



B.ENG. IN ELECTRONIC ENGINEERING

PROJECT REPORT

Implementing SIP for VoIP Algorithms

JM-3

Andrew Quinn

50375811

Acknowledgements

I would like to thank John Murphy for assisting me with this project and for valuable input in the organising of the project. I would like to thank Sean Murphy for his unending help during the implementation of this project, and his helpful guidance when times were good and bad. I would like to thank my Dad for his tireless work in proof reading and editing this project. Finally to my Mom, Brother, and Sister thanks for keeping me smiling through it all.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: Date:

Abstract

Internet Protocol (IP) Telephony has many issues that have to be overcome before it can be considered a rival to the existing telephony infrastructure. One such issue is the quality of service. The use of play-out buffering at the receiver helps to improve the quality of Voice over IP (VoIP). A buffering algorithm has been proposed [5] by

Narbutt, which uses a dynamic approach to buffering. This algorithm is adjusted automatically according to an estimate of the network delay. This is more suitable to the changing network conditions usually experienced. The buffer has been implemented using the H.323 signalling protocol.

The aim of this project is to incorporate Narbutt's adaptive buffering algorithm into the Session Initiation Protocol (SIP). SIP has been shown to be much easier to implement and update than H.323. The integration of the algorithm was done using VOCAL, a VoIP software library based around SIP. This report describes IP telephony and the protocols surrounding it, and the software used is also described. The manipulation of VoIP software to implement the play-out buffer and the issues involved in doing this are discussed.

Table of Contents

Acknowledgements.....	ii
Declaration.....	ii
Abstract.....	iii
Table of Contents.....	iv
Table of Figures.....	vi
1. Introduction.....	1
2. VoIP Overview.....	3
2.1 Protocols.....	4
2.1.1 The Session Initiation Protocol (SIP).....	6
2.1.2 RTP/RTCP.....	10
2.1.3 SIP vs. H.323.....	12
2.2 Loss Issues.....	13
2.2.1 Recovery Techniques.....	14
2.3 Narbutt's Algorithm.....	15
3. VOCAL Overview.....	18
3.1 VOCAL.....	19
3.1.1 Structure.....	20
3.1.2 Classes.....	22
3.1.3 SIP in VOCAL.....	22
3.1.4 The RTP Stack.....	25
4. Implementation.....	26
4.1 Understanding VOCAL.....	26
4.1.1 Building and Compiling the System.....	27
4.1.2 The RTP Stack.....	28
4.1.3 Deficiencies in Buffering.....	33
4.2 Anand's Code.....	34
4.2.1 Integrating Old and New Versions.....	36
4.3 Narbutt's Code.....	37

4.4 Integration of Narbutt's and Anand's Updated Code	38
5. Testing	39
5.1 The RTP Stack	39
5.2 Anands RTP Stack	42
5.3 Integrated Code	43
6. Conclusion	45
References.....	46
Appendix 1.....	47
Appendix 2.....	48
Appendix 3.....	49
Appendix 4.....	50
Appendix 5.....	51
Appendix 6.....	52
Appendix 7.....	53
Appendix 9.....	54

Table of Figures

Figure 1 Example of VoIP Call	3
Figure 2 Sip Procoesses	6
Figure 3 SIP Invite Message	8
Figure 4 RTP Packet Format	10
Figure 5 Diagram of Delays [5].....	16
Figure 6 Buffering Delay vs. Percentage Loss Rate [5].	17
Figure 7 High Level View of VOCAL System [8].....	18
Figure 8 VOCAL Server Overview [8].	20
Figure 9 SipSet Configuration Windows [7].....	22
Figure 10 Cefficient Input used in Narbutt's code.	37
Figure 11 Vocal System Hierarchy (Appendix1)	44
Figure 12 RTP Stack Window 1	45
Figure 13 RTP Stack Window 2.....	45
Figure 14 RTP Stack Window 3.....	46
Figure 15 Anand's Receiver Window	49
Figure 16 Anand's Transmitter Window.....	49

Chapter 1

1. Introduction

The ability to communicate properly over long distances has become an integral part of society today. Businesses are expanding to different regions in the world, but need to keep the same deadlines. This means it is necessary for employees in two different regions to communicate with each other over long distances, cheaply and trouble free. The public switched telephone network (PSTN) has developed itself to accommodate these requirements.

The internet has become a very popular means of communication in a very short period of time. It was set up as a network where people could share files and access other peoples work. It has since established itself as a massive communications infrastructure that provides many services such as electronic mail. In the recent years it has further developed itself into providing Internet Telephony or Voice over Internet Protocol (VoIP). This allows users to make voice or video calls over the internet. All the user needs is a computer with a network connection, a soundcard, and a microphone.

VoIP enables a lot of big companies to combine their communications and their networking infrastructures. This is the biggest advantage that VoIP has over the regular telephone system. It means that voice and multimedia services are joined together. This means that a number of calls can be made on the one line, as well as having a multimedia broadcast. The fact that you are putting elements that would use one line each, down a single line, means that costs are significantly cut in the management and leasing of lines. There is even no need to change the communication infrastructure that already exists in the company. The companies PABX (private automatic branch exchange) only has to connect to a VoIP gateway, so IP calls can be made. Although VoIP seems to be taking off more with the corporate market the emergence and interest of the general public with Broadband should mean that IP telephony service could soon be implemented to its full extent in the home environment.

Quality of service, and loss issues are the major factors that are holding the growth of VoIP back. Since the idea behind VoIP is that the data is sent in packets, there are problems such as packet delay and packet loss to contend with. Loss recovery techniques and playout adaptation are crucial in keeping the loss low and the quality of service high. There has been a lot of research into these two techniques, and with the advancements of these fields it is expected that VoIP will be able to compete with the high standard of quality of service we have come to expect from the tradition telephone network. We have though already begun to accept poorer quality, in the form of mobile communications.

There are many different methods and techniques into implementing the schemes mentioned above, and most of these centre around having a buffer at the receiver that controls the playing of the packets as they are received. There are different playout algorithms which define the different buffers. This paper discusses one such buffer using an algorithm that hopes to manage the delays associated with transmitting packets. This in turn will optimize the quality of service while keeping delays associated with buffering to a minimum. Details are given of this algorithm and show how it can be implemented on a particular VoIP protocol, the Session Initiation Protocol (SIP). We detail the software needed to do this and the associated reviewing and integrating of codes.

Chapter two of this report gives greater detail on the technical background surrounding IP telephony including the protocols used and loss issues. It also gives a comparison of the main signalling protocols associated with VoIP. Chapter three describes the software used to implement this algorithm. Chapter four depicts how the software and algorithm codes were implemented. Chapter five details the testing that was done and gives analysis of this. Finally conclusions are drawn up in chapter six.

Chapter 2

2. VoIP Overview

The fact that the internet, and connection speeds have developed so much means that VoIP has changed from being a way to provide a voice channel, between two hosts, to being incorporated with other protocols so that it gives tools that can provide integrated voice, and data. The services can be integrated with information on the web to provide a wide high class communication service. Consider the situation shown in the figure 1. A user (PC1) is sitting at his office desk doing paper work. The user's desktop computer gives a beep like an alert tone. The user looks up and sees there is a voice call from a colleague (PC2). The user accepts the call and the two start their video conference. The original user may decide to add a Powerpoint document or Microsoft word document to the conference, he does this and both users can view it. User two then decides that another caller needs to be included in this conference so he gives the IP address or telephone number of user three to his application. User three (PC3) again receives an alert tone on the desktop computer requesting the user to join the audio conference. The user accepts and joins in on the multicast, with both the original users being able to hear user three, and examine the relevant data.

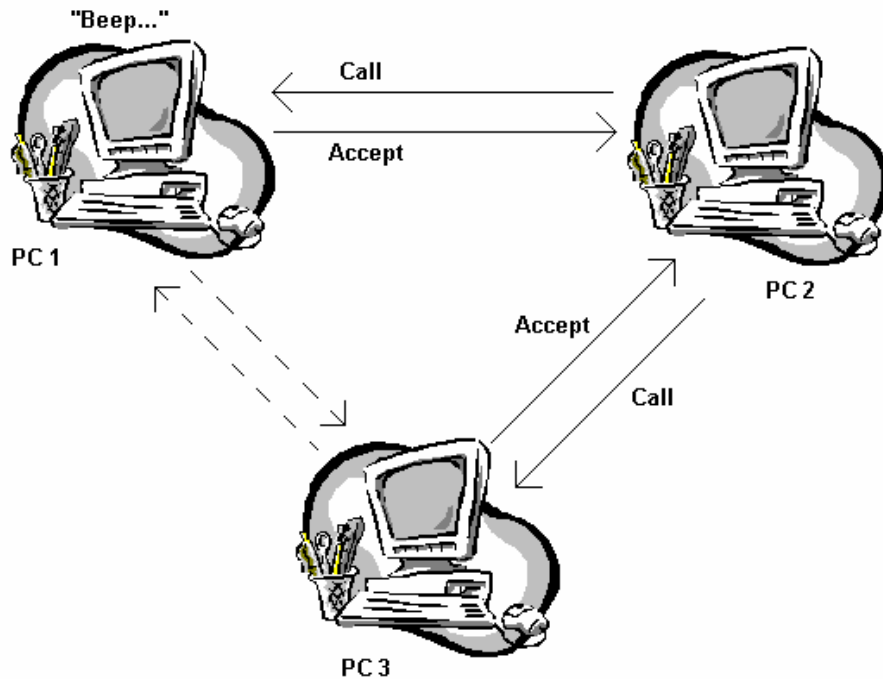


Figure 1 Example of VoIP Call

The situation above is now a reality with the development of VoIP. For the functionalities described above to be provided, VoIP has to be implemented using various protocols.

2.1 Protocols

There are a number of different protocols associated with VoIP, that provide for signalling, quality of service, and media transport. Shown below is a list of the types of protocols and the protocols themselves.

Signalling Protocols

- H.323
- SIP
- SDP

Quality of service Protocols

- RSVP
- RTCP
- RTSP

Media Transport Protocol

- RTP

These protocols are at the heart of any IP session and integral in keeping VoIP to a good standard of service. The protocols that are important to this project are described in sections 2.1.1, 2.1.2, and 2.1.3. A brief explanation of the other protocols is now given.

i) H.323

H.323 provides for call connection, call management, and call termination, in a VoIP session. It is a signalling protocol recommended by the International Telecommunications Union (ITU). The ITU describes and defines its own set of protocols that differ from those of the Internet Engineering Task Force (IETF). H.323 includes H.245 for control, H.225.0 for connection establishment, H.332 for large conferences, H.450.1 H.450.2 and H.450.3 for supplementary services, H.235 for security, and H.246 for interoperability with circuit-switched services [2].

ii) SDP

The Session Description Protocol is a signalling protocol that is used to describe multimedia sessions for VoIP. It provides information about the media streams being transmitted including the number and type of each media. Other information it provides is the payload type that can be transmitted, port numbers, and initiator information including name and contact number. This information is fed back to the users in a textual manner. An SDP message is often contained in a SIP invite message, discussed later on in this report.

iii) RSVP

The Resource Reservation Protocol is not strictly a quality of service protocol. RSVP handles routing and, as the name suggests, reservation of resources. The resources include bandwidth, grade of service, carrier selection, and payment selection. The grade of service and bandwidth relate it to being a quality of service protocol. These reservations can take place either before or after the data begins to be transmitted. Due to the complexity of RSVP, because of extensive features, it is becoming redundant. It has been proposed that the services be completed using Real Time Control Protocol (RTCP)

messages. An RTCP message can be modified to contain an additional field that would specify the desired grade of service [3].

iv) RTSP

The Real Time Streaming Protocol is used to control video and audio media across a network. It gives the user “VCR” like controls over this media [3], by controlling a stored media server. It instructs the server using these controls what to do with the media. This is useful for voicemail in IP telephony, or recording a video or audio conference so that it can be listened to again in the future.

2.1.1 The Session Initiation Protocol (SIP)

SIP is a very important signalling protocol, which was designed for VoIP. It provides the necessary tools for location services, call establishment, call management, and call termination. SIP does not classify the type of session that is set up, so it could just as easily set up an audio call with or without other data. SIP is similar in syntax to Hypertext Transfer Protocol (HTTP), requests are generated by one host and sent to another. A SIP request contains a header field that gives details about the call, and a main body which describes the media being used. There are several different methods that a SIP message can try and implement, these are described later in this section. The above functions are accomplished using two components a user agent and network servers. The user agent is used by the user to initiate and answer a call. The SIP servers ultimately control whether the request gets sent to the appropriate user or not. There are two types of SIP server, a description is given of each of them.

Proxy Server

This server receives particular request, processes it and then forwards it onto the relevant server. In some cases the proxy server will make small alterations to the header field of a SIP request. A proxy server has no way of knowing whether the next server to receive the request is a proxy server or a redirect server therefore request can traverse many servers [4], on the way to its intended receiver.

Redirect Server

A redirect server also receives requests, and processes them. However instead of the server forwarding the request onto the next relevant server, it sends back the relevant information needed to find the appropriate server, to the original proxy server. Therefore redirecting the proxy server to the address of the next server needed to process the request.

Consider the VoIP situation portrayed in the previous section, shown below in figure 2 is a diagram of what SIP is doing in the background. User one decides to make an IP call to user two. User one enters an IP address or email address, lets say `Andrew@yahoo.com`, into its application and presses send. A request is then sent to the proxy server (1), and this server looks up the name `yahoo.com`. It then forwards this request to the server that handles requests for `yahoo.com` (2). The server then looks up and recognises that Andrew is a user, but he is currently not at this location, but sees a note that data should be forwarded to `Andrew@dcu.ie`. This server then redirects the original proxy server to try this address (3). The proxy server then sends this request to the “`dcu.ie`” server (4). This server looks up its database (5,6), and sees that it has a user `Andrew@dcu.ie`, but the user is actually known as `Andrew@eeng.dcu.ie`. Therefore the main DCU server sends the request to the engineering server (7). The engineering server then finds where the user is logged in, and sends the request to the user at the desktop (8). The user then accepts the request and a VoIP session is opened (9,10,11,12).

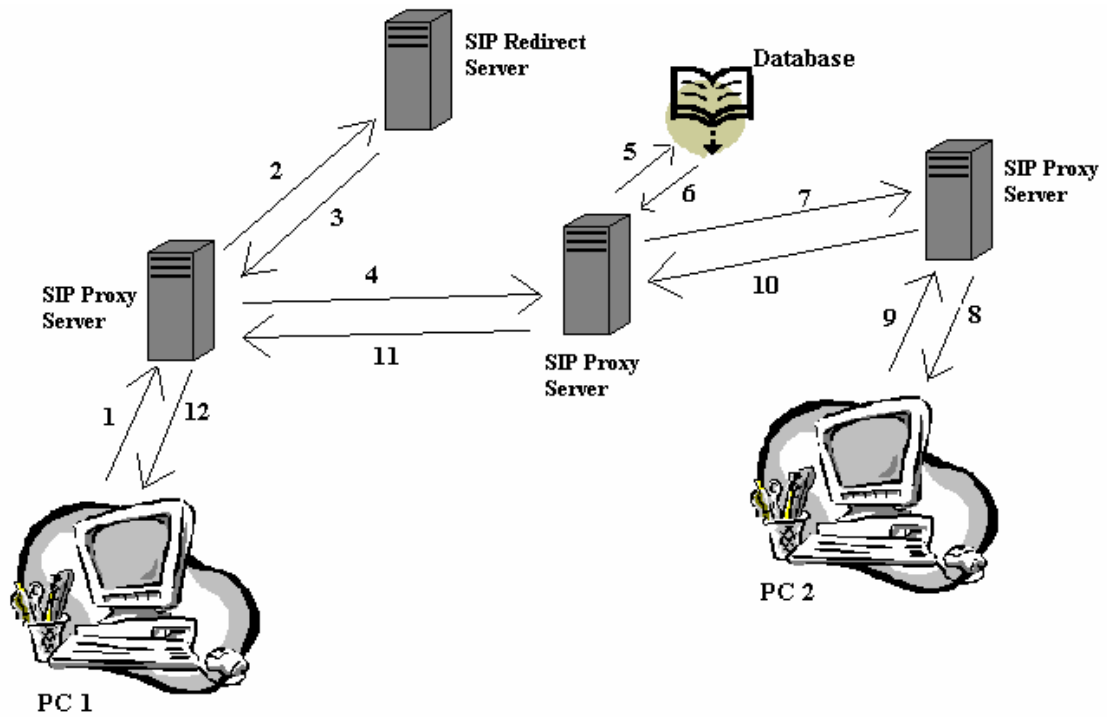


Figure 2 SIP Processes

In a SIP request, like the ones that would be used in the situation above, the header field contains information on the sender, including numbers and addresses. It also gives information on call services, in general the header field describes the call being set up. The body of a SIP message describes the media content that is being sent. The body is usually an object described by a particular protocol. SIP defines several different request methods these are explained.

INVITE

The INVITE request is used to invite a user to participate in a VoIP call. The header field usually consists of the addresses of the person calling, and the intended receiver. It gives the subject of the call, the features to be used, and preferences for how the call should be routed. The body of the INVITE request contains an object that describes the media content of the session. It gives information on what ports to be used and what protocols are used when sending media. An example of an invite message is shown in figure 3 [4]. As can be seen the body is an SDP description where,

v = Version, **o** = ID for the session, **s** = Note, **c** = Session Address, **t** = Start and stop times of session (0 = inf), **m** = media stream type, **a** = attributes.

```
INVITE sip:ann@lucent.com SIP/2.0
Via: SIP/2.0/UDP 131.215.131.13;maddr=239.112.3.4;tll=16
Via: SIP/2.0/TCP 10.0.1.1;received=128.13.44.52
From: John Smith <sip:jsmith@lucent.com>
To: Arun Netravali <sip:ann@lucent.com>
Subject: Raise
Call-ID: 132059753@mypc.domain.lucent.com
Content-Type: application/sdp
CSeq: 4711 INVITE
Content-Length: 187

v=0
o=user1 51633745 1348648134 IN IP4 128.3.4.5
s=Interactive Conference
c=IN IP4 224.2.4.4/127
t=0 0
m=audio 3456 RTP/AVP 0 22
a=rtpmap:22 application/g723.1
```

Figure 3 SIP Invite Message

OPTIONS

OPTIONS is a signalling message that gives information about the capabilities of the caller, i.e. if the user can receive both data and voice. It has nothing to do with setting up the call.

ACK

ACK is used to maintain reliable message exchanges. This is used in response to invitations.

CANCEL

Cancel terminates a particular request but doesn't terminate the call. Let's consider the case where two users were having a conference and they decided to invite another user three in, but as they were inviting the user three they realised there was actually no need for him to be included. In this case they would send a CANCEL request. This request appears at user three's end advising him to reject the call, but it doesn't actually stop him from answering it. If though the user three had already accepted the request, before the CANCEL command could be sent, it would have no effect.

BYE

The BYE request terminates the connection between two users in a VoIP call.

Although the functions described above make SIP a very useful protocol for setting up and managing VoIP calls, it still needs to implement other protocols to allow it to transport the media, and to give information about how this data is being controlled.

2.1.2 RTP/RTCP

The Real Time Transport Protocol (RTP) is a protocol that supports the transport of real time media. It basically provides the necessary information for voice and media to be sent correctly between end points. RTP uses information established by signalling protocols to transport the data from one user to another. The packets are then sent over the network and recovered at the receiver end. This is more complex than it sounds, because RTP must account for losses during transmission, therefore it needs to include some vital information in its packets. Like a SIP message, the RTP packets are made up of a header and a payload. When a packet needs to be sent RTP formats the payload and then adds its header to it.

The RTP header is just as important as the payload that it is attached to. The header provides vital information needed by the receiver to permit it to reconstruct the media. Figure 4 [3], shows what a typical RTP header looks like. As can be seen it contains information on:

- **Payload type** – The type of payload being sent
- **Sequence Number** – Used in reordering packets at receiver
- **Source Information** – List of sources that contributed to Information
- **Frame Identification** – The beginning and end of each frame
- **Timestamp** – Used to synchronize the packets when they are received

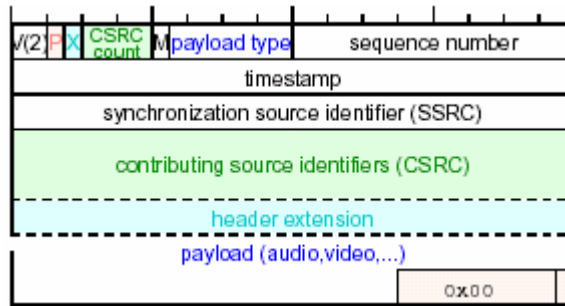


Figure 4 RTP Packet Format

All the information portrayed in the RTP packet is crucial in counteracting the different delay effects associated with packet transport across a network. To further assist in controlling the timing associated with these delays RTP has a companion protocol called the Real Time Control Protocol (RTCP).

RTCP is a protocol that provides additional information on top of the information portrayed by RTP. RTCP enables feedback to be sent to the transmitter about the quality of the information being collected at the receiver end so that the transmitter may modify its behaviour. Each participant in a session periodically sends RTCP packets to the other users. Each packet contains information on:

- **Identification** – This is additional information containing phone numbers, email addresses, or full names of participants in a session.
- **Quality of Service Feedback** – Users use RTCP reports to let other people know the quality of reception they are receiving. Information includes lost packets and jitter (delays).
- **Synchronization** – It’s important that audio and video are synchronized, so there are no “lip sync” issues. RTCP provides the necessary information to do this.

An RTCP packet can also be used as a BYE packet or can be used to let users know that you are stepping away from your desk for a few moments.

Sending RTCP packets periodically in a small session is fine, but this is not the case as the session gets bigger. Consider the situation were there are hundreds of people taking part at the session, in this case users can’t set up to send RTCP packets periodically, because if everyone does it at the same time the network would get swamped. To

manage this RTCP specifies an algorithm that allows the period to get bigger for increasing groups of people.

As can be seen RTP and RTCP are at the heart of the controlling of media transfer in SIP. SIP though is at the heart of signalling in VoIP. Thus the protocols discussed are key components in the running of high quality IP sessions.

2.1.3 SIP vs. H.323

Signalling protocols and the services provided by them have been discussed in the preceding sections. These protocols are pivotal in IP Telephony, they provide the services for call establishment, the management of the call, including extra services provided, and the termination of the call. There are two main standards that have become most commonly used SIP and H.323, both of which have been described previously. The protocols illustrate two different approaches to the same signalling challenge, each one with advantages and disadvantages. This study aimed to show that it is very beneficial that the buffering algorithms and loss recovery schemes mentioned above can be implemented using SIP. The main differences between SIP and H.323 are as follows:

- H.323 defines hundreds of elements, while SIP has only 37 headers, each containing values and parameters, that contain more information than in H.323
- H.323 uses a binary representation for its messages, SIP encodes its messages using text, in a similar manner to HTTP. This means that H.323 needs special code generators to decode its messages, whereas SIP decoding is quite simple, even if using high level programming languages. This also means that debugging SIP is much more straightforward.
- As seen in section 2.1, H.323 has many protocols augmenting it, SIP has very few and is based on single requests
- SIP allows for older headers and features defined in earlier editions to disappear, when they are not needed or updated. This doesn't occur in H.323, therefore as more features are updated and added the size of the codes gets bigger and bigger.
- The feature names in SIP are hierarchical and new features are registered with the Internet Assigned Numbers Authority (IANA). This means anyone can define new features, and compatibility is kept across different versions. This is not the case with H.323.

- IP Telephony demands a large number of different functions to keep it up to date. These functions though will change over time, and it is necessary to have them as separate modules so they can be easily taken out and updated. SIP applies this theory much better than H.323.
- H.323 was designed for use on a single Local Area Network (LAN), but there are now wider locations and domains. H.323 has difficulty adapting itself to the size of networks today.
- Crucially, H.323 doesn't provide as good quality of service functions as SIP.

In general SIP provides a comparable set of services, with more flexibility than compared with H.323, and it does this in a less complex manner. SIP is also much easier to update, and implement over large networks. This means that SIP will be able to evolve easily as the internet and VoIP are developed to become the backbone of global communication.

2.2 Loss Issues

As with any new technology there are always issues that slow down its development. There are many factors that affect the quality of service of VoIP including infrastructure, bandwidth, packet delay, and packet loss. These factors have to be constantly addressed if IP telephony is to become a serious rival to the regular telephony system. The last two of these factors are the most difficult to manage so as to minimize their effects.

Packet Loss

This effect occurs continuously in IP telephony, for the moment it seems that it is unavoidable that some packets will be lost during a session. Previously error concealment was the way to approach this. This entailed inserting a silence into where the packet was due to be played or repeating the previous packet again, which meant silence or stutter in the audio output. Now though, information is being sent in a packet that contains enough information to sufficiently reconstruct a lost packet. This packet is sent periodically so the information remains up to date. This type of adjustment is called Forward Error Correction (FEC).

Packet Delay

Playout delay is the delay associated from when a packet is sent (spoken) to when it is received and played out. In general we are more concerned about the delay between when a packet is sent, and when a packet is received. This delay is due to network conditions, and varies from one packet to the next, as network conditions change. Using this we can estimate a playout time. The delay jitter is defined as the difference between the biggest delay and the smallest delay. This needs to be kept to a minimum to maintain a good quality of playout. This can be achieved by buffering at the receiver end, although buffering itself does have a small delay associated with it known as the buffering delay. There is a trade off between the two loss effects mentioned above. To completely ensure that no packets are lost we use a big buffer, this means that there will be huge delays. Alternatively to completely cut down on the delays we have a very small buffer but there would be a lot of packet loss.

2.2.1 Recovery Techniques

There are two main ways to recover lost data either at the transmitter or at the receiver. The main transmitter technique is Forward Error Correction (FEC). This involves data representing the previous packet being sent along with the present packet. This data is then used by the receiver if the previous packet is lost. Another transmitter based technique is the resending of packets. The receiver requests which packet is needed and the transmitter resends it. There are long delays associated with this therefore it is inappropriate for VoIP.

Buffering algorithms at the receiver have been designed to smoothen out the effect of delay jitter. These techniques are based around the associated network delays. The problem for a buffering algorithm is to find out the correct time to playout the packets so that the loss is kept to a minimum. To do this they use an estimation of the network delay to calculate an appropriate playout time. In past buffering algorithms all packets would be given a delay equal to the longest network delay suffered by a packet in the network. Therefore all the packets would be played out properly altogether. A delay like this could become quite large in a real scenario as packets may become completely lost or suffer huge delays. This would mean huge delays at the buffer, and it would mean that the

receiver and transmitter would find it extremely difficult to interact with each other. In general delays are kept to a minimum and it generally accepted that the loss of a few packets doesn't have a great effect on the audio that is to be played out.

For a buffer to keep the playout time to a minimum it has to have some way of defining the delay that the packets should be given in the buffer, that would allow a good quality of audio being played. Early buffering algorithms have kept this delay constant during the session, once it has been calculated at the beginning. Network conditions can change during a session though, thus rendering the parameter and this particular approach ineffective.

Adaptive buffering algorithms are required to overcome this problem. They monitor the network delay so that the appropriate adjustments can be made to the playout deadline during the session, not just at the beginning. Due to the fact that packets have to be physically played periodically, this calculation has to be performed during silence periods in the audio. This means there will be a constant playout deadline during any given talkspurt. Most adaptive algorithms use voice detection to know whether there is a silence or not.

The algorithm described in this report uses voice detection to adjust the playout delay during these silences. It gives good ratio of buffering time to loss ratio for packet playout.

2.3 Narbutt's Algorithm

As explained earlier buffering is very important in maintaining a good quality of service during a voice over IP connection. The associated algorithms discussed so far are based around the mean and variance of the delay [5], and the fixed weighting factor α . The equations based on previous work [6], are shown below:

$$\begin{aligned}\hat{d}_i &= \alpha \cdot \hat{d}_i + (1 - \alpha) \cdot n_i \\ \hat{v}_i &= \alpha \cdot \hat{v}_{i-1} + (1 - \alpha) \cdot \left| \hat{d}_i - n_i \right|\end{aligned}$$

Narbutt proposes using the same equations, but with a variation in α . This parameter has a big impact on the convergence of these estimations. In early trials, the weighting factor is fixed and chosen to be high, to limit sensitivity of the estimation to short term packet jitter [5]. This high value is at $\alpha = 0.998002$ [5]. It has been shown that this high value of α is only good in situations where network conditions are stable, when the delay and jitter are constant. When network conditions change though, and there are increases or decreases in the delay, smaller values of α are more suitable.

To counteract these changes in the network Narbutt proposes to use an adaptive α . The value of α will be changed depending on the variation of the network delays. This new, dynamic parameter α , will be recomputed with each incoming packet. It will then be used to perform continuous estimations of the network delay, as stated in the equations. This means that α will be set high when end to end variations are low, and set low when end to end variations are high.

The new α that Narbutt proposes is known as α_i , and is described as:

$$\alpha_i = f(\hat{v}'_i) \quad [5]$$

The variable \hat{v}'_i is a new estimate of the variance of the end to end delays between the source and the destination. The function $f(\hat{v}'_i)$ was chosen experimentally to maximise the performance of the algorithm over a large set of network traces [5]. This means that the equations defined above now become:

$$\begin{aligned} \hat{d}_i &= \alpha \cdot \hat{d}_i + (1 - \alpha_i) \cdot n_i \\ \hat{v}_i &= \alpha \cdot \hat{v}_{i-1} + (1 - \alpha_i) \cdot |\hat{d}_i - n_i| \end{aligned} \quad [5]$$

Narbutt also changes the play out time, due to the fact of the changes in the above calculations. The play out time is the time at which the first packet in a talkspurt is played at the destination. This calculation has been changed to include the new variance of the end to end delays between the source and destination.

$$p_i = t_i + \hat{d}_i + \beta \cdot \hat{v}_i \quad [5]$$

Where β controls the delay/packet loss ratio.

The larger this β coefficient the more packets are played out at the expense of longer delays. Any packets, in the same talkspurt, played out after this packet, are played out with a play out time of:

$$p_j = p_i + t_j - t_i \quad [5]$$

This means that during a given talkspurt the same play-out delay will be used all the way through the talkspurt. It could then be changed for the next talkspurt, but only at the end of the current talkspurt, i.e. a silence. The variation of this play-out delay between talkspurts introduces longer or shorter silence periods between consecutive talkspurts. Narbutt uses voice detection to find the beginning of a talkspurt or a silence. The silence is when α is changed. Shown in figure 5 is a diagram outlining the parameters used in play-out time estimation, described by the above equation.

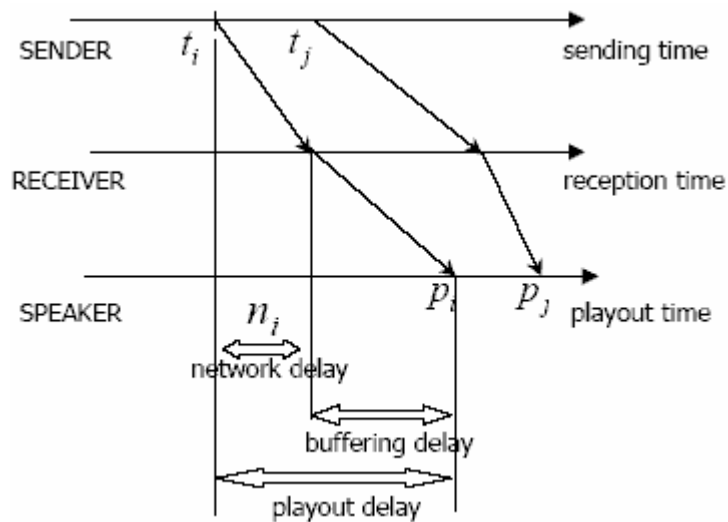


Figure 5 Diagram of Delays

Narbutt's algorithm has so far been implemented in the H.323, described earlier. The implementation of the algorithm using SIP could be beneficial as SIP becomes more popular. Narbutt uses a free source code to implement his algorithm [8], this source code is not unlike the VOCAL source code described in chapter 3. The H.323 source code isn't organised as well as the corresponding VOCAL code. This makes more difficult to distinguish exactly what is being implemented at a particular time, in H.323 files.

Narbutt has completed a rigorous amount of testing on this algorithm. It has been tested using simulations on a network emulator and on a real network itself. Figure 6 [5], shows a plot recorded when the dynamic α was simulated over a real network. The plot shows the delay vs. loss rate for both the algorithms. The dynamic α algorithm is shown in blue

the original algorithm is shown in red, with values of α at 0.7, 0.8, 0.9, and 0.996 in each plot. It can be seen that the adaptive algorithm performs much better. On average its buffering time is lower as is the packet loss. The original buffer with the value of 0.996, cuts down the loss rate substantially, but it gives much longer buffering times. The plot on the left is with hang-over time and the plot on the right is without hangover time. Hangover is the time a voice detection system elongates talkspurts by.

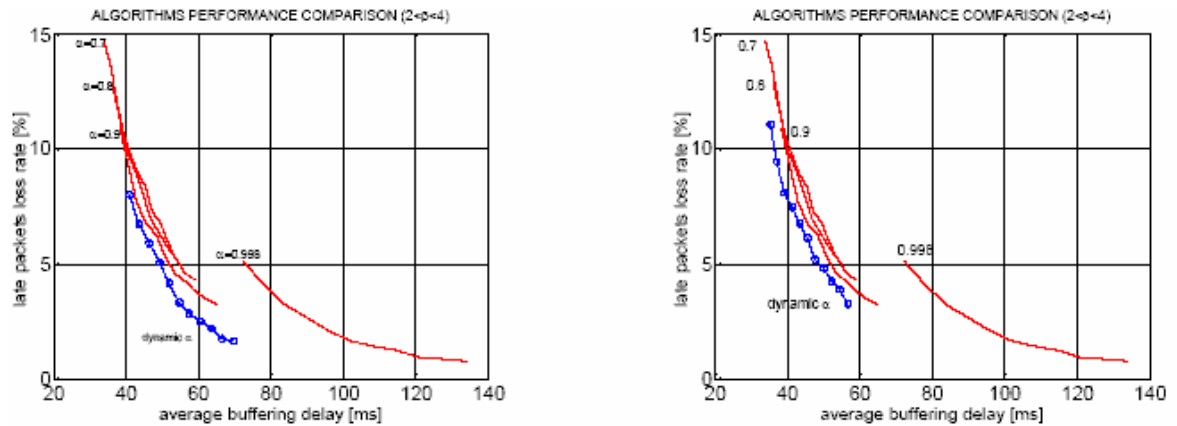


Figure 6 Buffering Delay (x-axis) vs. % Loss Rate (y-axis)

As can be seen the adaptive algorithm performed very well when compared with the standard playout algorithm, with different values. It gives the least amount of package loss, while maintaining a minimum buffering delay. Narbutt has proved that the adaptive play-out buffer algorithm predicts and performs much more effectively than the standard buffer algorithm with a constant α .

Chapter 3

3. VOCAL Overview

This chapter provides an overview into some of the software and coding used to simulate IP telephony. It describes in detail the Vovida Open Communication Application Library

(VOCAL), the specific software used in this research project, including its structure and its classes. It describes the individual components in VOCAL that were used in the project, and the coding structure involved with them.

3.1 VOCAL

VOCAL is a free package available from Vovida[8]. The VOCAL system is a network of servers which provide VoIP libraries which enables SIP based applications to be developed. Off-network calling allows users to connect to other parties through the internet or the switched telephone system. Figure 7 [8] shows a high-level view of the VOCAL system.

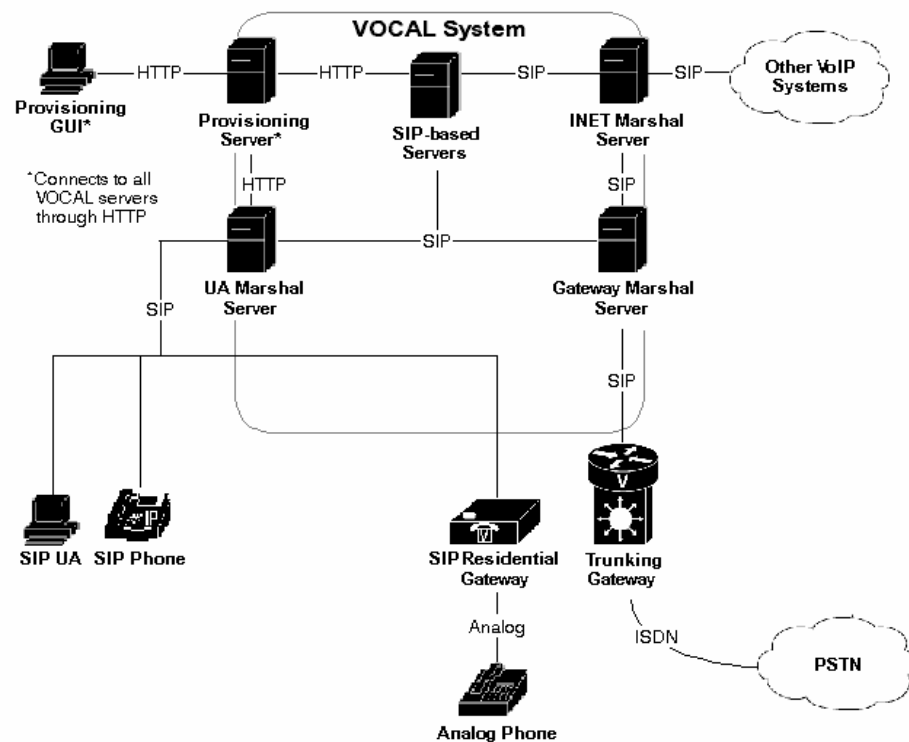


Figure 7 High Level View of VOCAL System

The VOCAL system can be set up as a whole, or as separate components and stacks, which can be built and compiled independently. This allows complete freedom in developing the VOCAL system so that it fits the requirements needed to simulate the users system.

As VOCAL is freely available over the web and its codes are written in C++ it is constantly being updated and changed. This means that while the basic VOCAL system may not offer a specific piece of software or code, a quick check through the online archives may reveal that someone has already edited or written a new piece of code to perform the required functionality. The VOCAL system can be deployed on a single host, as in this case, or over multiple hosts. This means VOCAL can be scaled to suit large applications.

The VOCAL system is best deployed on a linux based platform. In this system a terminal window is opened and commands are typed in to set up and maintain the system and its signals. The SIP user agent provides an interface as is seen in Figure 9. This makes it much more user friendly when trying to set up an IP call using VOCAL. There are other graphical user interfaces in VOCAL.

For VOCAL to provide wide ranging libraries and services it has to have a variety of different servers to support and communicate the various protocols.

3.1.1 Structure

The VOCAL system is divided up into a variety of different servers that provides different functions which enable the setup, maintenance, and termination of a VoIP call. Shown in figure 8 are the main servers contained in VOCAL but actually used in the day to day setting up of IP telephone calls.

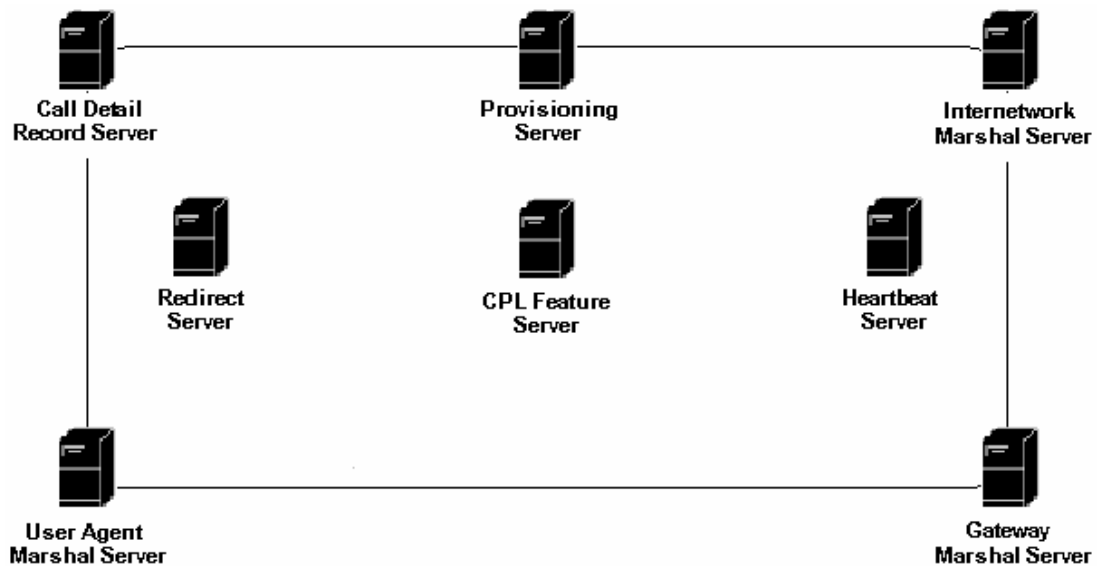


Figure 8 VOCAL Server Overview

A brief description is given below of the various servers shown above.

i) Provisioning Server

The provisioning server stores data and records information about each system user and server module. It then distributes this information throughout the system. This can be useful for a system administrator who needs to constantly monitor what is happening within the system, i.e. who is in the system and what servers are being used.

ii) Marshal Server

The Marshal server is a version of a SIP proxy server, as seen previously in figure 2. This server provides an initial point of contact for SIP signals when they first enter the VOCAL system. It provides authentication, forwarding and billing functions. Such servers would initially look up an IP address or domain name when a VoIP call is attempting to be set up.

iii) Redirect Server

The redirect server is used when setting up a VoIP call. For instance when an IP call is being set up a redirect server may be given an email address or IP address to look up to open a communication link with a user. The redirect server takes this address looks it up in its database, finds where the user is located and redirects the original proxy server to

look up this address. It acts like a directory but also gives the relevant forwarding information if a user is not at its location

iv) Call Detail Record Server

The Call Detail Record Server transmits call data to third party billing services for the purpose of invoicing the correct user. The call data it transmits is obtained from the Marshal server.

v) Heartbeat Server

The Heartbeat server monitors the flow of signals produced by the other servers. This helps the user to know whether or not the other servers are up or down.

3.1.2 Classes

As VOCAL is an extensive package, there are hundreds of classes defined and associated with it. A diagram describing the projects and classes is available in Appendix 1. It is not necessary to go into detail for most of them, but the main classes surrounding the RTP stack are described in section 3.1.4. These classes define specifically the operation of the particular project they are associated with.

The VOCAL system uses several protocols to communicate between its servers and components. However, the main protocol used is SIP. The signalling processes use SIP to both communicate internally between the servers in the VOCAL system, and externally with gateways etc.

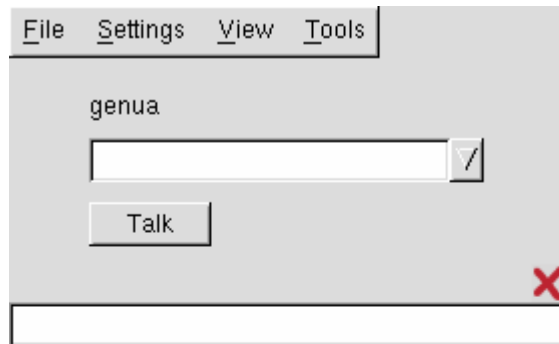
3.1.3 SIP in VOCAL

The Session Initiation Protocol plays a major role in the VOCAL system. The VOCAL system bases itself around the fact that SIP will handle all the communication between servers when setting up, maintaining and tearing down IP calls. SIP and the protocols associated with it were the basis for which the VOCAL system was designed. Most of the VOCAL functionalities were designed with SIP in mind. The SIP stack implements support for both the User Datagram Protocol (UDP) and the Transmission Control

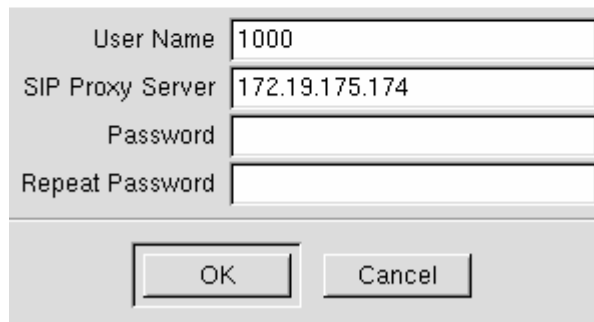
Protocol (TCP). It can also be used to build SIP servers and SIP User Agents. The servers have been discussed in the introduction and those same servers exist in the VOCAL system, namely:

- Location Server – This is a function within the VOCAL redirect server
- Proxy Server – The VOCAL systems includes specialised proxy servers called Marshal servers
- Redirect Server
- Registrar Server – This is also a function within the VOCAL redirect server

VOCAL provides a SIP user agent in the form of SipSet [8]. This is a graphical user agent that uses the SIP stack. Shown in Figure 9 is a snapshot of the window that is used when making an IP call using SIP, from one host to another. The main window is the window that automatically pops up whenever SipSet is launched and it provides a quick way to connect a call, when the configuration and address is known. A SIP URI or a phone number is entered into this field and the talk button is pressed. A green check mark will appear were the red “x” is to inform the user that he/she has been registered with a SIP proxy server.



Main Window



Basic Configuration Window

Figure 9 SipSet Configuration Windows

The configuration window gives you basic control over how the call is made. The user name is the user's contact identifier. The second field is the address of the SIP Proxy server that the user wishes to use. The password is needed if the SIP proxy server requires identification.

This is a brief overview of the function SIP performs in the VOCAL system. Even though the user interface discussed is helpful, the major part that SIP plays is managing the signals between the servers. There are other protocols defined in VOCAL, most of these are implemented using SIP.

3.1.4 The RTP Stack

The RTP stack is part of the full VOCAL system package but can be downloaded, to be used on its own. The stack provides voice channels between two end points. If the RTP stack is compiled and built on two user agents it is possible to set up an audio channel between the two. Information is then made available about the amount of data sent and received. There are other tests available when using the RTP stack that will be discussed later in chapter 6 of this report. The RTP stack and its test are all written in C++.

For the RTP stack to provide these tests, the code needed to implement them is quite complex. The RTP stack has about fifty classes directly associated with it. There is a main class for setting up the RTP session, and it creates instances of the other classes depending on what functionality is needed within the session. The general format of the VOCAL RTP session can be obtained by viewing this “RtpSession” file, it shows when each of the classes is accessed.

There are also “RtpTransmitter” and “RtpReceiver” classes in the RTP stack, these files define how the receiver and transmitter are set up. This means that they contain the information about the timing of a packet when it is sent, and when it is received. These timing parameters are needed to implement buffering, and that is why the emphasis is put on these classes later on in the report.

This chapter has given an overall description of the VOCAL system including its structure and briefly describing some of its classes. It has pointed where the important areas are in reference to the buffering algorithms that are due to be implemented.

Chapter 4

4. Implementation

This chapter describes the methodology needed to complete the research. It describes the how the information about the software and algorithms was gained. An analysis of the code is also given and how the various codes were integrated is also discussed. The solutions to various problems encountered are also described.

The preliminary research for this project was to examine the various aspects of VoIP. The protocols surrounding VoIP were investigated in depth with SIP, RTP and RTCP being the main focus. The protocols were studied to understand exactly what each of them did and how they related to one another. The problems associated with VoIP were researched and different techniques used to overcome them were investigated. The relationship between the protocols and the loss recovery techniques was examined and it was seen how important these protocols were in maintaining a good quality of service in VoIP.

When the information above was understood, the software required to simulate a VoIP network was then investigated.

4.1 Understanding VOCAL

As described in the previous chapter the VOCAL package is a very large package with an extensive code base. The first step was to understand the basic architecture of the system, and how it is used. Understanding the VOCAL code was the toughest aspect of the software. The large amount and apparent complexity of the codes was daunting. This was made easier when it was found what specific VOCAL projects were to be looked at for this research project. These projects were examined to comprehend the basic functionality of each one and to determine which ones implemented RTP and RTCP. It was initially decided that Sipset seemed the best tool to use as it implements SIP and both RTP and RTCP. At this stage it was not fully known if the separate components of

the overall VOCAL system could be downloaded and set up on their own. So it was decided that the whole system should be downloaded and investigated.

4.1.1 Building and Compiling the System

Compiling and building the VOCAL system was complicated. The requirements needed for VOCAL are:

- A Linux system RedHat version 6.2 or later
- Java runtime environment
- JDK 1.2 (needed for SipSet)
- Netscape Browser
- Apache Server
- Enough memory, approx 1GByte of hard disk space

The majority of the requirements above are contained in a Linux based system, with the exception of the Apache server. This was obtained from an external site. It was also noted that two computers were needed to fully test the functionality of the system, both with Linux on them. The Linux system is a powerful system which is relatively simple to understand, but some of the commands and files associated with it can be confusing. For instance a “.zip” extension in Windows is given the extension “.tar” in Linux. Issues like this appear trivial now but added to the difficulty at the beginning.

When the system was downloaded, the installations chapter in the VOCAL guide [9] was examined. In accordance with the guide, the steps needed to check the various parameters of the Linux system were present. This involved checking the loopback address of the computer and verifying the hosts file. The system was compiled and various parameters and choices were entered. Information such as the directories where the Apache server was contained was needed. The user was also prompted to enter various configuration parameters.

The compilation of the system encountered problems surrounding the Apache server. It was taking the address where the server was located, but it had problems when prompting testing on the server. The Apache server was needed to use the Java

provisioning required for programmes, such as SipSet. To be specific, VOCAL had problems restarting the server. After investigating the problem, it was found inadvertently that VOCAL actually provided an Apache server in its files. However, it was unclear how this was compiled and started.

In the process of compiling, VOCAL appears to have changed some of the display properties and this had some adverse effects on the computers. It was observed that after attempting to compile VOCAL on a particular computer, when that computer was next turned on it had considerable trouble booting itself into Linux mode. The exact reason for this was unclear, which was a concern, due to the fact that Linux had to be reinstalled on the computer before anymore work could be done. This meant a loss of any files that were previously worked on. This wasn't too much of a hindrance at the beginning but could have become more serious as the project progresses.

Due to the fact that this error had the capacity to cause major problems in the future, a more thorough examination of the separate projects in VOCAL was done. This revealed that each project could be downloaded separately, so it was decided to investigate what tasks the Rtp stack performed.

4.1.2 The RTP Stack

The RTP stack was crucial in the development of the project. The stack was the next project that was examined after the compiling of SipSet and VOCAL was unsuccessful. It was found that the RTP stack appeared to contain the necessary information that was needed to implement the buffering algorithm mentioned previously. The RTP and RTCP files in the RTP stack were compared with the RTP files in the VOCAL software package. This was to ensure that the files were the same, meaning that in future the stack could be implemented in a full VOCAL network for simulation purposes. The next step was to build the RTP stack on its own. This had to be done before it could be considered as a way of implementing the buffer algorithm. There was no point in finding where the buffering was implemented in the RTP stack only to find there were errors in physically compiling and running the stack.

The code for the stack was downloaded and compiled. The compilation stage was not as complicated as it was with the VOCAL system and there was a helpful “Readme” file provided with the code to give guidance on some of the commands needed. Once the code was compiled it was necessary to run it to see exactly what its functionality was. To do this a number of test files had to be compiled, this was done using the “make” command in the terminal window. The tests were generated from object files and the functionality and implementation of these tests is described in the chapter 5.

When the testing was completed the focus turned to the various classes within the RTP stack. All the classes were examined in brief to see what their function appeared to be and what type of methods they contained. Shown are the classes that provide the backbone to the RTP stack.

- RtpSession
- RtpPacket
- RtpTransmitter
- RtpReceiver
- RtcpTransmitter
- RtcpReceiver

The RtpSession class is the parent of all the other classes. This means that it has access to their variables and procedures. When an RTP session is being set up the main routine in the RTP stack creates an Rtpsession instance at the start and accesses the remaining classes of the stack only through this instance [9]. A description of the more important classes with regards to buffering, and the data structures and variables associated with them, in the RTP stack are discussed.

i) RtpSession

The RtpSession class is very important, as it is the parent class of all the other classes. In essence it controls the RTP session that is being set up. It contains everything needed for an RTP session including RTP/RTCP transmitter and RTP/RTCP receiver. It contains

numerous amounts of methods that control procedures like constructing the stack, changing the send/receive state of the stack, and setting the stack to receive packets. Other methods include checking if it is time to send RTCP packets, getting the amount of bytes sent, and setting what type of payload is to be sent.

Some of the more important variables in the RtpSession class are the variables that correspond to the number of samples in packet, sampling rate, and the packet payload size. Variables that show what type of encoding has taken place on the data, i.e. apiPayloadType, are also important. A lot of the variables described in the RtpSession class are also used in the RtpTransmitter and RtpReceiver classes. When a new instance of the RtpSession class is created, its constructor sets the values in these classes.

ii) RtpTransmitter

This is the class that is used to transmit the RTP session information. It has to implement methods that set up the packets that are going to be transmitted to the receiver. It has to perform functions such as setting up the port that it is going to send the payload out on. It has to decide what the payload is going to be and then construct a header that gives the correct information about the payload and the time it was sent so, that the receiver knows what it is receiving, what it does, and who sent it. There are methods to create a packet, get the port to send the information over, get the previous RTP time and setting the payload format, are defined in the transmitter code.

There are also important variables associated with the transmitter. These variables include the total number of packets sent (packetsSent) and payload sent (payloadSent). The timing variables are also important especially in terms of buffering, variables like “seedRtpTime” and “seedNtpTime” give the RTP and Network Time Protocol (NTP) times for the first packet. The NTP is a general network time and users at either end synchronize themselves with it. RTP time is the timestamp given in RTP headers. All these variables provide small but key information in the implementing of buffers at the receiver end.

iii) RtpReceiver

The RtpReceiver class handles the receiving side of the RTP session. It decides how to receive the information and what to do with the information that has received. This means that the RtpReceiver is a major class when buffering is being considered. The receiver has to determine, in simplified terms:

- a. Who is sending the information, check if source is reliable
- b. What information is being sent
- c. The timing of the information
- d. The order of the information
- e. What to do with the information
- f. Update its network calculations

These functions mean that the RtpReceiver has some important variables associated with it. Variables associated with the timing and positioning of the packets received are very important as these are needed when buffering the information. Some of these variables are described:

- **inPos and playPos** – These variables describe the position where the next packet is to be stored and where the next packet to be played is located.
- **seedRtpTime and seedNtpTime** – These are the RTP and Network Time Protocol (NTP) times of the first packet. One unit of RTP time is equivalent to one unit of the sampling period. There is a conversion from NTP to RTP time in the RtpReceiver class. It takes the form of:

$$\text{NtpTime} = \text{seedNtpTime} + ((\text{rtpTime} - \text{seedRtpTime}) * 1000 / \text{apiFormat_clockRate})$$
$$1000 / \text{apiFormat_clockRate} = \text{Sampling Period}$$

- **gotime** – This is the time when the next packet is to be played.
- **inBuff** – This is an where the data should be buffered when it is received
- **playCycles and recvCycles** – A play cycle is completed when a full length of inBuff has been played. A recvCylce is when the inBuff has been filled.

As can be seen a lot of these variables are self explanatory, but are essential. These variables are constantly updated so that the receiving process can be achieved in the correct way. The RtpReceiver is where the buffering algorithms discussed in this report

are implemented. Therefore, the receiver plays a major role in keeping the quality of service of the RTP session to a required standard. The basic format of the receiver code is shown:

- The Constructor
 - Setting up Constructor Variables that are important in the code
 - Set API format
 - Set Network Format
- Receiving the Packet
 - Checking Source
 - Convert Codec
 - Reorder Packets
 - Update Variables
 - Check if time to play
- Getting the Packet
 - Check for Network Activity
 - Receive Packet
- Update Source
 - Check if new source
 - Check to see if payload is being sent
- Add Source
- Convert from RTP time to NTP time

The main method in RtpReceiver is the “receive()” method. This procedure receives the packets from the network and stores them. It should also buffer the information received. Emphasis was primarily put on finding out exactly what the “gotime” was, it seemed to have something to do with the playout time. The importance and calculation of the playout time was highlighted in previously described algorithms therefore, this provided a good indication of how the buffering in the stack was done. The “gotime” is defined as:

$$\text{gotime} = \text{rtp2ntp} (\text{p->getRtpTime}() + \text{api_pktSampleSize}) + \text{jitterTime};$$

Where rtp2ntp is the function described above; getRtpTime gets the rtp time of the packet; api_pktSampleSize is the number of samples. Also, jitterTime is equal to

jitterseed which itself is equal to jitterNew. JitterNew is a parameter passed in at constructor level. This gotime is set again when initializing the source.

The section of code where it checks if it is time to play the next packet was actually commented out, as shown below. This meant that gotime did not actually do what it was defined to do.

```
/*
// check if time to play packet
if (getNtpTime() < gotime)
{
// cpLog(LOG_ERR,"wait");
//cout <<"w";
return NULL;
}
*/
```

4.1.3 Deficiencies in Buffering

It was eventually found that the RTP stack did not implement buffering at all. The basic parameters were in the RtpReceiver and RtpTransmitter to allow simple buffering to occur but this was never implemented. All the RtpReceiver did was set itself up to receive the information. It would then receive the packet that was transmitted. Next it would take this packet and put it into a buffer, but there was no decision made in the buffer as to when to play out the packet. Essentially, there was no playout estimation implemented in the buffer. They were simply getting played out as they were received.

The consequences of this meant that the assumptions behind the software were unfounded from the beginning of the project. When the project was defined, it was understood that VOCAL implemented jitter buffering of some form. This meant that the direction of the project changed somewhat. As a result, the main aim was reviewed so as to attempt to implement some type of buffer on the VOCAL software. If this was done then Narbutt's algorithm could perhaps be integrated into the software

After it was discovered that the RTP stack did not implement jitter buffering, it was essential to find out how it could be done. This would mean a large amount of in depth C++ coding in a short period, and would also require an excellent knowledge of how to design a buffering algorithm. A project that implemented buffering on a previous version of the RTP stack was found in the form of Anand's code [1].

4.2 Anand's Code

Anand [1] has taken a version of the RTP stack and edited it so that it could implement both adaptive and standard buffering, and has done it so that it implements both sender and receiver based loss techniques. The sender based algorithm is called "fecAlgo", modelled on FEC loss recovery and the receiver based algorithm is called "playoutAlgo". Within the "fecAlgo", the user can choose to have no FEC at all, FEC that repeats the data only, or FEC that repeats the packet. The user has a choice of an auto-regression algorithm, a histogram algorithm and a fixed (α) algorithm in the receiver based "playoutAlgo". The user's choice are then passed to the RtpReceiver in the constructor. These choices are made in the RtpSession.

Anand has done this by making changes to both the header and code files in "RtpSession", "RtpTransmitter" and "RtpReceiver". Anand's has added a variety of functionalities, thus there are numerous new variables. Some of the new variables added to the RtpReceiver were:

- **nPacketsReceived, and nPacketsLate** – These give information on the number of packets received and the number of packets that are late.
- **delayMean, and delayVariance** – These are the two parameters needed for most receiver based buffer algorithms.
- **playoutDelay, and initialDelay** – These represent the existing value of the playout delay and the initial delay respectively. The initial delay is specified by the user.

These were the main variables that were added, but there were many others defined that were needed for the calculation of the talkspurt and the histogram algorithm. New variables were also added to the transmitter to allow it to implement FEC loss recovery. This was also the case for the RtpSession, the variables added here were variables that

indicated the algorithm choice and how the data was to be encoded. An example of this is shown below:

```
RtpSession::setReceiver ( int localPort, int rtpLocalPort, int portRange,  
                          RtpPayloadType apiFormat,  
                          RtpPayloadType networkFormat,  
                          int jitterNew = 5, PayoutAlgo algo = autoregression,  
                          FecAlgo fec = noFec)
```

Anand implemented numerous new methods in the RtpReceiver to allow it to put the algorithms into practice. The functions were:

- i. **void RtpReceiver::setFecAlgo (FecAlgo newFecAlgo)** – This function defines what FEC will be used.
- ii. **void RtpReceiver::setPayoutAlgo (PayoutAlgo Algo)** – Defines what type of playout algorithm will implemented.
- iii. **void RtpReceiver::updatePayoutAlgoData** – This updates the timing associated with the different types of playout algorithm.
- iv. **int RtpReceiver::delayFromHist()** – This is one of many functions associated with the histogram algorithm. This particular function returns histogram delay. Other functions include “**::insertIntoHist**”, “**scaleHist**”, and “**printHist**”.

There are also a number of new functions in the “**receive()**” method. This included information needed to calculate the talkspurt, defining the FEC algorithms and controlling the buffer. The “**play()**” was the main method edited to add new functions associated with the playout algorithms. The jitter in the stack was calculated as follows:

```
Delay = arrival_time - (arrival_time_of_1st_pkt + (sequencenum - sequencenum_1stpkt)  
                      * pkt_duration);  
change_in_delay = delay - previous_delay;  
jitter = jitter + change_in_delay - (jitter+8)/16;
```

Anand's stack was compiled and the relevant testing is described in the "Testing" chapter of this report. This test demonstrated the extra functionality, that Anand has added. As stated before, Anand's code was implemented on an older version of the RTP stack. For Narbutt's algorithm to be used on the up to date code, Anand's functions had to be integrated into it.

4.2.1 Integrating Old and New Versions

The integrating of Anand's work with the new VOCAL version 1.5.0 involved looking at three sets of codes for each class, the VOCAL 1.1.0 code, the VOCAL 1.5.0 code and Anand's code. It was necessary to determine what had changed between the VOCAL 1.1.0 code and Anand's code. It was also necessary to take into account the changes that were made between the VOCAL 1.1.0 code and the VOCAL 1.5.0 codes. This involved careful examination, as variable names were changed slightly in Anand's code. When a change was found in a section, a search through the document was done to see if this variable appeared anywhere else.

Some of the bigger changes and new methods that were found between the two versions of code could be inserted using the cut and paste commands. This, however, had its drawbacks because every variable used had to be checked to see if it was declared properly if it was not done so in the particular method. A lot of the algorithm variables had to be added to the constructor as their values would be passed into it from the header file. This meant that the header file had to implement the exact same changes.

The code was compiled and tested and the results are discussed in the "Testing" chapter. The test would reveal how easy or difficult it would be to fully implement Narbutt's algorithm in VOCAL's SIP based software.

4.3 Narbutt's Code

Narbutt's code was split into two files; one file was called "jitter" and the other "rtp". These files had to be examined to find out how Narbutt's algorithm was put into practice. The information in these files would give an indication of how Narbutt's code could be integrated with Anand's code. The files were obtained from [7] and were edited by

Narbutt to include his algorithm and two others. The “rtp” file is an RTP session contained in one file. It contains information on the transmitter and receiver, and how the data was treated. It gives information on the size of the buffer and reports the delay and jitter calculations back to the user. However, it does not do the calculations behind them. The “jitter” file was a file dedicated to jitter buffering.

Changes to the “rtp” file were minimal. Narbutt has added and defined timing variables that are used to portray information that is needed in the buffering algorithms. This new timing information regards the different time stamps associated with a packet, including the RTP, NTP, and the RTCP reception times. He has added additional lines relaying information to the user on the size of sender and receiver reports.

The “jitter” file contains all the timing and talkspurt information needed for buffering at the receiver. Narbutt has edited the jitter file so that it can implement one of four algorithms depending on the choice. He also allows various other information to be assigned by the user. These coefficients are shown in figure 10.

Coef.txt description			
1			//alg. number
1	30		//per talkspurt or per packet playout delay updating
900	4		//coefficients for the first Ramjee's alg.
600	4	10	//coefficients for the dynamic alpha alg.
0			//fixed playout time
0	0		//packet delay correction
1	25000		//RTCP updating or clock skew removing
0	950	1000	//coefficients for the algorithm that removes the clock skew
0			//removing audio frames from the buffer when timeout
1	1		//generating trace files

Figure 10 Coefficient Inputs Used in Narbutts Code

As can be seen, these coefficients allow the user to control such parameters, such as the values for the different algorithm coefficients. It also allows the user to define if there is packet delay correction or not, and if audio file should be removed from the buffer when their playout time has passed.

Narbutt has inserted many new functions into the file to allow it to implement the buffer algorithms correctly. These functions include reading the coefficient file to see what the different variables are, and defining each of the four algorithms that are to be used. To do

this he uses one function for the H.323 algorithm and one function for the other three. There are variables added to portray the different delays, and the different parameters are defined for each algorithm (the α talked about previously). A key variable was “playoutDelay” as this defined the delay in playing the packets. These functions and variables are included in the methods that were previously there. Methods include:

- **RTP_JitterBuffer::~~RTP_JitterBuffer()** – This is a method that sets up the frames in the jitter buffer and was added to the code by Narbutt.
- **void RTP_JitterBuffer::SetDelay(unsigned delay)** – This method sets the maximum jitter time and the size of the buffer, this was also added to the code by Narbutt.
- **void RTP_JitterBuffer::Main()** – This is the main method where the jitter buffer statistics are obtained. This method was not added to the code by Narbutt. However, he edited this method to a large degree, which included getting the NTP and RTP times from the packet, and using them to decide which frame would be played out next. In this method Narbutt also adds the information on how the playout delay is calculated for each of the different algorithms. This method was the most difficult to understand as it contained a large amount of information, all of which was necessary for Narbutt’s algorithm.

The above information was then compared with Anand’s work.

4.4 Integration of Narbutt’s and Anand’s Updated Code

The key to getting Narbutt’s code working in SIP, was integrating it with the new version of Anand’s code that was incorporated with VOCAL 1.5.0 RTP stack. To do this key variables and functions had to be compared in each code. Although in the end there was not enough time to implement these changes into an actual code, the theory behind it was examined to outline what in general was needed to be done.

As Narbutt’s jitter file only contained information concerning jitter buffering, it meant that only the receiver class of Anand’s 1.5.0 stack would need to be edited, the transmitter and session classes need not be changed, as was the case with the previous integration.

The basic functions for setting the different playout algorithms are quite similar. As stated before Narbutt gives the option of using one of four receiver based algorithms, Anand gives a choice of three receiver based algorithms, but also implements a FEC algorithm as well. This FEC should be left unchanged in the code as it provides extra loss recovery power. Anand also implements an algorithm called “ramjees” algorithm, this same algorithm is implemented in Narbutt’s code, although Narbutt uses the same code for the other three algorithms. The code that Narbutt uses to implement the H.323 algorithm need not be integrated.

Anand uses the “gotime” variable from the RTP stack to decide when to play the packets, Narbutt uses “playoutDelay” to decide when the packets are played. The calculation for Narbutts delay would be the main calculations that would have to be integrated into Anand’s updated code. These include variables that are already in Anands’s code, just labelled as something different. There are new variables though that would have to be added including the clock update variables and the correction variables.

Chapter 5

5. Testing

The testing of the various codes and stacks discussed in this paper was required to show that the implementation of Narbutt’s buffering algorithm in SIP worked in simulated and real life situations. These tests were all conducted on two computers using the Linux environment.

5.1 The RTP Stack

Testing of the RTP involved implementing a series of tests that were built into the VOCAL code. The intended functionality of these tests was described in a text file in the defined directory. The tests had to be built separately after the remainder of the RTP code had been compiled. There were three main tests compiled:

- i. **rtpPlay and rtpRecord** – In this test rtpPlay sends a text file over a port and rtpRecord receives the text file.

- ii. **sampleUsage** – Sends and receives a 160 byte data packet every 20ms.
- iii. **phoneJack** – This was used to test the RTP stack with the quicknet phone jack card.

It was unclear what the sampleUsage test did, as when the tests were compiled, there was no sampleUsage test but a test called “soundcardTest” instead.

The first and main test that was examined was the test that used the rtpPlay and rtpRecord commands. It was described in the test file that the commands should be used as follows:

"/rtpRecord FILE_recv [host]"

"/rtpPlay FILE_send [host]"

These commands appeared straight forward. It required a file to be created in each test directory called “FILE_recv” and “FILE_send” respectively. The “FILE_recv” was to be left empty as this was where the data that was received would be stored. The “[host]” part of the command was the most difficult to use. It was unclear what information it wanted. It appeared to request a host name but when host names were put into each of the commands the test did not run. Different combinations of host names were used with the commands because it was unclear which host name it needed for each command. No combinations of host names worked. It was then decided to use IP addresses; these were tested in the same way. Again it was found that this was the incorrect use for the tests. In the end it was found that a mixture of IP address and host name was the correct way to implement the test. An example of the commands that are shown:

Computer “Proj1” : ./rtpPlay FILE_send 10.10.103.161

Computer “Proj2” : ./rtpRecord FILE_recv proj1

The reason the send command needed an IP address was because it needed to have the exact address to send the information from its port. The receiver simply listened for information coming from the “proj1” computer.

A text file was sent and received with this test. The commands above were used. Appendix 2 shows the windows and different stages of testing. The “rtpRecord” command was executed first. This looped while listening for data to be sent, Figure 12. The “rtpPlay” command was then executed and the data was sent, Figure 13. The looping receiver then received the data, Figure 14 (Appendix 3).

The data (text) sent, in “FILE_send” and received into “FILE_recv” is shown in Appendices 4 and 5 respectively. As can be seen there is data missing at the beginning and corruption towards the end of the file that was received, Appendix 5. The missing data is due to the fact that the data is sent and received in packets of 160 bytes. If the data does not fill up the 160 bytes allocated to the packet, it will not be received. This is what happened to the missing data in this case. The file to be sent was 2169 bytes in size which cannot be divided evenly by 160. This means that a portion of bytes was not received because the packet that it was in, didn’t make up all 160 bytes. Therefore it was not received.

The reason for the corruption was unclear.

The other test programs that appeared to be of some value were the “sampleUsage” and “souncardTest” programs.

The “sampleUsage” test did not appear with the other test files when the make file was run in the RTP stack. This was due to the fact that the test was commented out in the make file. It was uncommented and the stack was built again, this time the “sampleUsage” test appeared with the other tests in the directory.

The help file was a bit unclear in describing what function this test actually performed. It explained that it was a driver for the RTP stack. The test was run and the user was asked for a remote host name and a remote IP address, as is shown in the diagram in Appendix 6. These parameters were entered but an error occurred in the program and it could not be run. The error seemed to centre around the host name. Different possibilities were entered into this field but they all got the same reaction. The reason this might occur is because the help file describes client and server scripts. These are not to be found in the

RTP stack, so this could be the reason for the error, and why the test was commented out in the make file.

The “soundcardTest” was a test that was given no description in the help file but was included in the make file. It was assumed that this test used the RTP stack with the computers sound card. The test was set up on two computers with two microphones attached. The program appeared to do nothing as no files were transmitted, and there was no sound in the earphones. This could have been an issue with the sound card though, as there was some information on the VOCAL website [7], stating that a “quicknet” sound card was needed for this.

5.2 Anands RTP Stack

It was necessary to know what Anand’s stack did so that when his code was integrated with the new VOCAL version the tests that were supposed to be implemented would be known. Anand’s RTP stack was compiled and built in the same way as the VOCAL RTP stack. The compiling and running of the test programs was different however. Instead of Anand having a test directory he had a project directory and this is where the tests were defined. Anand created five different commands to implement his tests. The commands that were looked at in particular were the “tr16bit” and “tr16bit_receive_only”. These commands recorded and sent out audio and received the audio respectively. These commands were used in the same way as the previous test commands, in that the receiver used a host name to receive and the transmitter used an IP address to play to.

A microphone was obtained to carry out this test. This was connected to the transmitter end. The command “./tr16bit_receive_only proj2” was run in the receiver window. This looped waiting to record data sent to it. The “./tr16bit 10.10.103.160” command was then run in the transmitter window. This command set up the transmitter and then set itself in a state where it would transmit whatever noise came in from the microphone. When noise was transmitted a “t” appeared in the transmitter window. To stop the session each window had to be terminated using the “ctrl c” command. The set up and output of each window is shown in Appendix 6. As can be seen the transmitter gives information on the packets sent and the payload sent. The receiver gives information on the jitter latency, and the amount of packets that were lost.

After reviewing the functionality of the commands above, it was decided to use the “./tr16bit” command on both computers. In the previous test it seemed that when this command was run, although it was used for sending, there were fields there that could convey receiver information. It appeared that the command could actually be used for both sending and receiving.

The command was run on both computers, with a microphone headset at each one. The command ran the same way as described above, but both computers could send and receive packets. Depending on which mic you spoke into, that terminal sent the packets to the other terminal which received them and played them out. This is shown **Figures** in Appendix 7. In the figures you can see in the terminal window, a “t” appears when a packet is sent, and a “p” appears when one is received. These can appear at any time depending on if the terminal is receiving or transmitting a packet at that instance. When the test is terminated it gives both sender and receiver information, which can be seen in both the figures.

The noise on the receiver end was not the words that were spoken at the transmitter though. There was just a lot of crackling at the receiver end anytime something was spoken. There were relevant silences at the receiver when there were silences at the transmitter which was a good sign. The reason for the corruption of sound again could be to do with the sound card.

5.3 Integrated Code

Anands code that was integrated with the VOCAL 1.5.0 code had to be built and tested to make sure that the editing was done successfully. There was a dilemma when it came to building the code. It was uncertain which make file to use, Anand’s file or the RTP 1.5.0 file. Both stacks had one make file to build the actual stack and one make file to build the tests. Anands make file was chosen to be used for the tests, but a combination of the two make files was used to build the RTP stack. The RTP 1.5.0 file was the basis for this, with a few additions from Anand’s file.

A build was then run to create the stack using this make file. Errors occurred during this build which can be seen in the **Figure** in Appendix 9. This error focused on Anand's algorithm declarations that were added to the constructor. These parameters were checked in the code, as was the "RtpReceiver" header file. The build was run again but the same error was coming up. Changes were then made to the make file to see if it was the problem, this was not the case, the error kept coming up. The other make files were used but they brought up additional errors on top of the original ones. The portion of the code that the error pointed to was reedited but the errors still came up in the build.

In general it was believed that the errors had to do with how the parameters were being passed from the header file. This could be due to small unnoticed discrepancies when integrating the codes together in the header file and the actual receiver code file itself.

Chapter 6

6. Conclusion

This report has described the main features behind VoIP. It described the various functions and protocols associated with. The quality of service issues associated with VoIP were explained and the techniques associated with controlling these issues were also described. It was shown how buffering at the receiver greatly improves this quality of service.

This report has shown how Narbutt's adaptive buffering algorithm can be implemented using SIP. This was done by describing the VoIP software (VOCAL) that was used, and the buffering deficiencies associated with it. A description of the solution that was found to this buffering problem is given. The report explains how this solution was altered so that it could be used on newer versions of the VOCAL software. The theory behind the integrating of Narbutts algorithm into this new version of the VOCAL code was also described.

Results were shown depicting the various tests that were used at each step. Not all these tests proved to be successful, but they helped to prove how the SIP based VOCAL software implements an RTP session using other buffering algorithms. This gave a clear indication of how difficult it would be to integrate Narbutts algorithm into the software.

To further develop this project a more rigorous examination could be carried out the codes that were integrated together including the header and make files. This would get rid of the errors being experienced when building the code. The actual integration of Narbutt's code into the VOCAL software code could be achieved. When this integration is achieved a variety of tests will need to be implemented to ensure that the algorithms depicted in the code are working properly.

References

- [1] Muthukrishnan, A, “On Incorporating Playout Adaptation and Loss Recovery in VoIP Applications”, Department of ECSE, Rensselaer Polytechnic Institute, New York .
- [2] Schulzrinne,H and Rosenberg,J. “A comparison of SIP and H.323 for Internet Telephony”, Proc. NOSSDAV, Cambridge, U.K., July 1998.
- [3] Schulzrinne,H and Rosenberg,J. “Internet Telephony: architecture and protocols – an IETF perspective” *Computer Networks*, vol. 31, Feb 1999. pp237-255.
- [4] Schulzrinne,H and Rosenberg,J. “The Session Initiation Protocol: Providing Advanced Telephony Services Across the Internet” *Bell Labs Technical Journal*, October-December 1998. pp144-159.
- [5] Narbutt,M. “VoIP Playout Buffer Adjustment using Adaptive Estimation of Network Delays” Proc. of the 18th International Teletraffic Congress (ITC-18), p. 1171-1180, Berlin, Germany, Sept. 2003
- [6] Jacobson,V. “Congestion avoidance and control” in *Proceeding of ACM SIGCOMM Conference*, Standford, 1988.
- [7] Source code available at: www.antd.nist.gov
- [8] Source code available from: www.vovida.org
- [9] “Vocal Installation Guide”, available from Vovida Networks, Inc., www.vovida.org
- [10] Schulzrinne,H and Rosenberg,J. “The Session Initiation Protocol: Providing Advanced Telephony Services Across the Internet” *Bell Labs Technical Journal*, October-December 1998. pp144-159.

Appendix 1

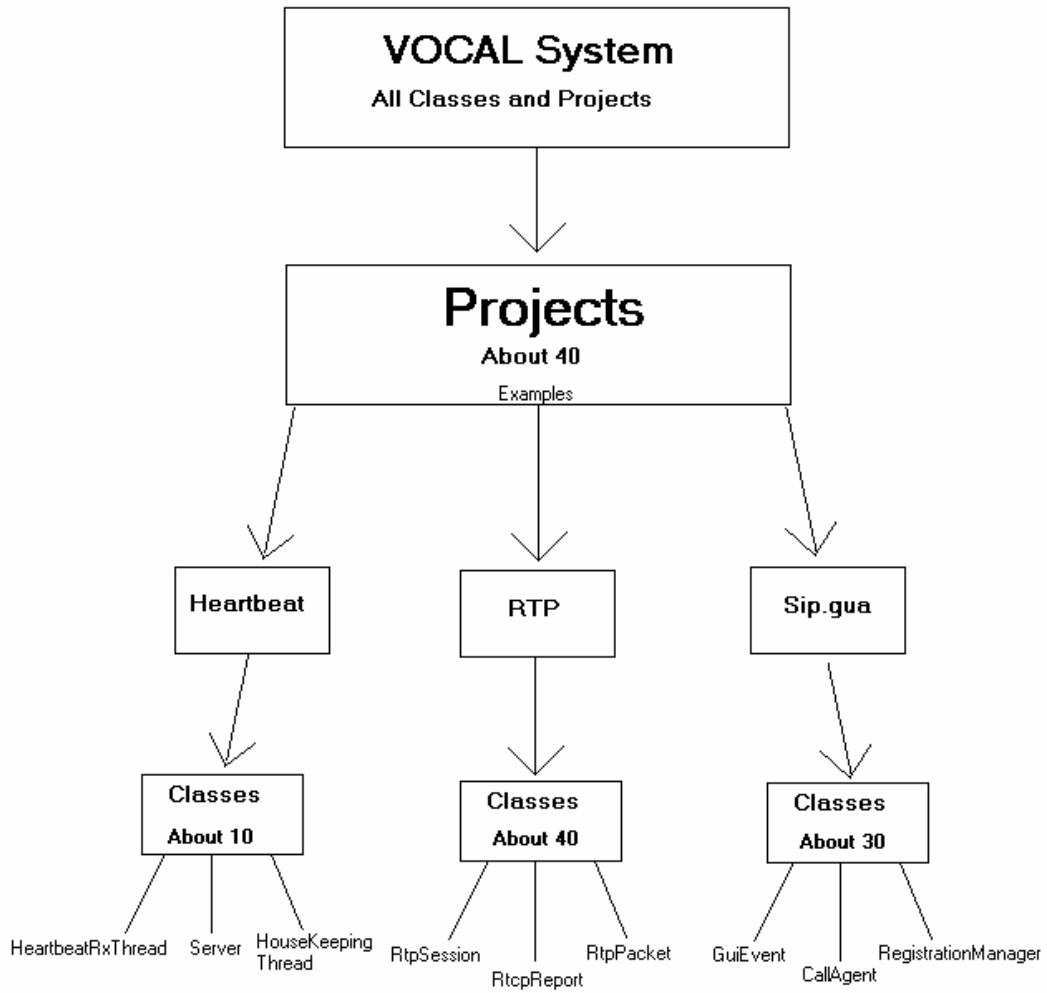
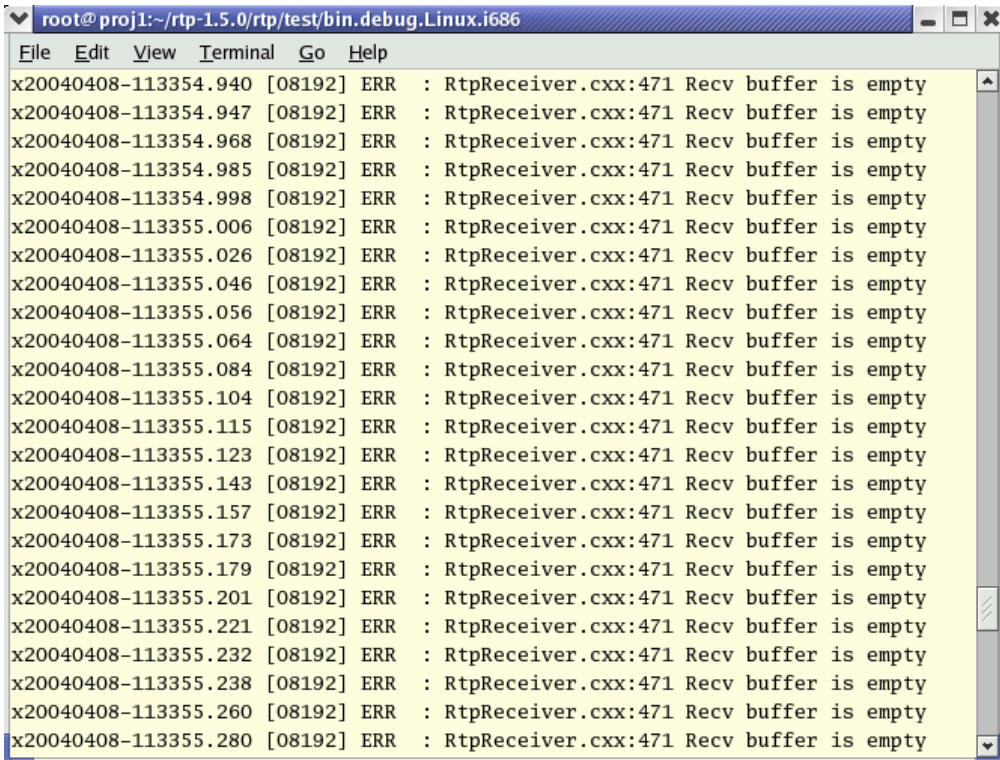


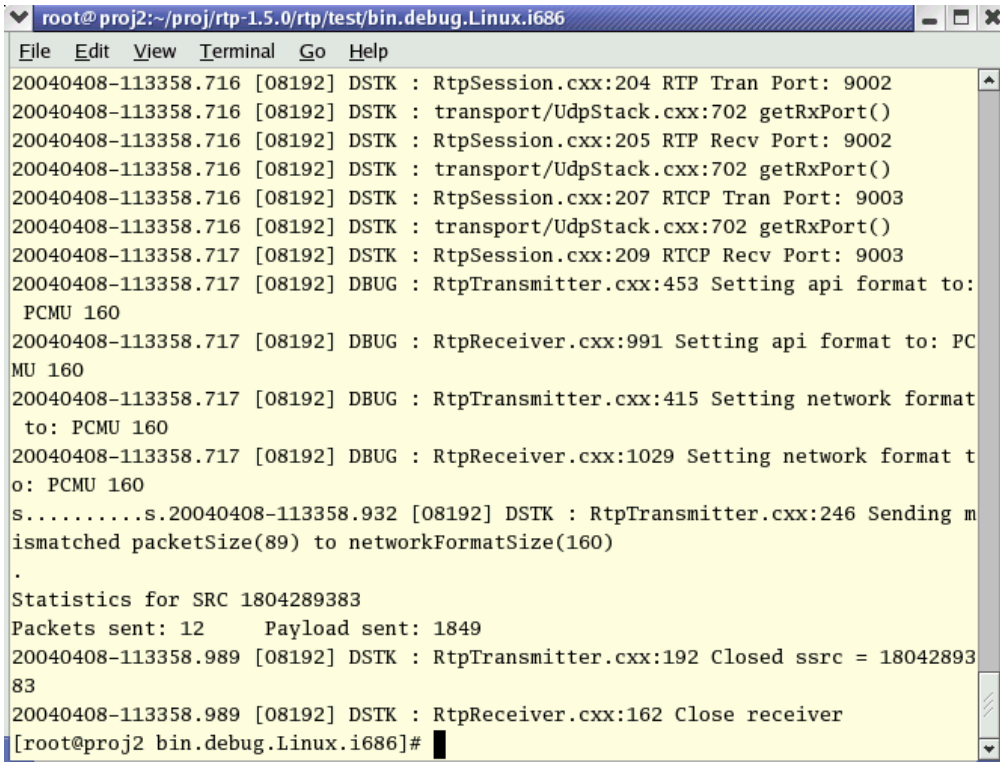
Figure 11 VOCAL System Hierarchy

Appendix 2



```
root@proj1:~/rtp-1.5.0/rtp/test/bin.debug.Linux.i686
File Edit View Terminal Go Help
x20040408-113354.940 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113354.947 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113354.968 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113354.985 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113354.998 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.006 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.026 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.046 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.056 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.064 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.084 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.104 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.115 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.123 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.143 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.157 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.173 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.179 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.201 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.221 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.232 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.238 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.260 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
x20040408-113355.280 [08192] ERR : RtpReceiver.cxx:471 Recv buffer is empty
```

Figure 12 RTP Stack Window 1



```
root@proj2:~/proj/rtp-1.5.0/rtp/test/bin.debug.Linux.i686
File Edit View Terminal Go Help
20040408-113358.716 [08192] DSTK : RtpSession.cxx:204 RTP Tran Port: 9002
20040408-113358.716 [08192] DSTK : transport/UdpStack.cxx:702 getRxPort()
20040408-113358.716 [08192] DSTK : RtpSession.cxx:205 RTP Recv Port: 9002
20040408-113358.716 [08192] DSTK : transport/UdpStack.cxx:702 getRxPort()
20040408-113358.716 [08192] DSTK : RtpSession.cxx:207 RTCP Tran Port: 9003
20040408-113358.716 [08192] DSTK : transport/UdpStack.cxx:702 getRxPort()
20040408-113358.717 [08192] DSTK : RtpSession.cxx:209 RTCP Recv Port: 9003
20040408-113358.717 [08192] DEBUG : RtpTransmitter.cxx:453 Setting api format to:
PCMU 160
20040408-113358.717 [08192] DEBUG : RtpReceiver.cxx:991 Setting api format to: PC
MU 160
20040408-113358.717 [08192] DEBUG : RtpTransmitter.cxx:415 Setting network format
to: PCMU 160
20040408-113358.717 [08192] DEBUG : RtpReceiver.cxx:1029 Setting network format t
o: PCMU 160
s.....s.20040408-113358.932 [08192] DSTK : RtpTransmitter.cxx:246 Sending m
ismatched packetSize(89) to networkFormatSize(160)
.
Statistics for SRC 1804289383
Packets sent: 12 Payload sent: 1849
20040408-113358.989 [08192] DSTK : RtpTransmitter.cxx:192 Closed ssrc = 18042893
83
20040408-113358.989 [08192] DSTK : RtpReceiver.cxx:162 Close receiver
[root@proj2 bin.debug.Linux.i686]#
```

Figure 13 RTP Stack Window 2

Appendix 4

1 BLACK

The lonely sound of a buoy bell in the distance. Water slapping against a smooth, flat surface in rhythm. The creaking of wood.

Off in the very far distance, one can make out the sound of sirens.

SUDDENLY, a single match ignites and invades the darkness. It quivers for a moment. A dimly lit hand brings the rest of the pack to the match. A plume of yellow-white flame flares and illuminates the battered face of DEAN KEATON, age forty. His salty-gray hair is wet and matted. His face drips with water or sweat. A large cut runs the length of his face from the corner of his eye to his chin. It bleeds freely. An un-lit cigarette hangs in the corner of his mouth.

In the half-light we can make out that he is on the deck of a large boat. A yacht, perhaps, or a small freighter. He sits with his back against the front bulkhead of the wheel house. His legs are twisted at odd, almost impossible angles. He looks down.

A thin trail of liquid runs past his feet and off into the darkness. Keaton lights the cigarette on the burning pack of matches before throwing them into the liquid.

The liquid IGNITES with a poof.

The flame runs up the stream, gaining in speed and intensity. It begins to ripple and rumble as it runs down the deck towards the stern.

EXT. BOAT NIGHT STERN

A stack of oil drums rests on the stern. They are stacked on a palette with ropes at each corner that attach it to a huge crane on the dock. One of the barrels has been punctured at it's base. Gasoline trickles freely from the hole.

The flame is racing now towards the barrels. Keaton smiles weakly to himself.

The flame is within a few yards of the barrels when another stream of liquid splashes onto the gas. The flame fizzles out pitifully with a hiss.

Two feet straddle the flame. A stream of urine flows onto the deck from between them.

Appendix 5

ff in the very far distance, one can make out the sound of sirens.

SUDDENLY, a single match ignites and invades the darkness. It quivers for a moment. A dimly lit hand brings the rest of the pack to the match. A plume of yellow-white flame flares and illuminates the battered face of DEAN KEATON, age forty. His salty-gray hair is wet and matted. His face drips with water or sweat. A large cut runs the length of his face from the corner of his eye to his chin. It bleeds freely. An un-lit cigarette hangs in the corner of his mouth.

In the half-light we can make out that he is on the deck of a large boat. A yacht, perhaps, or a small freighter. He sits with his back against the front bulkhead of the wheel house. His legs are twisted at odd, almost impossible angles. He looks down.

A thin trail of liquid runs past his feet and off into the darkness. Keaton lights the cigarette on the burning pack of matches before throwing them into the liquid.

The liquid IGNITES with a poof.

The flame runs up the stream, gaining in speed and intensity. It begins to ripple and rumble as it runs down the deck towards the stern.

EXT. BOAT NIGHT STERN

A stack of oil drums rests on the stern. They are stacked on a palette with ropes at each corner that attach it to a huge crane on the dock. One of the barrels has been punctured at it's base. Gasoline trickles

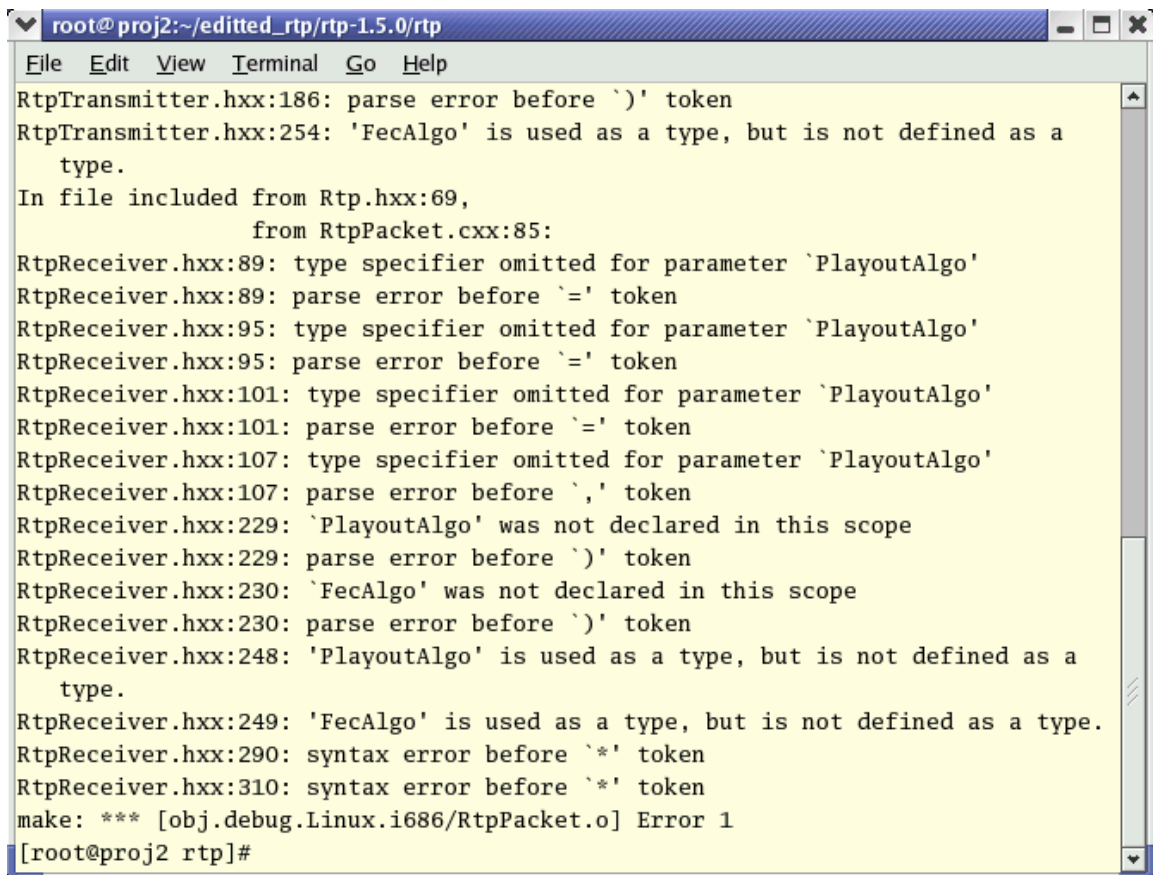
freely
when another
stream of liquid splashes onto the gas. The flame fizzles out pitifully with a hiss.

Two feet straddle the flame. A stream of urine

Appendix 6

```
root@proj1:~/rtp-1.5.0/rtp/test/bin.debug.Linux.i686
File Edit View Terminal Go Help
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]#
[root@proj1 bin.debug.Linux.i686]# ./sampleUsage
Enter remote hostname: proj2
Enter remote port number: 10.10.103.161
20040414-111104.411 [08192] ERR : transport/NetworkAddress.cxx:463 Failed to re
solve proj2, reason:Name or service not known
20040414-111104.457 [08192] ERR : transport/NetworkAddress.cxx:463 Failed to re
solve proj2, reason:Name or service not known
20040414-111104.502 [08192] ERR : transport/NetworkAddress.cxx:463 Failed to re
solve proj2, reason:Name or service not known
20040414-111104.548 [08192] ERR : transport/NetworkAddress.cxx:463 Failed to re
solve proj2, reason:Name or service not known
20040414-111104.548 [08192] ERR sampleUsage: RtpReceiver.cxx:471 Recv buffer is
empty
ERR100 80sampleUsage: RtpPacket.cxx:229: void RtpPacket::setPayloadUsage(int): A
ssertion `size <= getPayloadSize()' failed.
Enter local port number: API packet size: Network packet size: nAborted
[root@proj1 bin.debug.Linux.i686]#
```

Appendix 7



```
root@proj2:~/editted_rtp/rtp-1.5.0/rtp
File Edit View Terminal Go Help
RtpTransmitter.hxx:186: parse error before `)' token
RtpTransmitter.hxx:254: 'FecAlgo' is used as a type, but is not defined as a
type.
In file included from Rtp.hxx:69,
from RtpPacket.cxx:85:
RtpReceiver.hxx:89: type specifier omitted for parameter `PlayoutAlgo'
RtpReceiver.hxx:89: parse error before `=' token
RtpReceiver.hxx:95: type specifier omitted for parameter `PlayoutAlgo'
RtpReceiver.hxx:95: parse error before `=' token
RtpReceiver.hxx:101: type specifier omitted for parameter `PlayoutAlgo'
RtpReceiver.hxx:101: parse error before `=' token
RtpReceiver.hxx:107: type specifier omitted for parameter `PlayoutAlgo'
RtpReceiver.hxx:107: parse error before `,' token
RtpReceiver.hxx:229: `PlayoutAlgo' was not declared in this scope
RtpReceiver.hxx:229: parse error before `)' token
RtpReceiver.hxx:230: `FecAlgo' was not declared in this scope
RtpReceiver.hxx:230: parse error before `)' token
RtpReceiver.hxx:248: 'PlayoutAlgo' is used as a type, but is not defined as a
type.
RtpReceiver.hxx:249: 'FecAlgo' is used as a type, but is not defined as a type.
RtpReceiver.hxx:290: syntax error before `*' token
RtpReceiver.hxx:310: syntax error before `*' token
make: *** [obj.debug.Linux.i686/RtpPacket.o] Error 1
[root@proj2 rtp]#
```

Figure 17 Errors Associated with Compiling Integrated Code