Join	345
	80	During	141
	160	Join	345
	160	During	141
Sessions	80	Join	345
	80	During	141
	160	Join	345
	160	During	141
Stateless	80	Join	299
	80	During	299
	160	Join	300
	160	During	300

Table 5.1 Packet Sizes for Different Messages in Bytes

5.3.5 Recommendations

This section will use the test results to recommend how the system should be used and what network requirements exist. We recommend use of the Lean protocol with 160 byte packets. Bandwidth requirements vary depending on the size of the conference, and are outlined in Table 5.2. Link latency ceilings also vary with conference parameters, but a typical conference will require less than 80 ms of network latency for total quality or less than 380 ms for tolerable quality.

The Lean protocol is superior to others in terms of speed and size. The one failing it has, is that it lacks fault tolerance. We elect to use this protocol by default, only opting for the Sessions protocol if the network is unreliable, but the situation can tolerate the extra delay.

5.3.5.1 Network Requirements

Before we calculate network requirements, let's stop and review the rate of data production. The G.711 codec used creates 8 bit audio samples to send over the network. Because voice is sampled at a rate of 8000 samples per second, the byte rate is 8000 bytes per second. Refer to Appendix B for a refresher on these calculations. Thus, 160 byte packets represent 20 ms of audio and 80 byte packets represent 10 ms of audio.

Also, 160 byte packets and 80 byte packets incur roughly similar seal and unseal latencies, with the differences numbering in the tens of microseconds. The Lean protocol adds about 66 bytes of overhead to each message, regardless of size. This is a significantly larger percentage of size for 80 byte messages than 160 byte messages. Thus, to reduce network congestion and reduce link requirements we choose 160 byte messages.

Of course, by that logic, we might increase the packet size even more, say to 320 bytes. However, the size in milliseconds of voice production for each packet is essentially added to the total lag. That is, before we can encrypt a 40 ms packet, it must be produced first. We cannot seal the first part of the message before the last part is produced. Thus, increasing the message size to reduce overhead is unreasonable beyond a certain point. 160 byte messages have a small enough duration that the reduced message size overhead is acceptable. Message size remains, however, a parameter that future users can adjust to fit their needs.

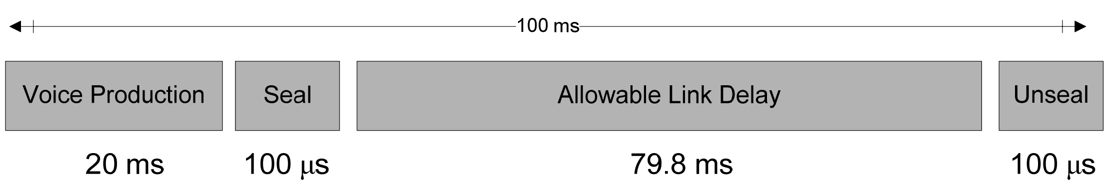

Figure 5.7 Production/Playback Delay Breakdown

With these 160 byte messages and the Lean protocol, the maximum network latency for total quality is about 80 ms. This comes from ~100 μs send side latency, ~100 μs receive side latency, and 20 ms of production time. For tolerable quality standards, we can have about 300 ms more latency in the network.

The bandwidth requirement depends on the number of participants, due to the peer-to-peer nature of routing. It also depends on the percent of time that the speaker is actually speaking, as VoIP's efficiency is due to routing only the non-negligible speech data. The bandwidth required is then 226 bytes / 20 ms * percent utilization * (number of participants -1). Table 5.2 illustrates this information. By comparison, 80 byte chunks would require about 30% more bandwidth.

Participants	Utilization	Bandwidth (bytes/s)
2	0.33	3729
2	0.66	7458
2	1	11300
3	0.33	7458
3	0.66	14916
3	1	22600
4	0.33	11187
4	0.66	22374
4	1	33900
5	0.33	14916
5	0.66	29832
5	1	45200
6	0.33	18645
6	0.66	37290
6	1	56500

Table 5.2 Bandwidth Requirements for 160-Byte Chunks

Because tolerable and total quality standards allow only 1% and 0% packet loss rates, links to support the conferences tabulated in 5.2 require roughly the full amount of bandwidth. Also note that if there are multiple concurrent speakers, the amount of bandwidth needed would increase in proportion. If the link capacity is the same size as the required capacity, concurrent speech could result in packet loss. Thus, network deployment should take into account whether this is a concern. Similarly, concurrent speech roughly doubles the seal and unseal latencies, as shown in the test section. If brief departures from quality standards are intolerable, the network must have short enough latency to stay within the bounds even during such occurrences.

Another interesting point to note is that the audio sending stack is a single thread. This means, that if the packet size is 20 ms, and the hardware cannot seal and package the chunk in under 20 ms, the next packet of audio will be late into the thread. This excess latency will accrue with each packet, causing more and more delay (Figure 5.8). The system can only "catch up" when there are pauses in speech.

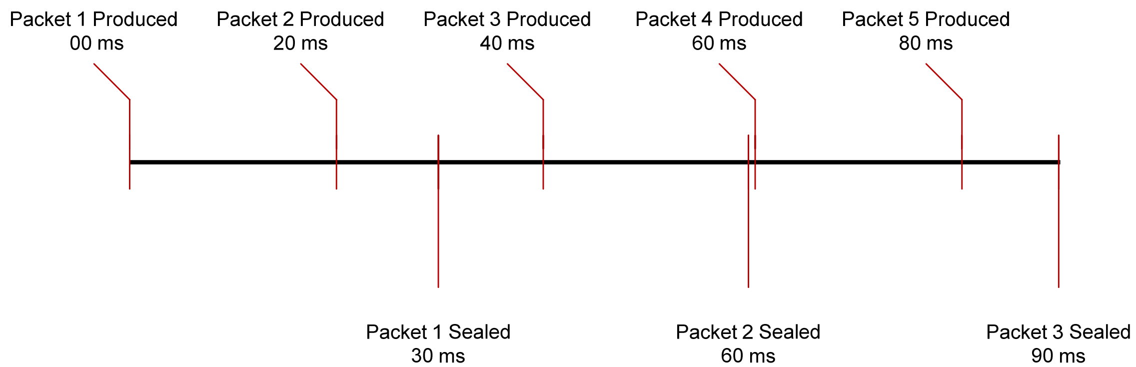

Figure 5.8 Increasing Seal Delays

This is also the case when join messages or concurrent speech increases the send latency. Normally, these are rare enough and pauses in speech are frequent enough that this will not be a problem. If the system is deployed in a unique scenario with incessant concurrent speech and/or frequent joins, it is important that the hardware can handle it fast enough that the delay does not increase. Other possible solutions to this case are described in the future work section, in which considerations for future systems are discussed.

5.3.5.2 Subnet Connection Requirements

With these calculations behind us, we can make recommendations for low performance links between subnets. Four low performance links were examined for the purpose of connecting the remote subnets: TCDL, Inmarsat, Connexion, and Iridium. TCDL, Inmarsat, and Connexion all perform well enough for tolerable quality standards. Unfortunately, the latency of the Iridium satellite connection make it impossible for use in normal conversations.

Link	Bandwidth (kbps)	Latency (ms)	Error Rate (bps)
TCDL	10000	2	10^{-7}
Connexion	128	325	10^{-6}
Inmarsat	128	325	10^{-5}
Iridium	2.4	2000	10^{-4}

Table 5.3 Connection Properties

TCDL is a line of sight, terrestrial wireless link. Inmarsat, Connxion, and Iridium are all satellite data links. Inmarsat and Connexion are privately held networks made commercially available, while Iridium is used by the US government. Table 5.3 shows the connection properties of each; borrowed from [19].

Because the subnet aggregators in the system design combine multiple streams into a single cohesive voice stream, there are a maximum of two streams on the link at any given time. This occurs when there are speakers in both subnets. Consequently, the max required bandwidth is 226/.020 * 2 * 100% utilization = 22,600 bytes per second = 180.8 kpbs. TCDL covers this easily, while Connexion and Inmarsat just miss the boat.

However, we can reasonably expect that such high bandwidth scenarios are very rare. When they occur, they are quickly ended. Moreover, when they occur, speech is often inaudible anyway, because there are multiple speakers. Thus, dropped packets only degrade speech that is already unusable. When there is only one speaker, and the utilization is under 100%, the required bandwidth is less that 90.4 kbps. Connexion and Inmarsat can make this grade; Iridium is nowhere close, providing about 3% of the necessary bandwidth.

As for latency, TCDL once again makes the grade for total quality. With a 2 ms latency, there is 98 ms left over for the seal/unseal, aggregation, and voice production. Even when there are several concurrent speakers in each subnet and there is a lot of overhead because aggregators have to unseal and reseal the data, the total latency should be well within the bounds set. 20 ms for production, 2 ms for latency, and a few microseconds for each seal/unseal call is still less than the total quality standard 100 ms, unless there are hundreds of speakers. When there are fewer speakers, the latency only goes down.

With Inmarsat and Connexion, the data links are the latency bottleneck, costing 325 ms. Because this latency isn't affected by number of speakers (unlike seal/unseal), total quality is impossible. However, it is well within the bounds for tolerable quality (400 ms). Iridium is out of the ballpark with a 2 second delay. It could potentially be used with radio-like speech protocols, however.

Thus, our measurements in this chapter have allowed us to asses the quality on a local subnet as well as in the two-tiered low performance link model. We recommend that satellite links such as Inmarsat and Connexion only be used when total quality is not expected and Iridium not be used for typical conversations. However, the TCDL wireless link is usable for any conversations requiring total quality.

5.4 Usability

Here we examine the usability of the system, comparing it to the design goals laid out. Goals included: use of GUIs, intuitive and easy controls for joining conferences, clarity of error/status information, and installation ease. The implementation has achieved all of these, with the exception that each system component needs to be started individually. Changing the startup process to be more cohesive is a subject of future work.

In order to load the system, the user must start X-Lite, Pidgin, and YATE separately. Also, Pidgin must have the YATE plug-in loaded. This is four steps, which could be reduced to one with a script that executes all three programs and modifies the Pidgin preference file to load the proper plug-in. On the plus side, running the programs requires a click of a desktop icon in Microsoft Windows, or a single command from the Linux terminal.

Because the user joins and leaves conferences from the membership management module, the user interacts with this component the most. Its usability is thus the most important. By using Pidgin, we piggyback on the sleek GTK-based interface they have setup. Once the user has loaded the entire system, joining conferences is a simple as a single click on the conference room of choice. Figure 5.9 shows the Pidgin window in which users can view available conferences. The information is clearly displayed, and the join action is easy and intuitive. Adding and removing conferences requires two clicks in the file menu. The graphic interface is also useful for loading the YATE plug-in from Pidgin, and includes simply selecting the appropriate option from the plug-in selection window (see Figure 5.10).

Figure 5.9 Pidgin Window for Conference Joins

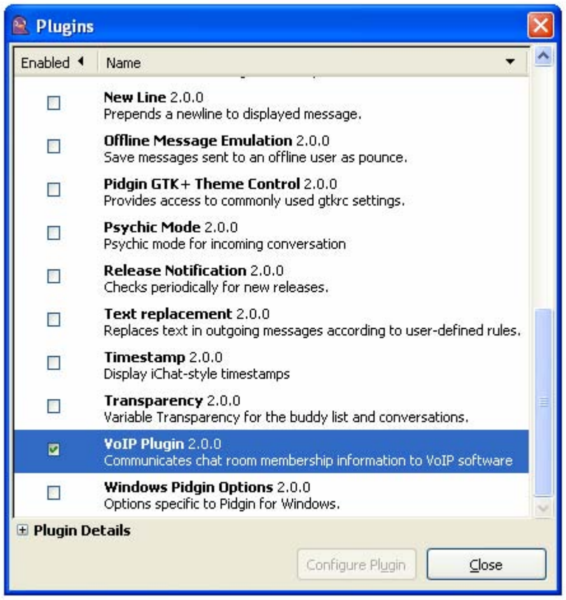

Figure 5.10 Pidgin Plug-in Load Window

Once a user joins a conference, Pidgin opens a new window that displays the conference membership. The display changes in real time as membership changes, ensuring that users are kept abreast of this important information. This is shown in Figure 5.11 .

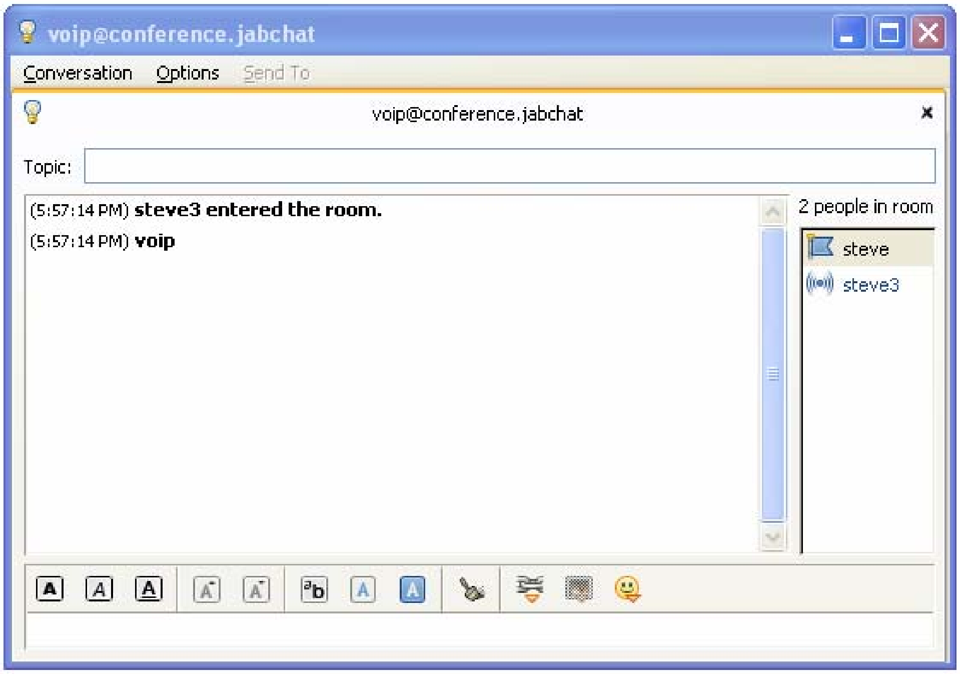

Figure 5.11 Pidgin Chat Window Displays Real-Time Membership Changes

The Pidgin GUI also makes it very easy to display errors. For example the user is always notified with a friendly and informative message box of the system's status. It should be clear, as shown in Figure 5.12, whether or not the user has configured the system properly. The message also includes instructions for how to remedy the situation when there is an error.

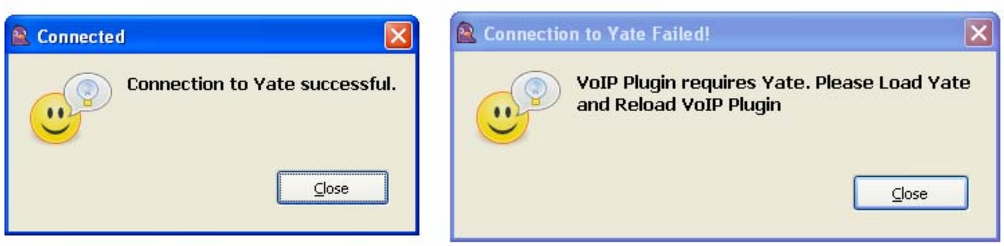

Figure 5.12 Pidgin Error Display Message

Figure 5.13 X-Lite Phone Interface

The other part of the system that the user interacts with is X-Lite, which the user must open and use to dial in to the YATE system. This requires the user to appropriately configure her/his instance of X-Lite from the options menu, specifying the IP address of the YATE system and her/his username. Once this is done, the process does not need to be repeated for subsequent use. Dialing into YATE requires two clicks. The user interface is quite intuitive, as it resembles a typical handheld telephone (see Figure 5.13). It also clearly displays errors in the window.

While YATE is needed for the system, the user does not interact with it, save to turn the system on or off.

Configuration and Installation of each of the components is complicated, and detailed in Appendix C. Linux installation involves obtaining the software for each component, compiling it, and installing it. Windows installation is similar, although X-Lite has an installer on this platform. Configuration is also complicated, but the user can either copy the config files from the appendix verbatim, or obtain a preconfigured version. In order to reduce this configuration and installation cost, a future version could include a script or installer that configures each component properly and then installs them. Similarly, the improved version could add a startup script that launches all three components at once, so that the user is ready to join conferences after a single click.

On this basis, the system fulfills usability requirements. It consistently uses graphic displays, makes joining and leaving conferences intuitive, and keeps the user updated of important information. Startup and installation are more complicated than necessary, and are thus the subjects of future development.

5.5 Evaluation Wrap-Up

This chapter has compared our first pass at implementation against the design goals set in chapter 3. Requirements for end-to-end security have been met, as have call quality standards on well-connected subnets and certain low performance links. Results show that the overhead added by the fastest PKGE protocol is small compared to the total quality standard, making the network latency the most important determinant in packet delay. PKGE does not seem to affect packet loss, provided that the extra bandwidth required does not exceed that bandwidth on the network. Major usability goals are complete, except for ease of installation and startup. Similarly, functionality goals were completed, except for implementation of the aggregator nodes in the two-tier design. These two features are left for future work.

6 Future Work

In this section we reflect on the lessons learned from this project and suggest a next generation system. The next system will revolve around GROK, which addresses some of the weaknesses in its predecessor PKGE. Moreover, it should focus on improving usability and cohesiveness of the system, so that users are not aware of the many components.

Also, based on the results of concurrent speech and membership changes, future designers should make optimizations in the system based on the expected nature of the calls. Optimization would be based on membership size, frequency of concurrent speech, and frequency of 'joins' and 'leaves'. As a final step, the next pass at secure VoIP should actually implement and test the two-tiered design to verify the success that our analysis suggests.

6.1 Replacing PKGE with GROK

GROK is the successor to PKGE, currently under development at Lincoln Laboratory. It features a number of improvements, including the ability to run multiple instances that use a common key database and the ability for keys to persist in a database after the application terminates. It even enables completely different key encapsulation protocols.

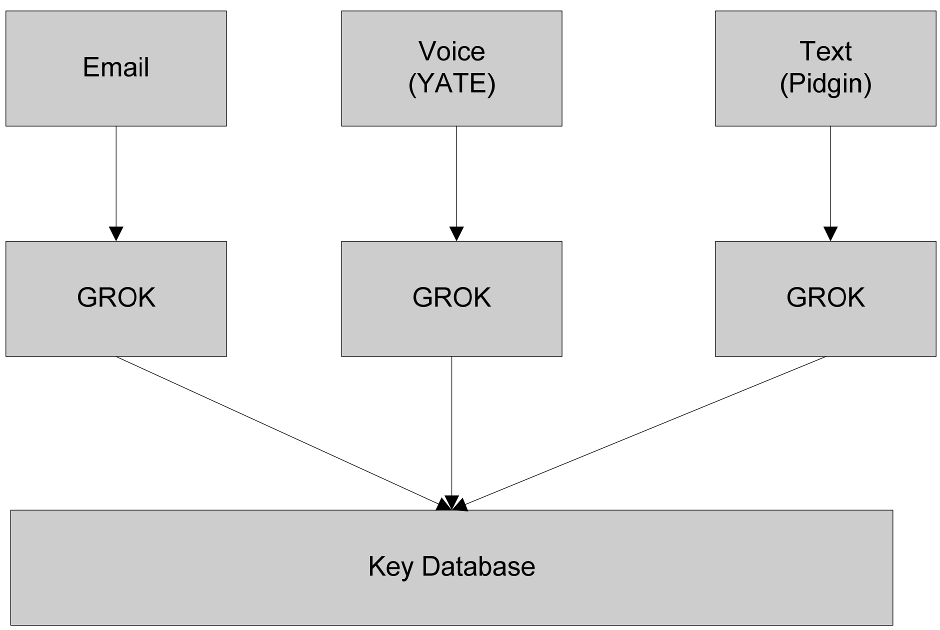

Figure 6.1 Running Multiple Instances of GROK

Allowing multiple applications to run GROK concurrently enables improvements to the VoIP system. The first such change is integration with secure text chat plug-in for Pidgin developed in [40]. Both Pidgin and YATE could use GROK simultaneously, using the same encryption keys. Thus, group conversations could take place that consist of both voice and text. Moreover, with Pidgin using GROK, the system could force key exchanges to take place over Pidgin's TCP connections rather than YATE's UDP connections. The advantage of this is that key delivery would be ensured, even on the unreliable networks that we intend to deploy on. This is important, particularly for the Lean and Optimistic protocols which only distribute keys once, making packet loss disastrous for the system.

Another logical step forward is the addition of a video conferencing program and/or secure email program to the application suite. Because multiple applications can share keys, these applications would not prevent the use of voice and text chat while it is running.

Moreover, because keys persist even after the applications terminate, keys could be re-used by future conversations. The results show that a user joining a conference is particularly costly because of the key distribution that accompanies it. With key re-use, this cost would be eliminated. With the Lean protocol, this is the only significant overhead added by security. Therefore, when using the Lean protocol for securing VoIP among a group that has existed in the past, GROK would add virtually no overhead. However, GROK will periodically change keys to achieve forward secrecy.

Even though the benefits of using broadcast encryption for key distribution are clear (described in 2.1.2), GROK will enable use of different methods. Examples include Dynamic Set Key Encryption (DSKE) [41] and S/MIME. The value of this extensibility is the ability of the designer to choose the protocol based on the exact needs of the system. Moreover, it will allow future protocols to be seamlessly added to the system as well.

Thus, use of GROK brings a host of improvements to the VoIP system we have outlined in this thesis. The next generation should replace PKGE with the newer library.

6.2 Functionality and Usability Improvements

In addition to the changes brought by GROK, the VoIP system we developed is in need of functionality and usability improvement. Usability is hindered by the use of multiple different applications that makeup the VoIP system, and the need to be independently installed, configured, and run. Within the applications themselves, users do not always have the maximum level of interaction with the system and some errors are not handled gracefully. Users could be given more power to use the functionality of the system.

The prototype developed requires users to launch three applications separately: X-Lite, Pidgin, and YATE. Then users must manually load the YATE plug-in for Pidgin and manually connect X-Lite and YATE. This functionality can be performed automatically with a single script. It was left out of the prototype due to time constraints. It is a requirement for future versions that they should allow users to start the system and be ready to join conferences with a single action.

Furthermore, while the Pidgin interface is aesthetically pleasing and convenient to use, we could give the users more information. For example, users do not know which other users have added the YATE plug-in and are ready for voice chat. A simple icon next to a user name could give this information. It could be disseminated by the system in the same way that IP info is disseminated, by overloading the text message functions. On this matter there is another needed improvement. Users who do not have the plug-in loaded will get useless IP info printed to their conversation window. Instead, IP information could be hidden in a *span* tag, in which the actual displayed message is only "VoIP Info." This solution is more graceful for users not running the plug-in.

It is also logical to authenticate IP addresses that are distributed to prevent hackers from diverting the flow of voice data by passing users erroneous information. The next generation system should include authentication of this certificate-based authentication, as is being developed at Lincoln Lab.

In the previous section we noted that key distribution with Pidgin would improve reliability. As an additional improvement, users might have an option from within Pidgin to re-key the group. This way if keys are lost in transit or due to failures of the end user's computer, a simple re-key would fix the problem. Without GROK, this would require Pidgin to tell YATE to re-send the key encapsulation. YATE would have to store the key encapsulation indefinitely, and resend it when needed. With GROK in place, Pidgin could send the key encapsulation itself.

These changes would give the user more information and improve system robustness. It would make the system easier to run and use. They are suggestions for a better second implementation of secure VoIP.

6.3 Addressing Usage Variables

Perhaps the most important lessons learned for the next generation system involve the various conference usage variables that affect the system performance. The results from chapter 5 show that 'join' messages cause slowdowns of roughly 30 ms on each end, which is longer than the duration of the voice included in a single packet. Similarly, concurrent speech doubles the required bandwidth, and doubles processing latency. Because these events are assumed to be rare, our system does not take measures to handle them. However, a future system might be adaptable to handle such situations for completeness and robustness.

If receiving a join message causes a 30 ms slowdown due to message unsealing, then the processing of the next message is delayed. Ideally, for 20 ms voice messages, messages should be processed every 20 ms. Longer than 20 ms delays cause the system's audio to slowly "get behind", assuming that the packets are processed in serial. That is, the delay beyond 20 ms is compounded with each join message, so after 10 such messages, the system will be 100 ms behind. There will be 100 ms of delay in addition to the transmission delay and regular processing latency. As these incremental delays add up, the total end-to-end delay rapidly exceeds the upper limits for quality.

The system compensates for this during silent periods, during which it can "catch up". That is, if the system is behind by 20 ms, it needs only 20 ms of silence during which no packets are received to recover. Our system assumes that this is the case. This is reasonable, as [21] reports that typical conversations include 200 ms "turn-taking" silence, which occurs in between speakers. Consequently, roughly 20 join messages can be received for each speaker turn. Thus, if the speaker changes frequently (with 200 ms of silence) the system can accommodate a high number of joins.

On the other hand, a conference with long incessant speech and a high number of listeners coming and going will likely experience increasing delays. Consider a scenario (irrelevant to the DoD) of a web concert with incessant music from a single source and thousands of listeners coming and going. The system developed would not fare well here.

A potential solution would be to systematically drop packets as delays accumulate. For every 20 ms of delay accrued, drop one 20 ms packet. This of course will quickly decrease call quality due to packet loss. However, one packet dropped every two seconds would remain within the tolerable quality bounds. Another solution would be to use larger packets, 40 ms for example, such that a join message does not cause accumulated delay.

The next system could also allow for greater packet loss by using a Packet Loss Concealment (PLC) algorithm as described in Appendix I of [25]. Such algorithms can improve sound quality in the face of packet loss by employing a number of techniques, including repetition of the received packets. In fact, the G.711 PLC algorithm can tolerate packet loss of 5% and still have acceptable quality [21]. In this case, 50 ms of audio can be dropped every second, allowing for about five 'join' messages per second. Thus, future system designers should consider the usage cases and determine whether PLC algorithms, changes in packet size, or systematic packet loss are appropriate.

Concurrent speech also increases processing latency. However, with the Lean protocol requiring virtually no processing time, the main drawback from multiple speakers is the increase in required bandwidth. If the usage case of the system involves frequent concurrent speech from two or more parties and requires this speech to be transmitted without packet loss, the network bandwidth must accommodate this. If the network only has enough bandwidth for one speaker, and two users speak at the same time, packet loss with exceed 5%. Thus, the solutions for accumulated delay are not helpful here.

Instead, the system designer might consider a different codec, which is something we did not consider in this thesis. More audio compression invariable results in lower packet loss toleration and more audio processing. However, if the codec sufficiently reduces bandwidth requirements, then packet loss will not be an issue. Similarly, with adequate transmission latency and use of the Lean protocol, extra processing might not be an issue either. Use of a codec such as G.729A [26] requires only 20 bytes for 20 ms, compared with 160 bytes for G.711. Excluding the bandwidth required for IP, RTP, and UDP headers, this is a factor of eight decrease. This could alleviate bandwidth issues for a system. Thus, a designer should consider different compression algorithms depending on the expected bandwidth of the network and the expected bandwidth required by the specific usage.

Another potential, and perhaps less desirable, solution is to allow quality degradation when there are multiple speakers. This solution assumes that the loss of intelligible voice will signal users that concurrent speech has occurred and one user must stop talking. In general, the individual voice streams during concurrent speech are unintelligible anyway, even if transmitted faithfully. This is particularly true if there are more than two speakers.

Thus, there are a number of scenarios that the system we describe does not cover. Depending on the needs of a next generation system, these scenarios can be accounted for using a combination of solutions. Solutions to ever increasing audio delay include deliberate packet loss, packet size changes, and PLC algorithms. Similarly, inadequate network bandwidth (due to concurrent speech or otherwise) can be alleviated with compression algorithms.

6.4 Implementing the Two-Tiered Model

In this thesis, we designed an elegant solution to minimize the impact of a weak link in a disadvantaged network. Our test results in chapter 5 show that the solution is feasible and could be used with several high latency, low bandwidth satellite links. However, the system never actually implements the design. The next generation system should use the building blocks of the first pass to actually develop the two-tiered model.

This development would require expanding the peer-to-peer model by implementing the aggregator node functionality specified. An aggregator node forwards audio data from the speaker on its network across the weak link. When there are multiple speakers, the aggregator combines them into a single stream, saving bandwidth. Moreover, the implementation would require an algorithm for dynamically selecting aggregator nodes as the conference membership changes. Because the current model already has functionality for combining voice streams, these changes are reasonable.

6.5 Summary

In this section we reviewed possible developments in a second pass system for secure VoIP. Logically, the system would be based around PKGE's successor GROK, enabling multiple simultaneous secure messaging applications. It would also enable new key encapsulation methods and key distribution methods. Moreover, the next system should work at improving some of the usability flaws of the one we describe. Such flaws include having multiple separate components that each require installation and startup, and limiting the amount of information and control the user has over the system. The new system should also build off the lessons learned here, and be designed with more specific usage in mind. This way the designer could consider issues such as the audio codec used, the packet size, and how to handle ever increasing audio lag. Of course, the next system should also fully implement the two-tiered design for disadvantaged networks.

7 Conclusion

At the conclusion of this thesis, we reflect on what we set out to do. Our purpose was to develop a proof-of-concept VoIP conference system that showcased the use of PKGE for secure group communication in a real time setting. This proof-of-concept would advance the DoD vision for improved communication, even in disadvantaged networks. Most importantly, the project was to contribute insight for further VoIP system development and to guide design decision for a second generation system. In short, we accomplished each of these goals.

We have a working VoIP conference system, composed of several different technologies integrated to accomplish our design goals of security, performance, functionality, and usability. Moreover, our analysis of the system suggests that it would perform well in disadvantaged networks, if our design for such situation were fully implemented.

Looking back at the engineering challenges encountered and solutions implemented, we came up with a few guidelines for the next generation. PKGE should be replaced with its successor GROK, which will allow multiple concurrent instances with a cohesive key database, thereby allowing multiple secure communication technologies. This will increase robustness by allowing the most reliable transmission protocols to be used for key distribution. Furthermore, the system, while composed of several different technologies, should be integrated well enough that its many components are concealed from the user. The next system should also consider specific usage cases in order to make appropriate trade-offs in design, between packet loss and packet delay or functionality and performance.

Because we designed the system with extensibility in mind, it should be possible to make these additions, and develop appropriate new features as well. A next generation system might add end-to-end confidentiality or authentication to membership management and location information distribution. It might also redesign the entire system for different goals entirely.

Whatever the case, this thesis should provide a solid starting point for upcoming developments. It highlights the challenges and needs of a VoIP system, and evaluates a set of solutions.

Appendix A Relevant YATE Details

This section contains an overview of the relevant components of Yet Another Telephony Engine (YATE). Included is an examination of YATE software design, analysis of the conference and RTP code modules, and configuration file information. For simplicity, only information directly relevant to secure conferencing using PKGE is included.

Note that the information here was acquired in part from [17] and [36], but mostly was assembled by viewing source code and debugging. Consequently, I make no guarantees about its veracity. It is merely the behavior observed.

YATE is a multifaceted software telephony engine capable of executing SIP call transactions, sending and receiving RTP streams, routing or redirecting calls to end users or servers, interfacing with the Public Switch Telephone Network, and acting as an H.323 gatekeeper, among other things. However, because the focus of this project is VoIP conferencing, we are primarily concerned with its functionality to host conference calls and mix multiple voice streams. Moreover, because our goal is to add new functionality to the system, extensibility of the software is vital. It is here that we begin our examination of YATE.

A.1 Software Design

Much of the YATE functionality, such as its ability to understand a given protocol, is encapsulated in a series of modules that communicate amongst themselves using a well-defined message passing system. The telephone engine provides the foundation for these modules with code to receive, queue, and dispatch messages. It also contains the classes and interfaces that define data flows for call transactions and media streams. These are ultimately implemented and subclassed for module-specific functionality. Of course, the telephone engine core also contains code that initializes each module and launches it in a separate thread, readying it to receive messages and carry out its duties. YATE is strongly object oriented; it is written in C++. The telephone engine provides support for Perl, PHP, and Python scripts to run in separate processes and communicate messages with the engine via TCP sockets or Unix pipes.

A.1.1 Message Passing System

The message passing system of YATE is its communication backbone. There are four classes to focus on here: Message, MessageRelay, MessageDispatcher, and MessageReceiver.

A Message instance encapsulates data that is passed between modules. The message has a name, which indicates the desired course of action, and a set of parameters that provide any information needed for that course of action. The parameter names and values are String objects. For example, if YATE receives an incoming call to a conference room, it would send a `call.execute` message to the conference module, with parameters containing the conference room number, the caller, whether to let the user hear her/his own voice, etc.

A MessageReceiver object contains the functionality for handling any number of messages. The YATE modules are MessageReceiver subclasses, and thus contain the appropriate functions that are invoked when messages are passed to them. Specifically, a MessageReceiver's `received` method is called when it is passed a message. Carrying on our example from above, the conference module is a MessageReceiver which handles several messages including `call.execute`. Its `received` method dispatches to other functions based on the type of message received. For `call.execute`, `received` dispatches to a function which creates a new conference leg an attaches the incoming party to it.

MessageReceiver objects are wrapped in MessageRelays that specify which messages they are equipped to handle. These MessageRelays are *installed* in the telephony engine. When the engine receives a new message, it jumps through a list of all installed relays in order to find ones that can handle the given message. The `received` method of each applicable MessageReceiver is invoked in order of priority for the message. Ultimately, one of the `received` methods will decide that the message was intended for it. Then, it will execute the appropriate functionality and tell the engine it can stop calling `received` methods by returning true. In the example, the conference module wraps installs a MessageRelay in the telephony engine for each message it is equipped to handle, including `call.execute`. When the engine receives the message, it may try a number of MessageReceivers that can handle it, before the conference module's receiver is invoked. The conference module will handle the message and return true so the engine will stop.

The engine itself receives the message whenever a module calls its `dispatch` method, passing a Message object as a parameter. The engine then refers to its MessageDispatcher, which holds all of the installed MessageRelays. This typically occurs in a new thread.

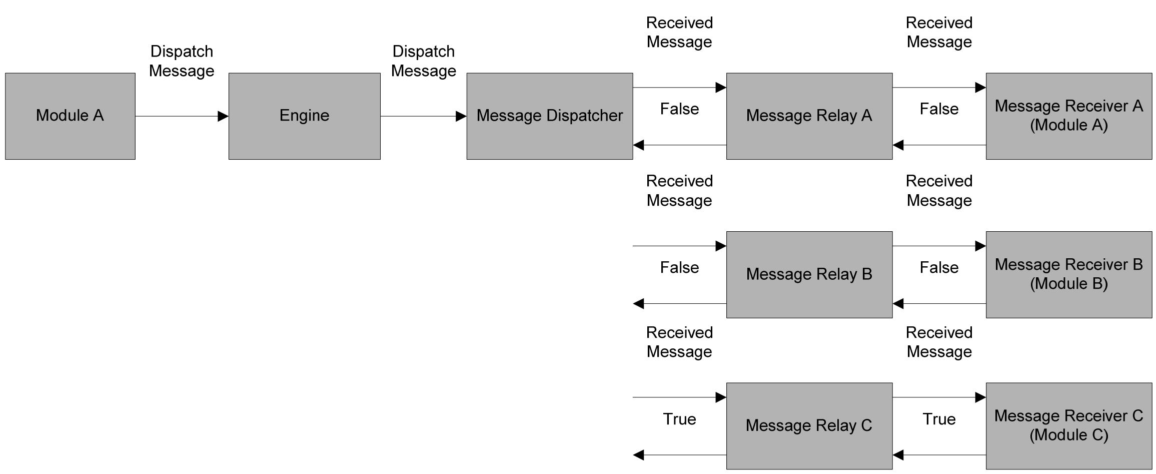

Figure A.1 Message Flow Diagram

A.1.2 Data Flows

While the message passing system allows different modules to send instructions to each other, modules frequently pass entire DataBlocks to one another in order to take advantage of the array of functionality offered by YATE. DataBlocks are simply encapsulations of chunks of bytes. They are used primarily to hold audio data being passed between modules for processing. For example, after audio data enters the YATE system, it is processed using the G.711 codec. This process is carried out by a DataTranslator instance that provides this function. The DataBlock is passed in to the DataTranslator and must subsequently be passed to another module for continued processing before being passed to the RTP module for transmission.

More specifically, data flow take place between DataConsumer objects and DataSource objects. A DataSource instance produces DataBlocks that are *consumed* by DataConsumer objects. The two classes are linked via composition; each contains an instance of the other as a member variable. Thus through the course of data processing by YATE, a DataBlock may be passed between several sources and consumers, each modifying it in its own way. Modules use these classes by subclassing them, and adding their own module-specific functionality. An RTPConsumer passes data to a socket, a ConfConsumer mixes multiple data streams and reroutes them to different endpoints, and a DataTranslator compresses or decompresses data.

A.1.3 Modules

The modules themselves provide the actual functions for YATE. They go to work when passed Messages and DataBlocks. Some important modules include the SIP stack, RTP stack, conference bridge, and several routing modules. By subclassing the Module class, it is easy for third party developers to add their own functionality to the YATE system. Modules must simply register a few MessageRelays to have their functionality invoked. Moreover, by understanding the ins and outs of certain key modules, developers can make modifications that suit their needs. Thus, I will take a closer look at two important modules for the secure conference system.

A.2 Relevant Modules

Instead of implementing new modules, I elected to change existing modules to enable the desired behavior. I do this because the existing modules' behavior is very close to the behavior of the the secure conference design, and a few tweaks make this project's implementation simple. Thus, we require an intricate examination of modules we intend to modify.

A.2.1 The RTP channel

The RTP channel starts with an RTPProcessor object which encapsulates the socket and address data for an RTP session. An RTPSession object subclasses the RTPProcessor, and contains all the additional data and functionality for an actual RTP session. This extra data includes an RTPSender, which packages and sends RTP packets using the RTPProcessor socket data, and an RTPReceiver, which receives RTP data on the same sockets. It also contains an RTPTransport object which uses the socket and address data of the RTPProcessor to invoke the actual socket functions. Naturally, RTPSender and RTPReceiver rely on these methods.

If this sounds confusing, the point is actually to achieve clarity and extensibility through functional separation. For example, the sole function of RTPSender is to create and RTP packet, which involves adding the appropriate headers. For it to send the packet, it invokes RTPTransport's `rtpData` function. RTPTransport handles all socket interaction. In this way, there should be no confusion about each class's role. Also, it is possible to use the classes independently of one another. If the entire packaging and sending of RTP packets were rolled into one method, then it would be impossible to dispatch packets without first adding RTP headers. If this seems useless, consider a scenario that required resending prebuilt packets. Combining packing and sending would force unnecessary computation.

In order to integrate this functionality back in with the message passing and data flow models described above, the RTPSession class is extended by YRTPSession. This, along with RTP's very own DataConsumer and DataSource YRTPConsumer and YRTPSource are wrapped in a YRTPWrapper. With this setup, data enters YATE through the RTPTranport methods that YRTPSession accesses by invoking methods in its RTPReceiver. Then the data is moved to the YRTPSource, which can feed the data into the rest of the system: compression, conference bridging, etc. On the other side of things, data leaves YATE by passing into the YRTPConsumer from some DataSource, and then leaving via a UDP socket in the YRTPSession's RTPTransport instance. This is invoked with the RTPSender.

A.2.2 The Conference Module

With the RTP module taking care of all things network related, let's move over to the conference module, where we find the conference bridge functionality. This of course consists of mixing audio signal from several sources and passing it back to each of the conference participants. In this module, we've got the ConfRoom, ConfChan, ConfSource, ConfConsumer, and ConfDriver.

ConfRoom encapsulates a list of ConfChans, which do little more than hold ConfConsumers. ConfConsumers are the DataConsumers for conferences, in that they bring data into the module. When a ConfConsumer has accumulated enough audio data, it triggers the mixing method of the ConfRoom it is a part of. This method combines all the data from its list of channels and their consumers, and forwards it out to the ConfSource which is the data bridge out of the module.

ConfDriver handles the message passing part of the module, having installed relays into the telephony engine for handling a number of messages, most notably call.execute messages marked for the conference module. When it receives a call.execute that signals an incoming call to a conference room, it creates a new ConfChannel and adds it to appropriate ConfRoom. If the ConfRoom does not exist, it creates it. The object model diagram in A.3 should clarify the various dependencies here.

A.3 Configuration Files

YATE has a pretty snazzy configuration file interface, making it easy to add additional configuration files for new modules. The Configuration class encapsulates a system configuration file. It provides a host of functions, such as int getIntValue(const String& sect, const String& key, int defValue), which allow modules to easily extract configuration parameters from the files. The Configuration class constructor accepts a String parameter that defines a file of type .conf saved in the conf.d subdirectory of yate. Code Chunk A.1 shows how Conference.conf (see below) is accessed by the conference module using the Configuration class.

```
Configuration s_cfg;
String s_pin;
String s_system;
…
s_cfg = Engine::configFile("conference");
s_cfg.load();
s_pin = s_cfg.getValue("user","pin",PKGE_USER_PIN);
s_system = s_cfg.getValue("general","system",PKGE_SYSTEM);
```

Code Chunk A.1 Accessing Configuration Files

Because deployment of the secure conference system requires specific configuration of a few modules, the important config files are described below. The examples given include the parameter values needed for the secure conference system. System users and developers should use these files.

Conference.conf - This configuration file was added as part of this project. It specifies PKGE system to use for the encrypted conference, and what the pin of the user is. Here is what was used:

```
[general]
system="C:\pkge\TESTSYSTEM\"
pin=12345
```

Ysipchan.conf - Here we define the SIP channel parameters, such as the port to use and the IP address to bind to. The file used was:

```
[general]
port=5066
addr=localhost
```

Regexroute.conf - This file handles call routing. That is, it uses regular expression parsing of called numbers to dispatch the call to the appropriate module. Each module has a routing prefix. For example, a prefix of "conf/" routes a call to the conference module, while the prefix "tone/" routes the call to a tone (dial, busy, etc) generating module. The regex parsing can be specified differently for each caller, by specifying in the contexts section which routing section in the file to use. An example is included here:

```
[default]
^0$=tone/dial
^1$=conf/1; lonely=true
^2$=sip/Roger@192.5.135.137:5066
^\(.*\)$=\1
```

This configuration file allows users to call into a dial tone (0), join a conference room (1), or make a SIP call to a specific user (2). The final line lets users compose their own routing string. For example, a user could call "sip/user@host:port" to have a SIP call made.

Regfile.conf - This config file specifies users and their passwords. YATE will only route calls from users included in this file. Example:

```
;usernames, passwords included in the form:
;[user]
;password=
[Steve]
password=Steve
[Roger]
password=Roger
[Joe]
password=Joe
```

Extmod.conf – This config file specifies what external scripts the telephony engine should launch, and how it should connect to them. The scripts to launch are listed in the "scripts" section. Scripts are connected using the listeners specified in square brackets, which listen for a certain connection type on a given address and port. Example:

```
[general]
[listener A]
type=tcp
addr=0.0.0.0
port=5039
role=global
[scripts]
C:\yate3\yate\scripts\ConfNet.py=
```

Appendix B Relevant VoIP Details

Voice over Internet Protocol (VoIP) is a recent technological innovation in telecommunications, crossing the infrastructure of the internet with the function of the public switch telephone network (PSTN). VoIP improves efficiency and flexibility of voice communication.

In this section, I review the technology behind VoIP, including system components and protocols involved. VoIP is compared with other communication technologies on the basis of its security, reliability, and quality. I recommend that readers unfamiliar with VoIP technology start here before diving into chapters 3 and 4, which contain more technical detail.

B.1 VoIP vs PSTN

The PSTN is a *circuit-switched* network. The entire duration of a phone call requires a dedicated circuit between the two endpoints for voice signal transmission. The circuit is set up using a complex system that has evolved for over one hundred years through research by AT&T and Bell Labs. Originally, this meant an operator manually connected the pieces of a circuit at a *switch.* Today, circuit-switching has come to involve a hierarchical, automated system that uses phone numbers standardized by the International Telecommunication Unit (ITU).

The dedicated circuits of a phone call require 64 kilobits per second of bandwidth for both directions of communication, thus requiring a total of 128 kbps. It is seldom the case that all of this bandwidth is actually used, based on normal human conversation. Generally, a only single user speaks at a given time and there are frequent brief pauses in speech. Consequently, a significant portion of the bandwidth is wasted.

In order to reclaim this wasted circuit time, *packet-switching* can be used for voice transmission. In this model, voice is broken into chunks or packets and transmitted over the wire only when it is actually produced, leaving unused bandwidth open for other data transmission. When no voice is produced (silence), bandwidth is not wasted. Thus, bandwidth is shared by users in proportion to the amount needed. VoIP employs this model, using Internet Protocol (IP) for packetizing and routing voice calls. In this manner, VoIP gets its first advantage over the PSTN.

B.2 System Components

A VoIP system includes audio capture, analog-to-digital conversion, data compression, call setup, voice data transmission, decompression, conversion to analog, and audio playback. Most of this is beyond the scope of this thesis, as I focus on adding security to conference calls. However, let’s take a brief look at how these systems are designed.

An end user typically executes a VoIP call using some combination of hardware and software. On one end of the spectrum, Cisco IP Phones are standalone hardware devices that mimic the typical PSTN experience. They handle all aspects of the call, from audio capture/playback and A/D conversion to using the VoIP protocols for setup and transmission. Also common is the use of a *softphone*, which is a software component that performs the data processing and transmission, and leaves the audio capture/playback and digital conversion to microphones and speakers. Skype [11] is an example of a softphone.

In between the two endpoints, the VoIP infrastructure can be anything from purely peer-to-peer to client/server architectures. Calls can use the open Internet, or can use closed off networks. These differences depend on the protocols used and the goals of the system designers.

B.3 Important Protocols

Session Initiation Protocol (SIP) and H.323 are the two main protocols for call setup and teardown. H.323 is rooted in the PSTN community and is generally considered more complex than SIP; consequently, SIP is rapidly becoming the de facto VoIP standard. Thus, I focus on SIP in this thesis.

SIP in an application layer protocol that can use any transport protocol, such as User Datagram Protocol (UDP) or Transmission Control Protocol (TCP). It is used to create media sessions between two or more parties, the exact nature of which is defined during the transaction. For VoIP, SIP creates and terminates real-time audio sessions.

Because some transport layer protocols, such as UDP, guarantee only best-effort packet delivery, SIP transactions use their own acknowledgements to ensure proper call setup. A SIP transaction typically consists of a request from one party to another, followed by a series of responses that indicate the success or failure of the request.

A call setup, for example, would consist of one party sending another party an INVITE to an audio session defined by specific codecs, followed by an OK response from the latter party. The caller then sends an ACK request to verify receipt of the OK. The session itself then takes place using Real-time Transport Protocol (RTP), and ends when one user sends a BYE request. A summary of the main SIP Requests and Responses detailed in [30] is included in Table B.1.

Request	Meaning
INVITE	Invites a user to a media session
ACK	Confirms that inviting client has received all responses for a session
BYE	Ends a session
CANCEL	Cancels pending invites
OPTIONS	Request for capabilities of a SIP server
REGISTER	Registers a client's address with a server

Response Family	Meaning
1xx	Information: ringing, queued, etc
2xx	Success
3xx	Redirect Responses
4xx	Client Failure: busy, forbidden, address incomplete, etc
5xx	Server failure: service unavailable, time-out, request not implemented
6xx	Global Failure: decline, does not exist, etc

Table B.1 SIP Requests and Responses

SIP requests and responses need not be passed directly between endpoints, however. In order to enable routing, SIP proxy and redirect servers intermediate SIP transactions. A proxy server receives requests from one party and forwards them on to the intended recipient, and continues to intermediate the rest of the transaction. Redirect servers merely resolve recipient user names to IP addresses for the calling party. Thus, users can employ either redirect or proxy servers to invite others to a session without necessarily knowing their exact locations. The RTP session itself takes place directly between the two parties, who are privy to each others addresses after the SIP transaction.

RTP, another application layer protocol, wraps the media data for the session. Because real time media does not usually require 100% packet delivery but does require speed optimizations, RTP uses UDP exclusively. RTP [31] contains just 12 bytes of headers for sequence numbers, timestamps, and payload identification before the payload itself. This enables jitter (packet delay variation) adjustments before playback, while being relatively lightweight.

Based on this understanding of SIP and RTP, the picture of a VoIP call from setup to teardown looks like Figure B.1.   In this example, a SIP server is used in proxy mode.

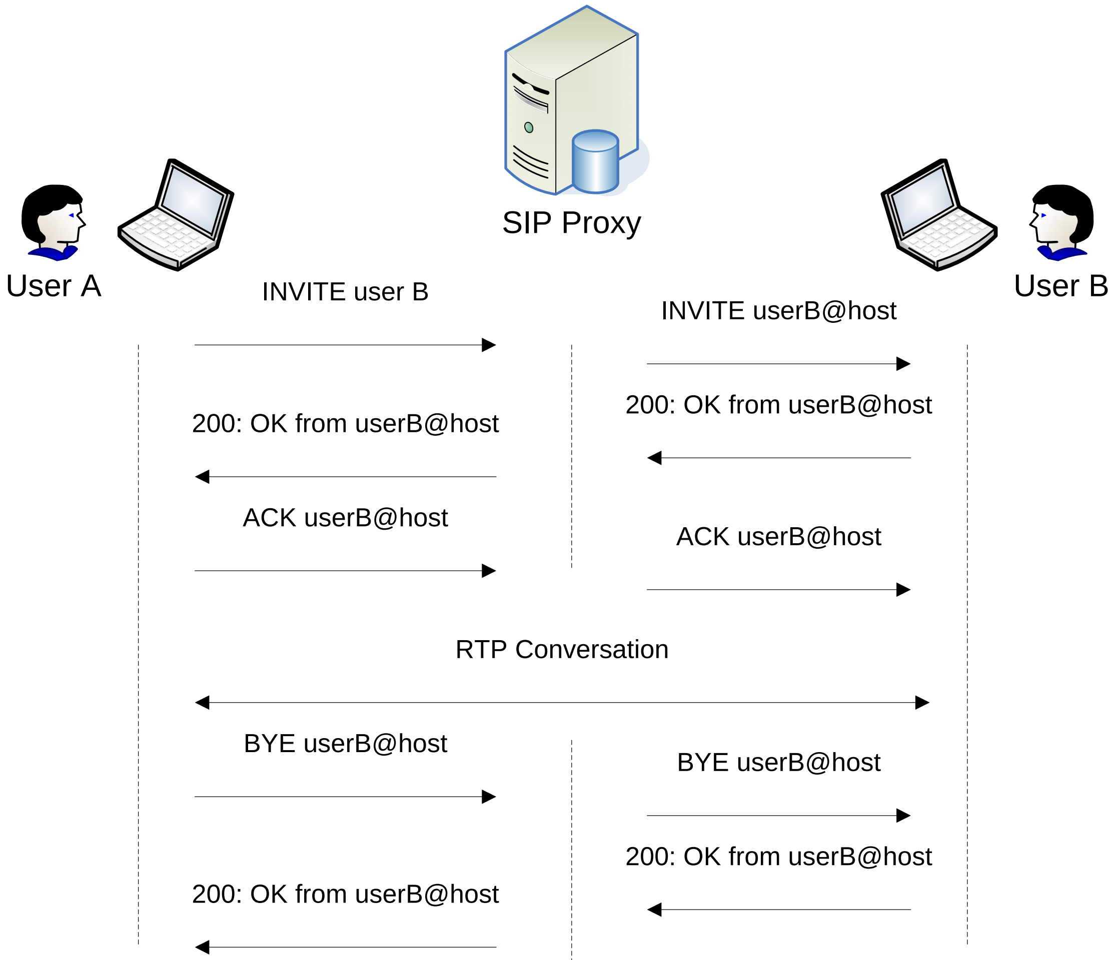

Figure B.1 SIP Setup and Teardown

B.4 QoS and Security

Quality of service in the PSTN is guaranteed by a dedicated circuit of a fixed bandwidth. The necessary bandwidth for human voice encoding is considered to be 4000 Hz; thus, telephony must use 8000 samples per second, according to Nyquist's Law. Because the ITU standard for PSTN communications is the 8-bit G.711 codec, the total bandwidth required is 64 kbps for each voice steam. This is a total of 128 kbps for a two person call. A VoIP system can mimic this quality by using the same codec.

However, differences in quality arise due to the differences in resource allocation. Due to the dedicated nature of circuit switched networks, once a circuit is allocated for a call, the call will continue uninterrupted with persistent quality. The ITU standard is 99.999% circuit availability time. To the contrast, use of the internet to transmit voice leaves the calls vulnerable to network congestion related quality degradations. Because RTP is built on top of UDP, voice packets will be dropped when network resources are overused. Moreover, voice packets may be delayed if the link is slow or overused. Such decreases in quality can occur at anytime as network congestion comes and goes. Thus, while VoIP calls require the same 128 kbps of bandwidth for 100% packet delivery, the actual data link requirements may be much higher if other applications share the resources. A work-around gives priority to real-time streaming data on a shared link. However, this conflicts with the principles of net neutrality.

Also, voice packets contain headers for each protocol layer used, thereby adding slightly higher bandwidth requirements. RTP has 12 bytes of headers, UDP has 8 bytes, and IP adds 20 bytes. This adds 40 bytes of overhead. If 160 byte payload are used (20 ms with G.711), then an additional 16 kbps needed.

Because of the shared nature of packet switched resources, VoIP is open to security vulnerabilities. The PSTN is a closed system that gives sole access to the phone companies. VoIP data is available to anyone at an intermediary hop between two endpoints, as well as anyone sniffing a link that the packet traverses. For example, two machines on the same Ethernet can see all the same packets.

There are many ways to achieve security in such scenarios. Everything from link layer security to separate networks to application layer security can be employed, with different advantages and disadvantages. This thesis examines one such method.

Appendix C Installation Guide

This section contains the information necessary for users and developers to install and run the VoIP system described in section 4.

C.1 Linux Installation

1. Obtain the PKGE source, which is available in subversion from within Lincoln Laboratory, at http://subversion/svn/sgc/trunk/pkge. Configure, make and install pkge. Make sure it installs to /usr/local.

2. Export the directory containing the pkge library files as part of the library load path so other programs can link to it. Use the following command:

   export LD_LIBRARY_PATH=/usr/local/lib

3. Obtain the YATE source code files in their modified form. From within Lincoln Laboratory, these files can be found in the group subversion directory subversion/svn/sgc/trunk/yate. Make sure the configuration files are correct, as defined in Appendix A.

4. Run the configure script. Alternatively, do not run the script, and test if running make succeeds, if it does, run make install and ignore steps 5-6.

5. Adjust Makefile in yate/scripts to include ConfNet.py in the list of scripts to launch. Adjust the Makefile in the yate/modules to include another two target lines:

   Conference.yate: LOCALLIBS = -L/usr/local/lib -lpkge
   Yrtpchan.yate: LOCALLIBS = -L/usr/local/lib -lpkge

6. Run make and make install in the yate directory. Make install may need the noapi option. In this case run make install-noapi.

7. Download and install the latest version of Pidgin. Source and installation instructions (including dependencies) can be found at http://developer.pidgin.im. Make sure it installs to /usr/local/.

8. Download and Install X-Lite, which is freely available at http://www.counterpath.com/.

9. Make sure the pidgin/plugins directory contains yate.c. Run make yate.so. Copy yate.so to /usr/local/lib/pidgin.

10. The installation is complete. Each program should be runnable from the

command line. Start yate first by typing ‘yate’ or ./run from the yate directory. Pidgin is invoked with Pidgin, and X-Lite from within its directory.

C.2 Windows Installation

Windows installation requires Cygwin [3] and MinGW [7] to be properly installed in C:\Program Files. Microsoft Visual Studio is also required.

1. Obtain the PKGE source, which is available in subversion from within Lincoln Laboratory, at http://subversion/svn/sgc/trunk/pkge.

2. Add --output-def,pkge.def to the pkge.dll compile statement in the pkge Makefile. It should look something like this now:

```
pkge.dll: libpkge.a
 $(CC) -shared -o pkge.dll \
        -Wl,--output-def,pkge.def,--out-implib=libpkge.dll.a \
        -Wl,--export-all-symbols \
        -Wl,--enable-auto-import \
        -Wl,--whole-archive ${pack_libs} \
        -Wl,--no-whole-archive ${link_libs} \
        -Wl,--kill-at \
        $(LDFLAGS)
```

3. Configure, make and install pkge

4. Copy the pkge.dll file to C:\windows\system32. Copy the pkge.def file to the visual studio subdirectory VC\include. Copy the pkge header directory from mingw/include/pkge to VC\include.

5. Use the Visual Studio tool lib (probably in VC\bin) to create a .lib file that VS can use (it can't use libpkge.a). At a command line type:

```
vcvarsall.bat
lib /machine:i386 /def:pkge.def
```

   Move this pkge.lib file to VC\lib

6. Obtain the modified YATE files from svn as in step 2 of the linux instructions. The source files all end in LF characters rather than CRLF, which is needed by visual studio. WinZip should let you make this conversion, if not, use Wordpad. Open the YATE project in Visual Studio. This file should be in the windows subdirectory of YATE.

7. In VS, link to PKGE by right-clicking the project, and selecting properties->Configuration Properties->Linker->Input. Add the full path of the pkge.lib.

8. Now you can build YATE either for DEBUG or RELEASE. Certain modules can be omitted from the build, if they don't compile due to dependencies. These are: h323chan, gsmcodec, wpchan, Gtk2Client , mysqldb, and pgsqldb.

9. Obtain Pidgin, either from subversion or from the Pidgin website. If from the Pidgin website, obtain yate.c from subversion and put in the pidgin/plug-ins directory.

10. Install Pidgin according to the instructions on its website. Run make yate.dll from the pidgin/plugins directory. Move yate.dll into the pidgin/win32-install-directory/plugins.

11. Obtain and install X-LITE from http://www.counterpath.com.

12. All components are now installed. Run YATE first by double-clicking the yate-console application in the Release or Debug subdirectory of yate/windows. Then run Pidgin and X-Lite.

C.3 Running Information

There are a few pieces of information necessary to properly run the system.

1. The yate plug-in must be loaded in Pidgin.

2. The PKGE directory must have a system generated. This includes the PKI for the conference. This system is referenced in conference.conf in yate/conf.d or yate/windows/conf.d.

3. After loading X-Lite, users must "dial-in" to the conference system by dialing 1. X-Lite must also be configured properly. This includes changing the SIP proxy in the configuration menu to the local host with port 5066, and changing the username and password to one referenced in regfile.conf in the conf.d directory.

Appendix D Related Work

Before embarking on the design and development journey, we carried out a thorough examination of related projects. Such an evaluation allowed us to borrow concepts and code. Specifically, we focused on secure VoIP systems to ascertain how security is added in existing commercial products, and how such methodologies could be improved. We assess the advantages and disadvantages for each approach, noting what is relevant and useful in our project.

D.1 Skype

Skype [11] is a free but closed-source, proprietary VoIP system. Users download and run the client software, which performs all of the processing, including audio compression and decompression, call signaling, and encryption. Voice data is transmitted from peer-to-peer on the Skype network, but protocols used are engineered by Skype, and are not released to the public. According to the FAQ in [11], call encryption is done from end-to-end using 256-bit Advanced Encryption Standard (AES) keys. The keys themselves are exchanged using 1024 bit RSA encryption. Moreover, Skype allows encrypted conferences to take place as well.

On this basis, Skype has a number of desirable features for our system. End-to-end encryption is valued; however, the trust model still extends beyond the conversing parties because Skype user public keys are certified by Skype servers. Similarly, because the protocols are closed and Skype takes measures to prevent reverse-engineering, users must trust Skype not to do anything malicious. Because our system should be versatile for any number of uses, it makes the most sense for it to rely on as little outside trust as possible.

Moreover, conference calls exist only as long as the host party remains active. The dynamic system that we develop should not crash when one of the parties leaves. Consider the case of a call network among fighter pilots. Any one of the pilots might leave to refuel or separate from the group at anytime. The system would sacrifice a lot of usability if it had to be restarted every time the host leaves.

Skype has a number of properties that are difficult to assess. For example, it is not clear whether conference encryption uses a shared group key (like PKGE) or if conference encryption is point to point between the host and each user. The latter case suggests the need for a lot of overhead to encrypt data to each user separately. If a group key is used, the closed nature of the protocol makes it hard to determine if the system offers forward and backward secrecy. That is, if a user leaves the conference, it is not clear if she/he can still eavesdrop on the conversation.

Also, as noted in [42], Skype does not release any information about key exchange algorithms or any significant detail about its encryption and authentication implementation. Thus, it is hard to make any concrete conclusions, and it seems unwise to assume that everything it implemented properly and securely. As mentioned in [42], Skype might use encryption, but use it poorly.

Thus, Skype presents a promising model because it offers encrypted conference calls, however, it has several features that our system must improves upon.

D.2 Zfone

Zfone [15] is a VoIP product developed to add security to existing VoIP systems. It detects call setup between two parties, and negotiates encryption key exchange in the media stream. Call data can then be encrypted and decrypted. Zfone is available as a plug-in for a number of VoIP programs, including Gizmo [13] and Asterisk [1].

Zfone uses the ZRTP protocol [39], which is an extension to Real-time Transport Protocol (RTP) [31] that provides a key exchange on initiation and subsequent encryption. The key exchange does not rely on certificates, servers, or third parties of any kind; it is strictly peer-to-peer. Nevertheless, Zfone claims immunity from man in the middle attacks [15].

On the plus side, Zfone has a very narrow trust model, making it useful regardless of the level of trust that is granted to operators of the VoIP network. However, because Zfone is point to point, support for conference calls is difficult. It can support encryption of any number of RTP streams, however, each RTP stream is independent. Thus, in a peer-to-peer conference containing four parties, there are six connections and six key exchanges. In a VoIP conference containing six parties, there are 15 exchanges. This $O(n^2)$ rate of growth will likely slow down the system as size increases.

Thus, Zfone provides an attractive trust model, but makes conference calls costly to setup. Its success as a peer-to-peer protocol (much like Skype's) indicates that our system may benefit from such a model.

D.3 WAVE

WAVE [44], [45] is a proprietary, closed-source system that uses a client/server model for VoIP conferencing. That is, endpoints participating in a conference transmit their voice data to WAVE Media Servers, which mix the voice signal and pass it back to users.

WAVE documentation does not mention security; however, WAVE is included in the review of relevant technologies because of its clever use of multicast to avoid network congestion. WAVE claims usefulness in disadvantaged networks under heavy traffic because the multicast model minimizes the amount of traffic on any given link. In short, multicast works by transmitting a single stream of data over a network link when there are multiple recipients on the other side. This way, bandwidth usage is minimized. Our conference model will take inspiration from this and optimize for disadvantaged networks.

References

[1] Asterisk: The Open Source Telephony Platform. http://www.asterisk.org.

[2] Connexion by Boeing. http://www.connexionbyboeing.com (deprecated).

[3] Cygwin homepage. http://www.cygwin.com.

[4] Inmarsat, "The Mobile Satellite Company." http://www.inmarsat.com.

[5] Iridium. http://www.iridium.com.

[6] Jabber: Open Instant Messaging and a Whole Lot More, Powered by XMPP. http://www.jabber.org.

[7] Minimalist GNU for Windows. http://www.mingw.org.

[8] PGP Corporation. http://www.pgp.com.

[9] Pidgin Project. http://developer.pidgin.im.

[10] Python Programming Language. http://python.org.

[11] Skype. http://skype.com.

[12] Tactical Common Data Link (TCDL) Airborne Data Terminal. http://www.l-3com.com/csw/product/specs/Airborne/TCDLAir.asp.

[13] The Gizmo Project. http://gizmoproject.com.

[14] The OpenSSL Project. http://www.openssl.org.

[15] The Zfone Project. http://zfoneproject.com.

[16] X-Lite Softphone. http://www.counterpath.com/xlite-overview.html.

[17] Yet Another Telephony Engine. http://voip.null.ro.

[18] A. J. Simon. Overview of the Department of Defense Net-Centric Data Strategy. DoD CIO/Information Management Directorate.

[19] C. L. Zue. "Modeling and Assessing Secure Voice over IP Performance," MEng thesis, MIT, Cambridge, MA, May 24, 2005.

[20] D. Boneh, C. Gentry, B. Waters. Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys, 2005.

[21] Intel. “Overcoming Barriers to High-Quality Voice over IP Deployments,” 2003.

[22] International Telecommunications Union Telecommunication Standardization Sector (ITU-T). Recommendation G.114: “One-Way Transmission Time”, May 2003.

[23] International Telecommunications Union Telecommunication Standardization Sector (ITU-T). Recommendation G.107: “The E-Model, A Computational Model for Use in Transmission Planning,” July 2002.

[24] International Telecommunications Union Telecommunication Standardization Sector (ITU-T). Recommendation P.800: “Methods for Subjective Determination of Transmission Quality,” Aug 1996.

[25] International Telecommunications Union Telecommunication Standardization Sector (ITU-T). Recommendation G.711: “Pulse Code Modulation of Voice Frequencies” Nov 1988.

[26] International Telecommunications Union Telecommunication Standardization Sector (ITU-T). Recommendation G.729: “Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear Prediction (CS-ACELP),” Jan 2007 .

[27] Internet Engineering Task Force (IETF). RFC 2246: “The TLS Protocol Version 1.0,” Jan 1999.

[28] Internet Engineering Task Force (IETF). RFC 2311: “S/MIME Version 2 Message Specification,” March 1998.

[29] Internet Engineering Task Force (IETF). RFC 2401: “Security Architecture for the Internet Protocol,” Nov 1998.

[30] Internet Engineering Task Force (IETF). RFC 3261: “Session Initiation Protocol,” June 2002.

[31] Internet Engineering Task Force (IETF). RFC 3550: “RTP: A Transport Protocol for Real-Time Applications,” July 2003.

[32] Internet Engineering Task Force (IETF). RFC 768: “User Datagram Protocol,” Aug 1980.

[33] Internet Engineering Task Force (IETF). RFC 791: “Transmission Control Protocol,” Sep 1981.

[34] Internet Engineering Task Force (IETF). RFC 793: "Internet Protocol," Sep 1981.

[35] J. Janssen, D. D. Vleeschauwer, and G. H. Petit. Delay and Distortion Bounds for Packetized Voice Calls of Traditional PSTN Quality. *Proceedings of the 1st IP-Telephony Workshop*, GMD Report 95, pp. 105-110, Berlin Germany, 12-13 April 2000.

[36] M. Kaminski. "Fast Prototyping of Telephony Applications with YATE," June 2006. http://www.oreillynet.com

[37] Network Centric Warfare: Dept of Defense Report to Congress. http://www.defenselink.mil/nii/NCW.

[38] NIST. FIBS PUB 197: The Advanced Encryption Standard. csrc.nist.gov/publications/fibs.

[39] P. Zimmerman, A. Johnston, A. Callas. "ZRTP: Media Path Agreement for Secure RTP," March 2006.

[40] R. Figueiredo. "Secure Group Communication in Dynamic, Disadvantaged Networks: Implementation of an Elliptic-Curve Pairing-Based Cryptography Library," MEng thesis, MIT, Cambridge, MA, September, 2006.

[41] R. Khazan, R. Figueiredo, R. Canetti, C. D. McLain, and R. K. Cunningham "Securing Communication of Dynamic Groups in Dynamic Network-Centric Environments." *Military Communications Conference (Milcom)*, 2006.

[42] S. L. Garfinkel. "VoIP and Skype Security," March 2005.

[43] Takahashi, H. Yoshino, and N. Kitawaki. "Perceptual QoS Assessment Technologies for VoIP". *IEEE Communications Magazine*, pages 28-34, July 2004.

[44] Twisted Pair Solutions. "How WAVE Product Works Overview," WAVE from Twisted Pair Solutions. http://www.twistpair.com

[45] Twisted Pair Solutions. "WAVE for Conferencing," WAVE from Twisted Pair Solutions. http://www.twistpair.com.

[46] XMPP Standards Foundation. XEP-0045: "Multi-User Chat," April 2007.