A DELAY-EFFICIENT REROUTING SCHEME FOR VOICE OVER IP TRAFFIC

By

NARASINHA KAMAT

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

Dedicated to my Aai and Pappa

ACKNOWLEDGMENTS

I wish to express my most sincere appreciation to my thesis advisor, Dr. Jonathan Liu, for guiding me through this thesis. He gave me the freedom to explore several topics and select the topic that I enjoyed working on the most. Without his advice and guidance this thesis would not have been possible. I would also like to extend my gratitude towards Dr. Michael Frank and Dr. Douglas Dankel for agreeing to be on my thesis committee.

I take this opportunity to thank Ju Wang of the Distributed Multimedia Group for helping me during my work and during the writing process. I would also like to thank other members of the research group for their invaluable suggestions.

I would like to thank my parents, who have always stood behind me and encouraged me to pursue higher studies. Their love and constant support have been instrumental in my success as a graduate student. They have endured the struggle of my graduate education as much as I have. I also wish to thank my sister, Nivedita, and my grandmother for their constant words of encouragement and their belief in my abilities.

Last, but not the least, I want to thank my friends Nandhini, Seema, Nikhil, Akhil, and Kaumudi for all their love and support they have given me for these past two and a half years. Their friendship has played a big part in my life here in Gainesville. They have made me feel a sense of a home away from home.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A DELAY-EFFICIENT REROUTING SCHEME FOR VOICE OVER IP TRAFFIC

By

Narasinha Kamat

December 2002

Chair: Jonathan C. L. Liu
Major Department: Computer and Information Science and Engineering

Routing the packet flows through the network is a very important function and if done efficiently, it can contribute appreciably towards ensuring the desired levels of Voice over IP (VoIP) quality. Current implementations do not attempt to reconfigure the path of the flows in the case of loss of quality. There is no guarantee that the real-time traffic will maintain a single path throughout the transmission. Based on source routing, we propose a dynamic rerouting scheme called the BPAR scheme for voice traffic, in which the packet voice flows are dynamically reconfigured to be sent along different paths, if the default or the currently used path is discovered to be unsatisfactory. The proposed scheme consists of three steps: (1) a probing technique to measure conditions on the network, (2) an admission control and routing technique to send out the data on the most desirable route, and (3) a rerouting scheme to change the routes taken by the voice packets in case the chosen route becomes congested.

Rerouting decisions are made after an initial probing of the chosen path and a periodic probing of the alternate paths when the chosen path is found to be congested. We

have analyzed the potential gains of the source routed rerouting scheme called the BPAR scheme and performed a series of emulations to measure the performance. This scheme ensures that the path taken by all the packets in a flow will remain the same for the duration of the connection until the source decides to switch it, after which all the packets follow that new path. For GSM-compressed traffic, the dynamic reconfiguration proposed in this study shows an improvement in delay performance by about 12% and a better connection acceptance of 16% over traditionally used methods under conditions of network congestion.

CHAPTER 1
INTRODUCTION

The Internet is no longer restricted to transferring pure data traffic alone. With the ever increasing bandwidth and the improvement in related technologies, the transfer of real--time data such as voice and video over the Internet has increased manifold over the past few years. In the recent past, more research has been focused on techniques that would allow co--existence of the real--time applications along with the traditional form of data. These include VoIP, streaming audio/video, and video conferencing based on Internet technologies.

VoIP (Voice over IP--voice delivered using datagram packets) and Internet Telephony have become major research areas over the last decade. The low cost to the end--users has been a major incentive for a boom in the amount of efforts put in to develop suitable techniques. The other important factor seems to be the fact that, with better compression techniques and more available bandwidth, a larger amount of real--time traffic can be transferred over existing networks. The telephone networks used all these years have been using 64Kbps data rate for making the telephone calls. VoIP can ensure through a variety of compression and adapting algorithms that calls can be made at very low rates (e.g., 20 Kbps).

However, an important consideration for any VoIP networks is that the voice data has to share the network with the normal data, which might affect the performance of the voice traffic. Realtime data has a much higher requirement in terms of delays, losses, jitter, and overall quality if the data at the receiving end. Therefore it is critical for such

schemes to maintain a minimum QoS (e.g., an least average bandwidth per connection), otherwise the whole idea of sending voice over the IP network does not sound too inviting. The problems have been compounded by the fact that the real--time traffic has severe restrictions in term of packet delays and loss. For voice communication, a 200 ms delay will result in a perceivable jitter. The challenging problem presented to the research community is: *how to support the delay sensitive real--time applications with networks providing only best effort service*?

We believe that the long--term benefits of VoIP include support for multimedia and multiservice applications. As more and more improvements are made, the use of this technology should become widespread in areas like video conferencing, distance learning, electronic courses, and broadcasting of news and entertainment programming, as also in replacing traditional forms of telephony. Although VoIP seems to be most attractive, the technology has not been developed to the point where it can replace the services and quality provided by the PSTN (Public Switched Telephone Network). To compete with today's PSTN, there must be significantly lower total cost of operation coupled with a perceived equivalent quality of service. These savings should initially be seen in the area of long distance calls. It provides a competitive threat to current telephony services.

The transmission of voice over a network requires a steady stream of 64 Kbps for it to be decipherable at the receiving end. However this is a very high requirement of bandwidth if the total number of such voice flows on the networks are considered. Hence there have been several attempts to compress the voice data before transmission. This has resulted in an increase in the amount of voice traffic that can be accommodated on the

same links. In this thesis, we carried out an experimental study for our scheme with baseline traffic at a rate of 64 Kbs and also a compressed traffic rate of 24Kbs. Moreover, the human voice patterns need to be considered while carrying out the emulations. Hence we consider what is called the talkspurt type of traffic. This is primarily an on--off kind of traffic trace that closely resembles the way actual voice traffic would be sent out in the form of packets using IP.

A very important issue in dealing with any kind of data transfer over the Internet is that of network routing. Routing over the Internet is a very complex issue with hundreds and thousands of hosts. However existing Internet routing algorithms are designed to deal with non--realtime data, where all traffic is treated equally. Furthermore, the routing algorithms give no guarantee that delivery of individual packets of the same connection will go through the same routes. As a result, packets of the same connection might end up passing through different routes, which could result in large deviation of packet delay. In summary, the network conditions of the Internet are likely to be changing continuously. The delay, loss, and jitter parameters depend on the amount of congestion on the links, which varies continuously. For example, experiment results in Wong et al. [17] that an average of 34.4 Kbps stream could be achieved late at night with low background traffic, whereas in the daytime, rates obtained were as low as 19.8 Kbps. Also there is a chance that some of the links might be rendered unusable due to development of faults. As a result, the route's ability to support the real--time traffic may change over a period of time. A static route for the entire duration of data transfer is likely to lead to a loss of quality. Moreover current routing protocols do not guarantee

that the realtime data that is being transferred will follow fixed routes. This can lead to loss of performance guarantees.

This thesis proposes a source--routed architecture for routing the real--time packets, such that upon finding unsuitable network conditions, the route is dynamically switched to a better route, if one is available. Such a technique ensures that delays encountered by the packets are minimized and can optimize the amount of traffic that can be carried by a particular route. The important components of this scheme are (1) probing different routes to find a suitable route, (2) source routing the real--time traffic to the destination along a chosen route, and (3) switching to a different route if the existing route fails to provide a desired level of performance. This scheme has been analyzed for performance using emulations on end--to--end unicast connections. An important factor in this scheme is that all the intermediate nodes are not burdened with the information of the changing nature of the routes. Important design issues were the rate of probing, use of additional buffers and the correct time to switch the route from the existing one to a better route. The thesis explains why the static and dynamic routing techniques currently used for non--realtime data cannot be extended when it comes to sharing the route with real--time data and suggests an alternate scheme for that.

The probing module is responsible for sending out periodic packets called as “probe” packets or simply “probes” on the chosen links to measure network conditions. It also has to make decisions after analyzing the received data from the probes and decide whether to reduce the probing rate and when to increase it again. The source decides that the route has to be changed when the delay results degrade by about 40 ms from the baseline delay results for a relatively prolonged probing period of about 10 seconds.

When the source decides that the current route is no longer able to support the real--time traffic that is being sent through it, it makes the decision of using the alternate path (from among the other routes, if available) to transmit the data. Before using that path, it probes it as well and then makes the decision regarding when to change the route. While changing the route, it has to take into consideration issues like buffering and route setup along the other path.

An important consideration in any routing technique should be whether any alternate paths should be considered when routing packets through a particular path. In this scheme, the alternate paths are computed and when the currently used "best" path becomes too congested, some of the packets are sent through the other route, which is the next best available one. The static and alternate routes are computed using relatively static topology information and the dynamic changes in the network conditions are used to decide the time when the packets have to be sent along a different path. It reduces the overhead of actually distributing the network information among all the nodes.

Probing rates were analyzed to decide a good heuristic for setting an optimal probing rate. The results were compared with a scheme that uses shortest routes and with a scheme that uses alternate routes only if the best routes are all blocked. An analysis of the simple, alternate routing and the alternate routing with rerouting schemes with baseline traffic showed an improvement in the average number of connections that were admitted was higher by about 13%. Similar results were observed when the scheme was tested with talkspurt traffic using three different talkspurt traces and emulated compressed voice traffic. With talkspurt traffic, average number of connections admitted is 12 % higher when using the alternate routing with rerouting scheme. Compressed

traffic results followed the same trend. Delays were lower by nearly 10 % with compressed data traffic, while the number of connections admitted was up by 16%. The alternate rerouting scheme also has a need of additional buffer space. It requires an additional buffer space of nearly 700 Kbytes for up to 15 simultaneous flows. Such an additional memory requirement is not observed in the other schemes.

The major contribution of this research has been a scheme that would allow a larger number of connections and optimize the delay performance of the connections. The most important part of the proposed scheme is that existing routing and admission control scheme do not reallocate another route to a connection once it is setup. Significantly, the results show that it is feasible to implement such a delay--efficient rerouting scheme for VoIP and get results in terms of delay performance and connection admission that are better than some of the existing schemes used.

This thesis is organized as follows. We discuss related research topics and also give a background of some of the important topics that have been incorporated in this work in Chapter 2. Then, in Chapter 3, we discuss the system architecture and design and discuss the various components of the system. We also explain how the actual system works as a whole unit. Chapter 4 gives the details of the emulation model used to carry out the different experiments. It details how the traffic patterns were simulated, how the network was setup, and how the readings were taken. Chapter 5 gives a detailed performance analysis of the proposed scheme. It provides graphs and figures to justify the various proposals that we make in this thesis and also compares the scheme to a few other schemes. Finally, Chapter 6 summarizes the entire work that has been done. It also identifies the areas of work that are worthy of investigation in the future.

CHAPTER 2
BACKGROUND AND RELATED STUDY

In the recent past, researchers have been focusing on what is now commonly called as QoS routing. Other research issues have been different architectures for admission control, reducing the packet delays, and error correction techniques to offset jitter. This thesis concentrates on the issues of delay reduction and admitting more calls to the network via dynamic rerouting.

The Internet is basically a huge network of networks that was primarily built for data transfer using the best effort model of datagram service. Until a few years ago, the use of the Internet had been restricted to email, displaying web pages, file transfer, and such other applications. These applications are not very time--dependent, and as such, have a higher tolerance for the best--effort model of the Internet. However, with the growth of the Internet in recent times, its use has expanded to include real--time services such as streaming audio and video, web--conferencing, Internet Telephony, etc. The real--time services are very time--dependent and delays and packet losses hamper the operation of a real--time session beyond repair. The important factor when considering real--time services is the Quality of Service [12, 13] that the users desire. Even a few seconds of delay can render an Internet voice conversation useless for the users. The quality of these applications is largely dependent on the bandwidth of the links to which the users are connected. The efficiency of packet voice with deterministic delays is discussed by Baldi et al. [1]. Depending upon the available capacity, the QoS ranges from very

good to extremely poor. The crux of the problem is that the current Internet structure does not have a dedicated channel for serving the real--time data. The delay--intolerant real--time packets get treated in the same way as the normal data packets in many of the links over the Internet, which results in the degradation of the quality of service.

It is in the background of these constraints that research on the transfer of real--time data has gained a lot of attentions in recent times. There have been a lot of schemes that have detailed the use of resource reservation protocol (RSVP) [2] as a solution towards ensuring the QoS. This protocol entails the intermediate routers to reserve the resources for the session of real--time data transfer. However, a rejection from one of the routers could result in the rejection of the whole call setup. A general comparison of reservation--based protocols could be found in Braden et al. [2].

Another method is to use alternate path routing. The original ARPANET used distributed adaptive routing algorithms based on measurements of queuing delays at each link. These measurements were propagated to all the routers and packets were forwarded along the calculated paths to generate the minimum amount of delay. Popular algorithms used were the Distance Vector algorithm and the Link State Updates algorithm, which replaced the distance vector algorithm. As the ARPANET evolved into the Internet, the need for autonomous control emerged. Different agencies had their own backbones and internal routing mechanisms, and they wished to manage their internal routing differently. This led to a two--level routing hierarchy that exists even today. Load based routing algorithms were discarded in favor of more stable routing protocols like Open Shortest Path First (OSPF), Border Gateway

Protocol (BGP), and Routing Information Protocol (RIP). These techniques adapt more easily to changes in network topology.

2.1 Rerouting Concepts

The idea of dynamic rerouting has been tried in circuit switched networks [16] to increase throughput by transferring calls to alternate paths when the directly chosen path is blocked or overloaded. In this method, a routing decision has to be made at call--arrival time based on the network information available at that time. The basic idea of this scheme is to redistribute the network load so as to avoid the traffic bottlenecks. The calls are routed on to alternate paths and rerouted back to their original direct path according to the prevailing conditions. Rerouting in circuit switched networks is the practice whereby the calls on the alternate paths can be routed back to the original path or to some other less congested alternate path as the situation warrants. Results [16, 17] have shown that rerouting can provide a significant throughput increase in all conditions. Rerouting has been shown to be an effective means of maintaining the stability of the network under dynamic routing.

However, our focus has been a different scheme in which a new incoming call is directed through the optimal direct path to its destination, if that route can provide the desired levels of QoS. The situation for virtual circuits using IP datagrams transmitted over the Internet is vastly different from the one experienced in the circuit switched networks. Bandwidth is no longer the only consideration for packet--switched networks. There are a lot of other factors such as delays, loss, and congestion that have to be considered. The proposed scheme has a probing module that will ascertain the network conditions at the start and later periodically to get a good heuristic of the best path for the connections to go through. It has to be ensured

that all the packets will go through the chosen route. In case a chosen route starts performing poorly, a new route is setup along another route, if available. The whole process of ascertaining that a particular path is no longer providing the desired quality levels, finding an alternate path and switching it onto that path after setting up a new connection, without the user's knowledge is called as dynamic rerouting or what we call the BPAR scheme.

2.2 Admission Control

Admission Control is a concept that applies mainly to voice traffic, not to data traffic. If an influx of data traffic oversubscribes a particular link in the network, queuing, buffering, and packet drop decisions are taken to resolve the congestion. The extra traffic is simply delayed until the interface becomes available to send the traffic, or, if traffic is dropped, the protocol or the end user initiates a timeout and requests a retransmission of the information. Under normal circumstances, networks resort to dropping packets to resolve congestion based problems. However, the quality of voice traffic would take a beating if that were done. The best way out in case there are congestion or network delays is to start refusing calls. There has to be some way of deciding that if the number of calls in the network exceeds a certain value, then the quality of all the calls is guaranteed to degrade.

Admission control algorithms are used to make sure that the quality of service provided by VoIP remains in acceptable quality of service levels. There is a wide genre of admission control algorithms that are being researched. Among the most widely used admission control techniques are resource allocation based algorithms and measurement--based algorithms. Resource reservation algorithms have been shown to have a scalability problem [9,10]. Also they require the applications to give

an accurate description of the flows, which is somewhat of a limitation. Measurement based approaches [3, 6, 11] though better than the older traditional approach, need to adopt per hop measurements and have caused problems resulting from the measurements.

A new method suggested by [4, 8] is endpoint admission control, based on end--to--end probing. Specifically the host first probes the network by sending probe packets at a specific rate and records the results in the form of delay, packet loss, and congestion statistics. It admits the flow only if the loss percentage is below a certain threshold. A benefit of this scheme is that it can work even when the end--point by itself is not aware of the total resource capacity of the network. The scheme proposed in Elek et al. [8] has a test and probing module before the packets are actually sent out. To ensure that the probe packets themselves do not disturb the quality of the network traffic themselves, they are not sent out continuously, but in periodic intervals and avoiding burstiness. Results have proved that such a scheme may be more sensitive and accurate to respond to the dynamic fluctuation of the network conditions. We believe that this line of research is one of the optimal ways to implement admission control and rerouting and we use such a end point based probing scheme in our proposed BPAR scheme.

2.3 Source Routing

Internet routing is computed in a distributed manner, where each routing node making a decision to forward a packet to the next node or to the destination. Since it is important to ensure quality for real--time services, it would be of utmost importance for the routing strategies to ensure a single quality path for the packets to go through. This would ensure a constant level of quality to the users. Unfortunately

with the current setup, it is not possible to guarantee that. Source routing is an alternate to hop--by--hop routing, in which the source itself determines the entire route and the intermediate nodes forward the packets to the next hop specified in the source route.

A primary advantage of the source routing scheme is to reduce the complexity of the routers and the gateways. The header size and with it, the packet size do get considerably larger because of the increased amount of routing information that the packet carries. However, unlike the normal non--realtime data that is sent out on the Internet, we are sending a large amount of voice data at very high speeds. Therefore, the added cost of carrying the address information is considerably reduced. In a source--routed setup, it makes more sense for an initial path setup packet that sets up forwarding information in all the intermediate nodes. After that, the subsequent packets only have to carry an identifying variable to indicate that they have to be transferred on the pre--decided route.

Another advantage is that the implementation of intermediate routers becomes easier. Also the ability of a source to select a route that best suits its interests is increased in such a scenario. A disadvantage proposed about source routing is slower convergence to topology changes. But major topology changes do not happen very frequently, given that people are very less likely to keep changing their network setup unless there is some real problem. However hop--by--hop routing may use sub--optimal routes during the period of topology changes as well.

However, source routing does allow fast adaptation to changes in network condition because the source node can quickly and correctly detect when the

performance deteriorates and thus enables it to switch to better alternate routes. Similarly, source routing has been used in bridged Local Area Networks. It has similar advantages in this environment as specified for the others. Considering IP's best-effort packet delivery protocol, source routing is perhaps the most practical method for packets to follow the same route for a majority of the session durations.

It is also true that source routing provides a flexible and an easy--to--implement technique without requiring global consistency across network nodes. It eases the computational burden on the intermediate nodes, and if the initial route setup is carried out, it reduces the overhead involved. Source routing also enables us to explore the use of alternate paths when the currently used path is no longer able to provide the desired level of QoS. It is for these reasons that we decide to include source routing as a way to implement dynamic rerouting for voice packets over the Internet.

Using the concepts of admission control, source routing and rerouting, we proceed to demonstrate the proposed adaptive rerouting scheme for voice traffic sent on the Internet. The next 4 chapters of this thesis give the complete design and implementation details as well as the performance results of the new scheme.

CHAPTER 3
SYSTEM DESIGN

The quality of voice through IP network is decided by the network congestion situation of the path that the voice packets will travel through. A major design issue in voice over IP is thus, how to choose the best path. Using the source routing technique in IPv4 and the capability of flow-based routing in IPv6 network, the end system is now able to discover different routing paths and force the traffic through a designated route.

Figure 3-1 describes the system level architecture of our source--routed voice over IP scheme. The whole system consists of three function parties: the source node, the intermediate network, and the destination node. The source represents the call initiator, and the destination node represents the call receiver. For a given source--destination pair, multiple paths with different link quality could exist to provide voice service. The control protocol in the source node contains several modules. The probing module is used to probe the given routes to gather the network performance along that route. The admission control module implements the admission control policy that is also based on the system load and capacity. We adopt a simplistic abstraction of the admission control module where the capacity on each of the links is presumed in advance. Once the information of the routes is made available, there are several ways to decide the traffic routes.

3.1 Routing Strategies

In its simplest form, the source node selects one route that is deemed to be the best route for the transmission from the source to the destination.It does not consider any alternate routes if the current route seems to be congested and is not performing

satisfactorily. No matter how many future traffic arrive, the source node will always use the same route.

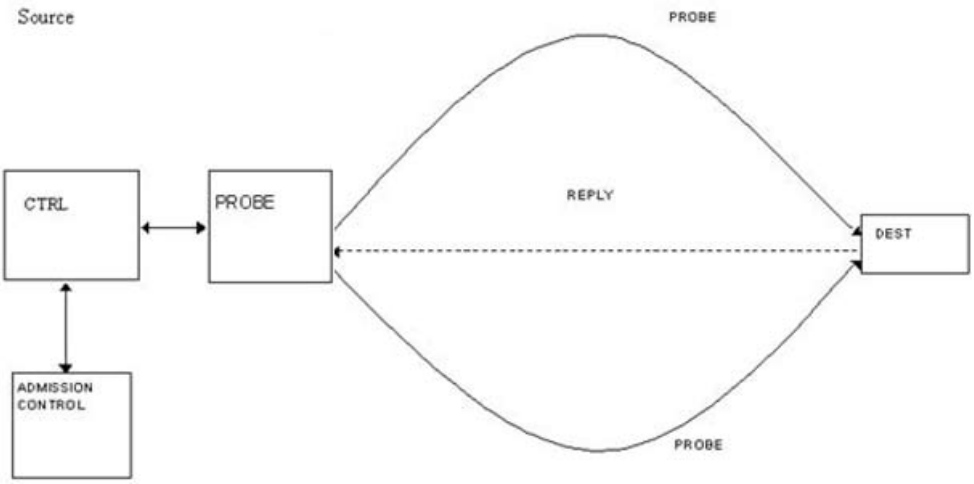

Figure 3-1 Two modules of route selection: probing and admission Control

We name this strategy as **SR**--Simple (shortest) path scheme. The Simple Path Scheme will not perform very well in case of network updating or overloaded traffic. For example, if that particular route is unavailable or unable to provide satisfactory service due to hardware/software failure, then all existing connections will be blocked.

The second approach is called **AR** -- Best path strategy with alternate routes. In this strategy, the source node selects one route that is deemed to be the best route for the transmission from the source to the destination. However if the default shortest route is not able to provide the required levels of service, the strategy will consider the alternate shortest routes if they are available. If there are no shortest routes but some other alternate routes, then this strategy will pick that alternate route. The chances of the flow being denied of service completely, gets a bit reduced in this strategy, as there are a few alternate paths to choose from.

The first two strategies can only partially deal with the changes in the network; they are essentially static methods in the sense that the path will not change once the

connection is established. Our third scheme considers both alternate routes and dynamic rerouting. It will try to select the best possible route initially. But if that is not possible, this strategy will select one of the alternate routes. The strategy also makes dynamic changes to the routes during the transmission if at that time it detects a loss in performance along that route. To accomplish the dynamic rerouting, the source node needs to probe the network constantly to detect the network change promptly. In the following discussion, we focus on the different aspects of the third scheme. For the sake of convenience, we use a short name for the scheme -- **BPAR** (Best Path with Alternate Rerouting).

Figure 2 shows the control logic of the BPAR scheme for the source node. The protocol will be executed each time it received a request for new call setup. This is followed by routes discovery and initial probing. The source node is usually provided with the knowledge of the routes it can probe in prior. However, in case the network topology information is not available, the source node itself has to discover the routes using the router discovery algorithm (such as OSPF). Probing is done by transmitting probe packets on a particular route. The length of the probe packets is chosen to match the size of the voice data packets that are to be sent later. Once all the routes are probed, the source node computes the measurement statistics in terms of delay and loss. After checking all the three routes, the decision is made to send the voice flow along the best route. After the decision of route selection, a virtual connection is setup along that path to allow quick switching for the rest of call time.

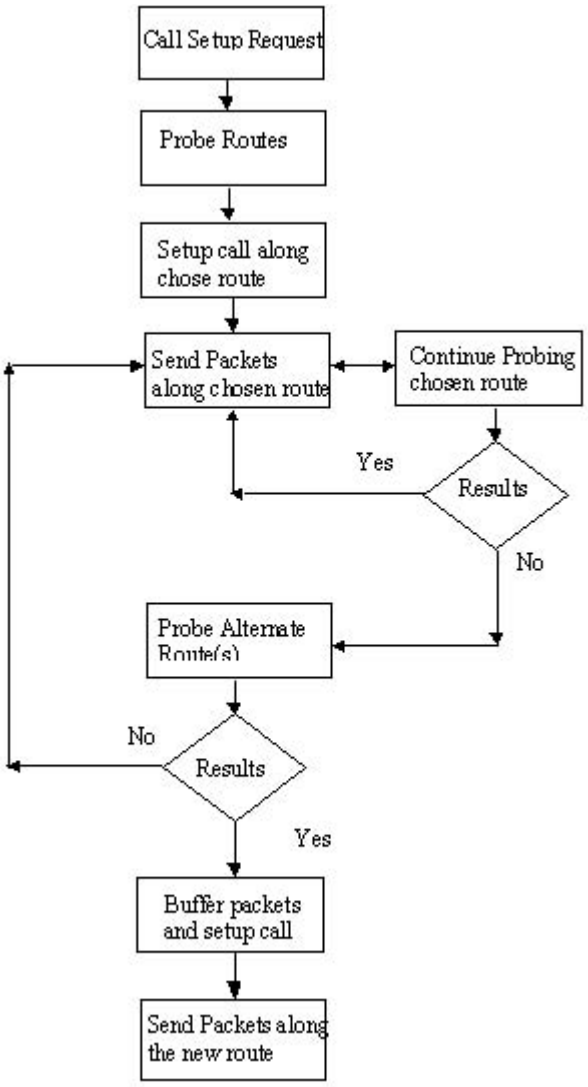

Figure 3-2 Control logic of BPAR scheme.

Meanwhile, probing is continued to make sure that the change in network condition can be quickly discovered. The result of each probing is analyzed, if the network is running in good conditions, the voice traffic will remain on the current path, and the protocol will loop back to execute the next round of probing and checking.

If the checking of the probing results indicates that the current path cannot provide the desired levels of quality, the alternate paths are probed immediately. Depending upon the results of that probe, a decision is made as to whether to switch the route to the newly probed route. In case the route has to be switched, further packets need

to be buffered in addition to the pre--buffering that the source always carries out before sending. This additional buffer is for the packets that the source keeps transmitting while the route is being switched. The new route connection is established along the chosen path. The old route is kept in the candidate route pool.

To evaluate the performance of the BPAR scheme, we need to investigate the effect of several design factors. First of all, how should the probing be carried out? Probing should be performed with enough frequency to have a quick system response. Meanwhile, over--probing will become a major overhead imposed to the network, which could potentially decrease system capacity. Secondly, we need to decide how the route should be selected in case of network congestion. Different criteria should be considered. The rest of discussion of this chapter addresses these design factors in greater detail.

3.2 Probing Module

The basic premise of the scheme is the assumption that multiple paths exist between the source and the destination. Prior to setting up a connection along any one path, there is a need to probe the path and decide whether it has enough resources for the transfer to be initiated. This probing mechanism is implemented at the source node.

The probe packets are sent from the source to the destination. To ensure that the packet travel through a desired path, we use strict source routing option to specify the intermediate routers. The destination will reply with an immediate ACK packet along the same path such that the delay can be accurately estimated. The source node then calculates the network statistics such as delay and packet losses for future use. When probing is finished for all candidate paths, the protocol will choose the one with best quality.

Once the route has been selected, all the following voice data will be delivered through it unless there is a change in the network condition. Two types of events will trigger the changing of voice path. First, the change of network topology can make existing route unusable. The network changes are relatively rare. As a result the added cost of discovering the routes in such a case does not affect the overall efficiency of the system too much because the route rediscovery itself is not done frequently.

Network load changing causes the other event. Unlike circuit switching, it is not sufficient to select one particular route and hope that it would continue to provide the desired levels of service for the entire duration of the transfer. It is equally important to keep up the probing on the selected route so that when the desired network measurements are obtained, a decision can be made about the continuing of the transfer along that route. Keeping in mind that probing is an additional burden on the computational power of the source; it may not be advisable to keep probing all the alternate paths as well. But it is certainly feasible to do that along the one selected route, as results will show later.

When the system is heavily loaded, there will be a frequent change in the network congestion. This could result in frequent invocation of the probing process in the source node. Aggressive probing under such scenario could worsen the network congestion. Thus it is important to design the probing algorithm with least overhead. We use an adaptive probing in our proposed scheme: Initially the source node sends 4 packets per seconds. If after a duration of 20 probes the results continue to remain satisfactory, the probing rate is reduced to 2 packets per second, then to 1 and so on. To reduce the route setup delay, the source probes up to 3 paths initially.

However the intensity is reduced to 2 probes per second for 20 probes. After that if the probe results are still favorable, the period between the probes will be further reduced. Notice that the alternate paths are not probed at this time.

3.3 Admission Control and Route Selection

Once the statistics of each candidate path are obtained after the probing process, the source needs to choose one as the traffic channel. That is a decision that is made by the underlying admission control protocol. The routing selection algorithm executed in the admission control module has to ensure that the selected route has a minimal chance of being rejected at the present time or at a later stage. For example, consider that there are two qualified routes with three links each and same overall mean delay of 80 ms. Furthermore it is observed that the first route has a STD delay of 30 ms and the second route has a STD delay of 40ms. The delay of both routes can exceed the 100 ms delay requirement in case of network variation caused by random factors. However, to maintain a smooth voice quality, our protocol should select the first route. The reason behind such a rationale is that the quality of that first route in term of probability of outage is less than route two. That does not mean that the first route is a perfect route. It is still subject to changes in network conditions and loads. But under the given conditions, it is better to select the first route than the second.

The delay is not the only consideration of the admission control. The route should be decided based on a more complete statistic set, including the number of hops of the route, the mean delay time, STD delay time, and the percentage of packet loss. These parameters usually did not agree with each other, e.g., the shortest path (in term of hops) is not necessarily the path with shortest mean delay. We developed weighted admission criteria to select a best path over the network parameter mentioned above. The mean

delay is put into the highest priority, with the highest weight. The second priority is given to the percentage of packet loss. The packet loss is an important consideration in real networks, where loss could prove to significantly affect the performance. In our emulated network, the loss percentages for each type of scheme used are very small and comparable. Thus the effects of the loss are found to be similar across all the schemes. The third consideration is for the STD delay, and the number of hops is the last one. The score for each route can thus be calculated by the following expression:

$$S= W_{md} * D_m + W_{pl} * P_{pl} + W_{sd} * D_{std} + W_h * H.$$

Here D_m and D_{std} are mean delay and standard deviation in delay in the unit of millisecond. P_{pl} is the packet loss rate, multiplied by 1000 to match the scale of delay. H is the number of hops of the given route.

The following algorithm can describe the overall admission control protocol:

1. Gather the statistic result for all routes from the probing module.
2. Select all routes such that the mean delay is less than the pre--defined threshold (a typical value is 100 ms).
3. Delete the routes already reached their capacity limit
4. Calculate the score for each of the pre--selected routes.
5. The routes with the smallest score will be used as the traffic route.
6. Update the number of active connection for the chosen route.
7. Establish the connection through a signaling process.

The choosing of the weights for the different network statistics can result in different admission schemes, which is beyond the scope of this study. In our study, we fixed these parameters such that W_{md} =1, W_{pl} =0.5, W_{sd} = 0.2, and W_h =0.1.

This approach to route selection differs from traditional approaches that always go for the route with smallest hop number. In our scheme, the decision is made entirely

after ascertaining the network load through the probing mechanism. If there is more than one good path, the path with the least load is likely to be chosen.

3.4 Dynamic Rerouting

Initially when the source node decides which route to take to send the packets to the destination, it sets up the connection with the destination along that path. All the packets in that session have to follow the same path. When more calls are assigned to the path, the traffic on that link starts increasing and network become congested. Further calls could be scheduled into alternate routes according to our route selection criteria in the admission module.

However, the network condition of a route could become congested even if no new calls are admitted on the current route, due to factors such as increases in background traffic, or simply due to software/hardware problems for the routers along the path. One way to offset the situation for the source node is to offload non--realtime traffic from it. However, it is impossible for the source to control other traffic on the link that is not originated from it. This makes it necessary that the existing calls be switched to another route with enough quality. The advantage of this approach is that the source has a complete control over the route that is chosen for the packets. Hence it can easily ensure that the following packets now take the other route that it selects for them.

To detect the changing of network conditions, the source node needs to probe the network on a regular basis. The source node continues to probe the link to the destination after the connection has been setup to gather the delay and loss statistics. These statistics give it a good idea of the amount of load on the probed links. If it decides that the current route can no longer provide the desired quality, continuing on this route might lead to the loss of quality and an increase in the load on that link. Once the decision of switching

route is made, we need to select a new route. The source has the freedom to choose from the alternate routes available to it. However, before starting transmission on any of the alternate routes, the source needs to consider the possibility that the alternate route, which has not been probed since route establishment, will have experienced changes in the network conditions. Thus we perform a full scale probing for all the alternate routes to be able to select the target route for the voice connection of interest.

If the probe result along the alternate path does not guarantee better conditions than the ones prevalent on the currently used route, then the currently used route is not discarded. The admission control module would ensure that no more calls are allowed to go through that link that would further deteriorate the network conditions. If the probing result shows that the alternate routes can provide guaranteed service, we can simply re--execute the admission module as discussed above.

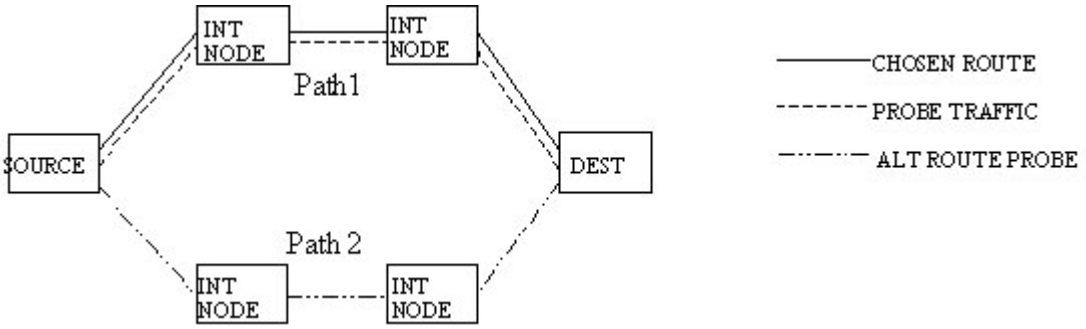

Figure 3-3 Scenario after initial probing and route setup.

Figure 3-3 illustrates an example of the rerouting scenario. After the initial probing is done, path 1 is setup as the traffic route, and path 2 is the alternate route as a backup. As described above, path 1 is continually probed afterward. Assume that the route is determined to be unsatisfactory when the probe results start showing a mean delay of above 100 ms over a probing period of about 20 probe packets. Now assume that the mean delay at path 1 become 120 ms after a certain amount of time, the rerouting

procedure will be invoked. Further assume that path 2 can provide a mean delay of 80 ms. It will be set as traffic route. Figure 3-4 shows the situation after the call is rerouted on the alternate path. In this case, we no longer probe the original path that was chosen.

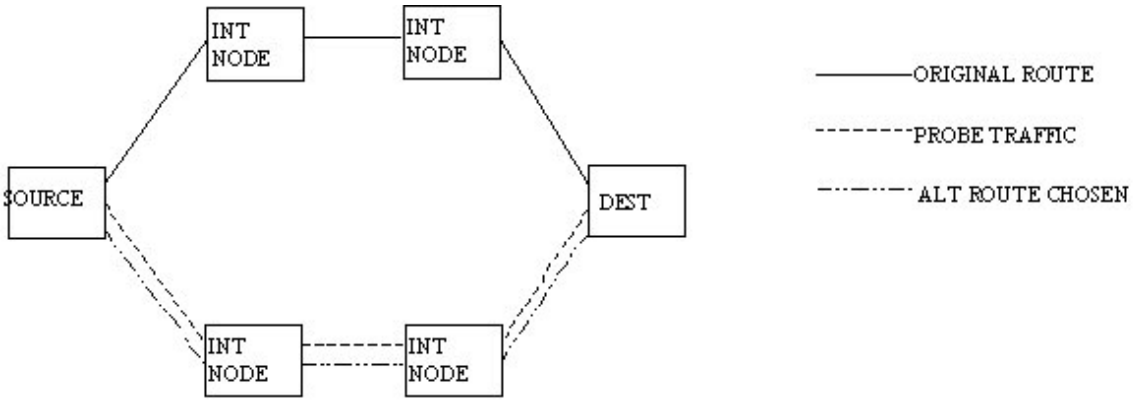

Figure 3-4 Scenario after alternate route probing and route setup.

One design factor is how frequently the existing route should be probed to detect the network congestion. Too less probing will be unable to detect the network congestion early, thus could decrease the quality of voice traffic in the congested route. Too much probing will cause additional traffic to the route, which could be significant at the time of network congestion. One way would be to reduce the amount of probing under slightly stable network conditions. If the probe results are relatively steady for some amount of time, then the time interval between successive probes can be relaxed. Our scheme would increase the duration between the probes exponentially when the probing results remain steady. On the other hand, if the probes discover a considerable changing of the route quality, the duration between the probes can be decreased in the same way that it was increased. So when the probes are consistently returning favorable results, the frequency of probing is decreased by half and then half of that and so on after each interval of 20 probes. If the probes continue to show a favorable pattern, the duration between the probes is doubled for the next 20 probes and so on.

The figure 3-5 shows the delays obtained with different probing strategies. The probing traffic 1 uses a constant probing rate of 8 probes a second and no change in the probing frequency. Probing traffic 2 uses a probing rate of 4 probes a second and no changes in the frequency either. Probing traffic 3 has an initial frequency of 4 probes/sec, and it reduced exponentially after every 20 probes, if the probe result is stable. As can be observed, the high probing frequency result in a high traffic delay in general. When the number of concurrent voice connection is low, the delay difference is within 3 ms range. However, when more connection is supported, the probing strategy with 4 packet/sec (probe traffic 2) shows an even smaller delay than the probe traffic 1 (8 packets/sec). The third probe traffic has the smallest delay at any case. At 15 voice connection, our exponentially decrease probing traffic result in 5 ms than the probe traffic 1. As many voice connections are supported (e.g., 100 connections), it can be expected that the difference of delay between the probing traffic 1 and 3 will be significant. Due to limitations, we will leave this as a part of the future work.

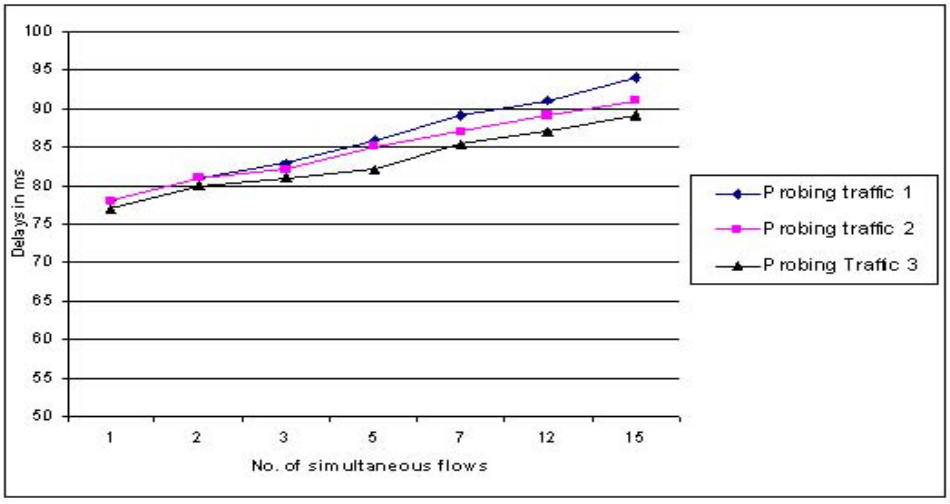

Figure 3-5 Difference in delays with different probing durations.

3.5 Rerouting time delay and Buffer Requirement

Buffering is an essential part in any streaming audio/video application. For the voice communication, the buffer requirement for each connection is not very demanding. For example, with 64 kbps raw voice data, we need 6.4 kbits to have a 100 ms buffer. However, the amount of buffer is proportional to the number of active connection, as well as the amount of buffer time. Thus the accumulated buffer space can be very significant in term of implementation. By calculating the smallest buffer requirement, we can guarantee that a large number of connections are supported with a good quality.

When a decision of route switching is made, the new connection must be established. Our scheme must provide enough buffer space for the existing connection in all cases. In the worst case, such as failure in intermediate node of the original path, the old connection is broken before the new connection can be use. Thus we need to buffer all the packets right after the decision of reroute, until the new connection can be used.

The expected amount of buffer space that will be required can be computed based on three factors; the number of active connections, the amount of time that we need to buffer the packets, and the total number of bytes that are actually sent during that time. The amount of time at least includes the delay taken to probe the alternate paths and the setup time on one alternate route. As we will see in the simulation results, this would mean a duration of approximately 70 ms for every flow. We can get an estimate of the actual buffer size that will be needed by using the formula:

$$B \text{ (in Kbytes)} = T_{RR} * N_o * N_{b.}$$

Here, B is the buffer space required, T_{RR} is the time required for rerouting, while N_o and N_b represent the number of number of connections and the number of bytes transmitted in that time.

3.6 Summary

This chapter describes the architecture and design of the proposed rerouting scheme. It details the various components of the system. The probing module is responsible for sending out periodic probes on the chosen links. It also has to make decisions after analyzing the received data from the probes and decide whether to reduce the probing rate and when to increase it again. The admission control and route selection does the job of selecting the routes by applying the algorithm and deciding whether to admit the new calls or not. When the source decides that the current route is no longer able to support the real--time traffic that is being sent through it, it makes the decision of using the alternate path (from among the other routes, if available) to transmit the data. Before using that path, it probes that as well and then makes the decision regarding when to change the route. While changing the route, it has to take into consideration issues like buffering and route setup along the other path. The emulation model is described in the next chapter followed by the experimental results.

CHAPTER 4
EMPIRICAL STUDY WITH NETWORK EMULATOR

We studied the performance of the proposed rerouting scheme using simulator developed to closely model the topology of an actual networks and traffic characteristics. The performance metrics to be measured include: (1) average delays when number of active connections vary with and without rerouting at source, (2) new call acceptance probability with and without rerouting, (3) the rerouting time and buffer requirement during the rerouting. In this chapter, we describe the basic simulator setting used to carry out the investigation.

4.1 Network Topology

Our simulator mimics a network topology consisting of network nodes that can serve as the end point and the inter--connecting routers. The nodes are connected by bi--directional links. Once the topology is generated, each of the nodes will have complete knowledge of the paths to the other nodes. Thus each node can decide the routes to use when it is acting as a sender. Two nodes are chosen as the end points, one as the source and another as destination. Source routing is used to deliver all the data in the designated path, thus intermediate nodes will not make independent decisions to route the voice packets. A simulated 6--node network is showed in Figure 6.

Our simulation parameters for network are based on an end--to--end measurement of network performances between our department and machines at the University of Toledo, Ohio. We believe that this represent a typical delay and bandwidth of the INTERNET in current situations.

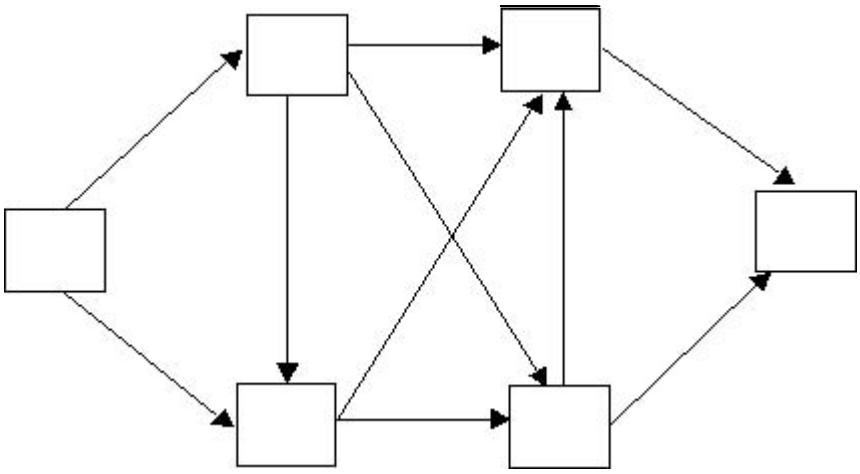

Figure 4-1 Simulated network topology.

The average delay on the links with just one connection is shown in Figure 4-2. The average round--trip delay in each of the links is approximately 80 ms, and the maximum and minimum delay is 130 ms and 75 ms respectively. Figure 4-3 provides the delay statistic with one source transmitting versus three sources transmitting at the same time. It clearly shows the behavior of the transmitting and receiving modules, which increases the average delay from about 80 ms to an average of around 84 ms. Table 1 shows the mean and standard deviation of the end--to--end delay observed. We will use these delay parameters to randomly assign link level delay for the inter--node connections.

Table 1 Mean and standard deviation of base statistics

No. Of Simultaneous Transmissions	Mean	Standard Deviation
1	80.34 ms	9.1
4	84.1 ms	10.9

Table 1 shows the mean and the standard deviation of the delays. The achievable bandwidth observed in the above link is about 1.3 Mbps. This could support less than 20 full rate voice traffic (64 kbps). In our simulation we assume that the maximum number of concurrent connection is 15 for 64 kbps voice traffic, and 30 for the compressed voice traffic.

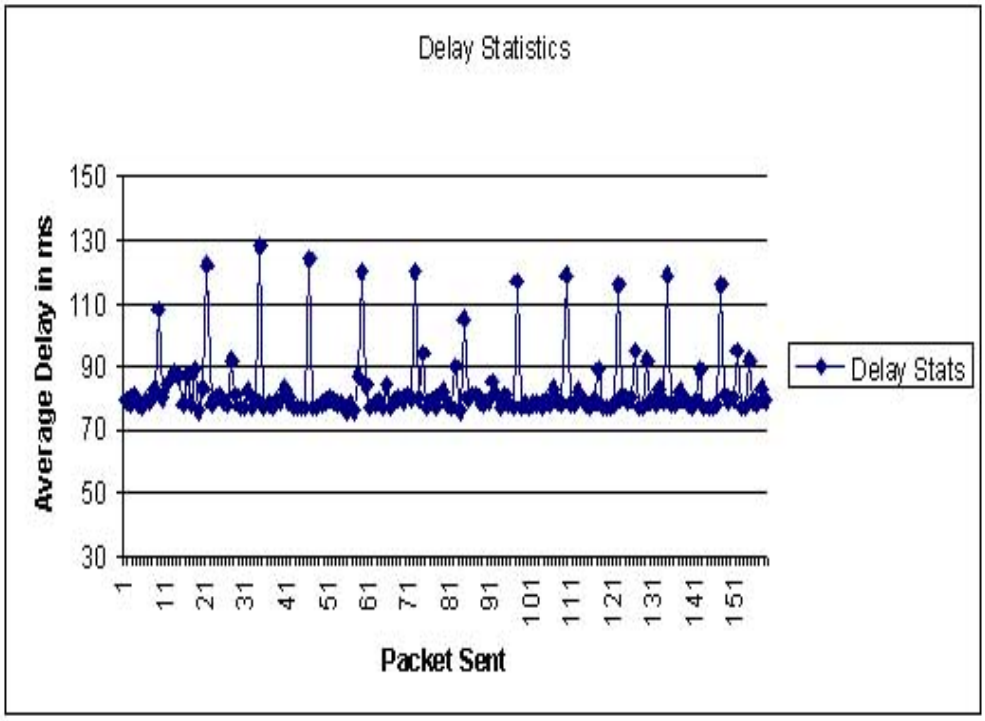

Figure 4-2 Normal delay characteristics of the simulated network--single connection

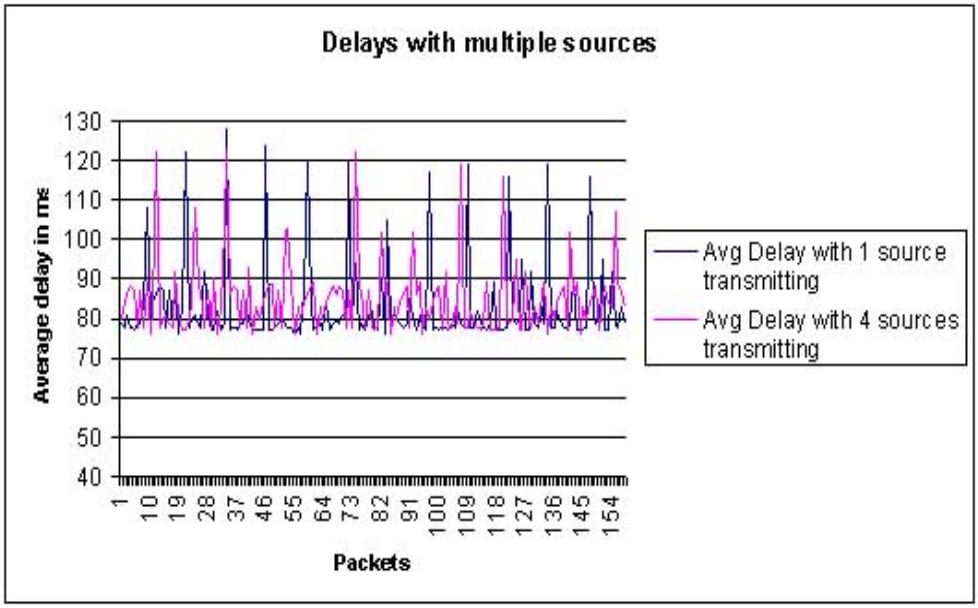

Figure 4-3 Normal delay characteristics of the simulated network--multiple connections.

The network is loaded with background traffic. Each inter--node connection is independently loaded according to Poisson distribution with mean packet rate of 160

packets/sec, and each packet is 1000 byte long. This will occupy about 10% of the total bandwidth.

To assess the performance for different kind of voice traffic, we repeat all the simulation for three types: (1) flat--rate voice without talkspurt consideration and compression, which need CBR of 64 kbps, (2) Uncompressed voice traffic with consideration of talkspurt, and (3) compressed voice data using linear prediction voice encoder, such 20 kbps GSM.

4.2 Voice Traffic Simulations

Traditionally, voice data are usually transmitted at a data rate of about 64 Kbs. This rate must be maintained to ensure the audible quality of the transmitted audio traffic. The voice data are packetized into 480 byte UDP packets. The source send voice packets every 60 milliseconds such that the outgoing traffic is 64 Kbps CBR rate ((480 bytes per datagram) * 8 bits per byte / 60 ms)) = 64 Kbs per second). Since we are using UDP, no acknowledgement is needed from the destination node. Also no retransmission is used in the source side.

Another type of voice traffic takes talkspurt into consideration. The talkspurt voice model considers the human speech patterns and how they are converted into packets that can be transmitted on the networks.

As can be seen from the figure 4-4, human speech is not a continuous form of communication, but a series of intermittent spells of voice and silence. In a true voice packet generator, the analog voice is then digitized using a sampling rate that equals the Nyquist rate. A corresponding number of packets are formed. They have a predefined length that includes all the ubiquitous headers and the data.

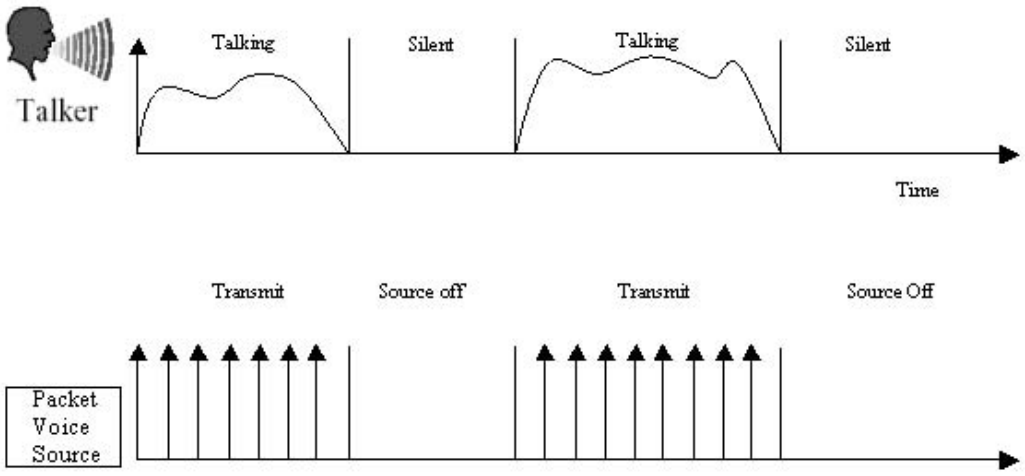

Figure 4-4 Typical human speech and corresponding voice source behavior

The periods of silence are symbolized by an "off" period, where no packets are generated. The "on" periods are usually called talkspurts. The traffic characteristics of packet voice are discussed by Deng [7]. We simulate the talkspurt traffic in such a way that it resembles the on--off nature of a normal voice conversation.

The modeling of talkspurts and the silent intervals is a separate research issue by itself. There is no well--established model that could suggest the best way to simulate all types of talkspurts and periods of silence. We adopt the on--off based model [15]. The figure below shows one of the talkspurt--silence traces generated by our simulator. We believe that this model should provide a fairly correct statistic of the average time that a human speaks and remains silent when reasonably used over a period of time to carry out the complete simulation. With the information of on/off of the voice activity, the transmitting model will check the voice state first, if the current voice state is still on, it will continually operate on the 64 Kbps CBR rate. Otherwise, the voice model will switched to off, and the source will stop transmitting voice packet. Meanwhile, it will check the status of voice model every 60 ms to see if there is a new talkspurt.

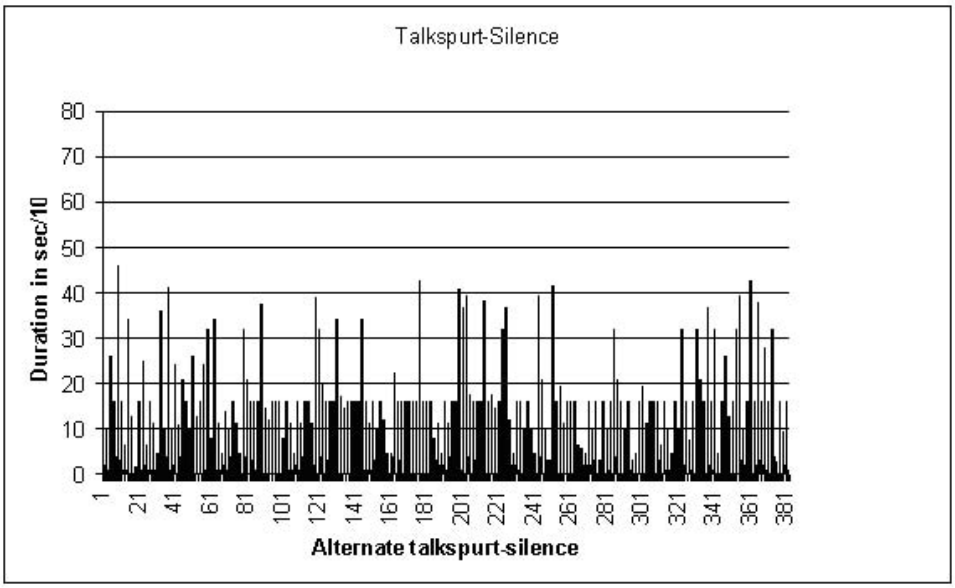

Figure 4-5 Talkspurt –silence durations. Alternate periods are talkspurts and silence.

Recent advances in compression techniques have made the need for the data rate much lower. Researches have showed that compressed voice can be transmitted at rates of 24 Kbs at real--time voice [14]. In this study, we use GSM as the example for the compressed voice data. The GSM voice is transmitted in a similar way as that for the talkspurt traffic, except that we can send the data at a lower data rate when there is voice conversation. The average data rate during the active period for GSM is 20 Kbps.

CHAPTER 5
PERFORMANCE RESULTS

We use the emulation model described in previous chapters to evaluate the performance of the rerouting scheme. We also performed these measurements to compare the results of this scheme to some alternate schemes. In this chapter, we present the different results obtained.

5.1 Average Delays

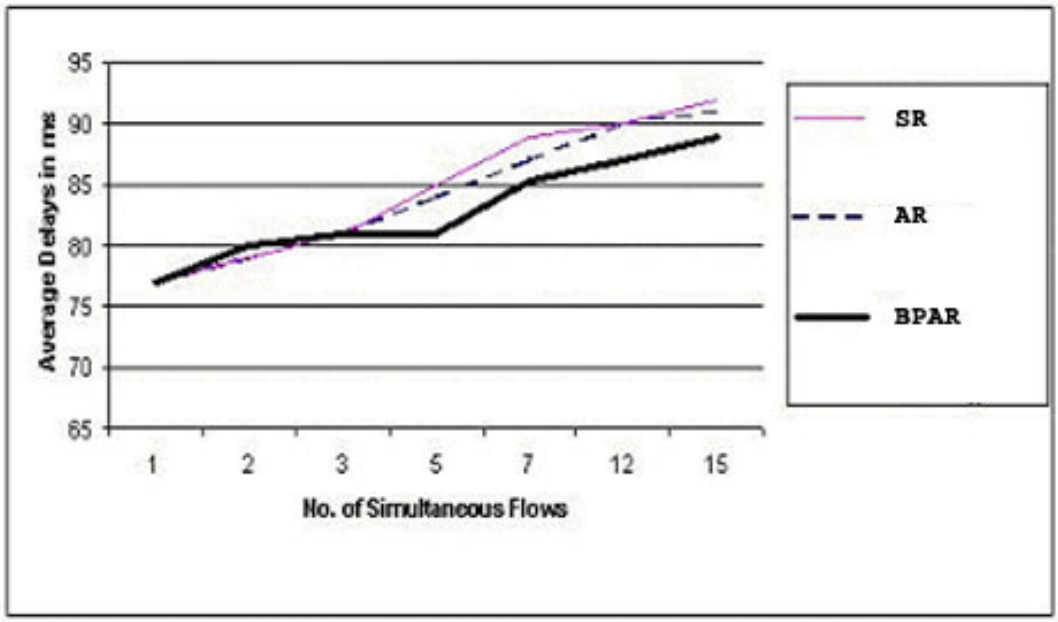

Figure 5-1 Average delays when numbers of active connections vary with and without rerouting at source with flat--rate CBR traffic.

Figure 5-1 illustrates the average delays when the number of simultaneous transmissions is steadily increased. The following observations can be made:

- In general, when the number of active flows on the links increases, the average delay will also increase. For the SR scheme, the delay is 76 ms when there is one connection. When two connections exist, the average delay become 79 ms, which is about 4 % increase. With 15 connections, we get a delay of 93 ms. A similar trend is found for the AR scheme and BPAR scheme. This indicates that the increasing amount of connections actually caused the network congestion.

- The AR scheme performed better than the SR scheme. When there are less than 4 connections, the delay is very close for both schemes. However, when there are more than 5 connections, the AR scheme starts to show a lesser delay than the SR scheme. With 5 connections, the AR scheme results in a delay of 83 ms, which is 3 ms less than the SR scheme (86 ms). When further increase the connection, a similar delay difference could be observed. With the AR scheme, some of the flows may be sent along different routes and this reduces the average delay for the packets. But, as the number of flows goes on increasing, the performance of the AR scheme is almost similar, though still better as compared to that of the SR routing scheme. This is because the routes are assigned at the start and not changed as the network conditions change.

- Our BPAR scheme results in the smallest delay among all the cases. The reduction in the delay is especially noticeable when many simultaneous connections exist. With 15 simultaneous connections, the BPAR is 87 ms, which is 6% less that of the SR scheme (92 ms) in average for every connection. It could be expected that the delay of BPAR will be even more attractive when a larger amount of connections exist. We can see such a trend with the results obtained for the compressed traffic shown later in this chapter.

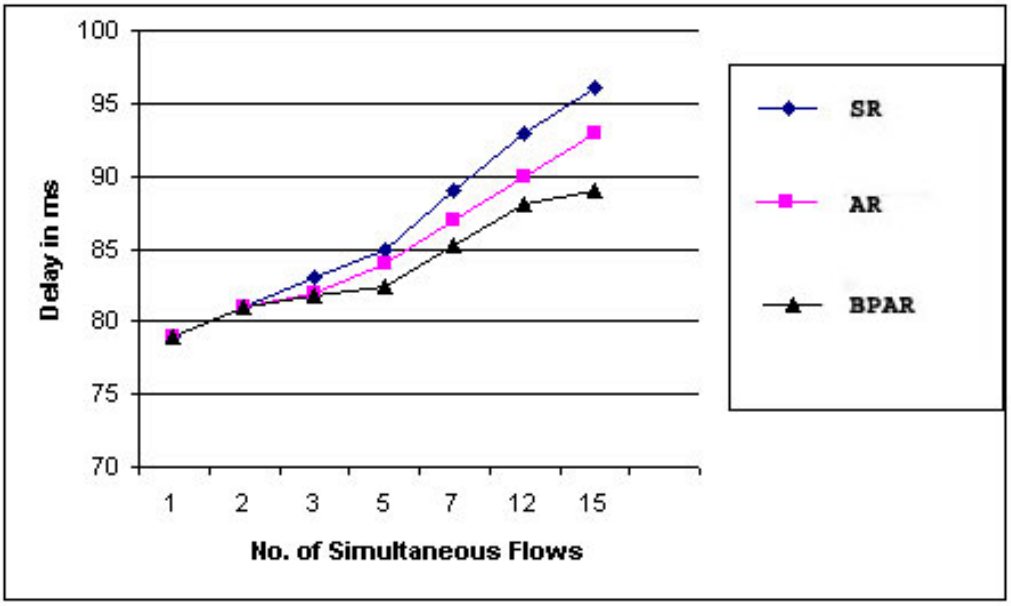

Figure 5-2 Delay statistics with talkspurt voice traffic trace 1

Figure 5-2 illustrates the average transmission delays with the first type of talkspurt traffic trace. It shows consistent results with the results obtained when there was a continuous data flow. With SR and 7 flows the average delay is about 88 ms, whereas with 15 simultaneous flows, the average delays are approximately 93 ms. For a similar

setup with the AR scheme, we observe that the performance is better by nearly 5% in each case. Our BPAR scheme continues to show a better performance even with the talkspurt type of data. With 7 connections, it has an average delay of 86 ms while with 15 connections it shows an improved performance by nearly 8 % over the SR case and 6 % over AR scheme.

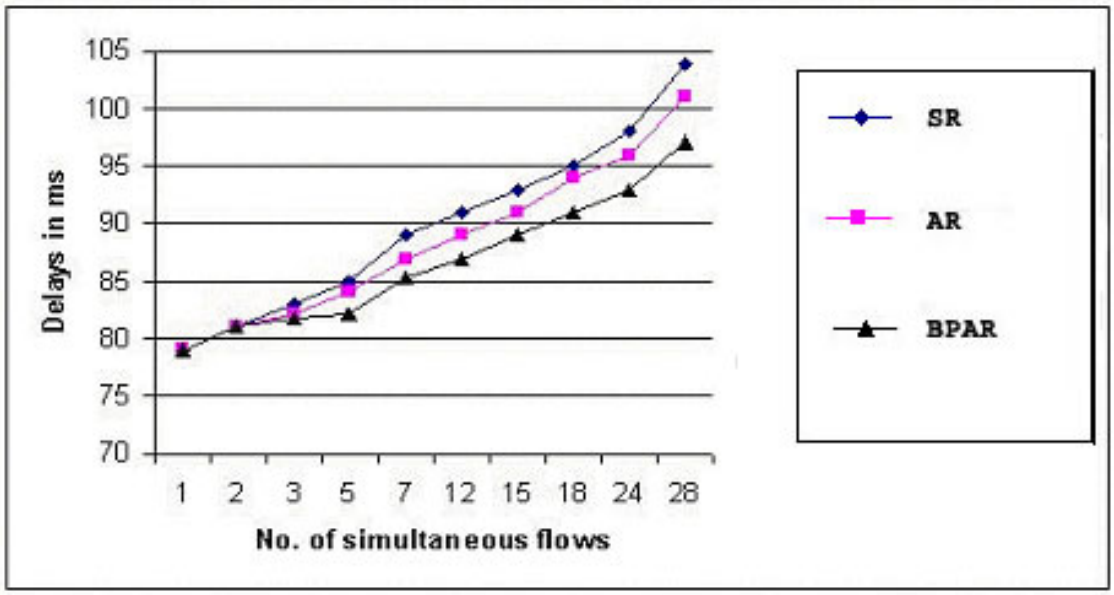

Figure 5-3 Average delays with varying number of simultaneous flows for compressed voice traffic (GSM).

Figure 5-3 shows the average delays for the compressed voice traffic. It shows a marked increase in the delays with the SR and AR schemes as the number of simultaneous flows goes on increasing. With SR the delays are nearly 105 ms and 101 ms with AR scheme. In comparison, the BPAR scheme offers a delay performance that remains lower than the others consistently by a margin of nearly 15 % over the SR routing scheme and 8% over the AR scheme. These results are corroborated by the earlier results with the baseline traffic and the talkspurt traffic emulations.

A comparison of the behavior with the three types of traffic should ideally yield a behavior in which the results with the flat--rate traffic are found to be lesser than or equal

to those with the talkspurt traffic, which in turn should perform better than the talkspurt type of traffic. This observation is based on the fact that the three types of traffic have a decreasing amount of bandwidth requirements. The results obtained are in line with this observation. For example with the BPAR scheme, with 12 connections, the baseline traffic has an average delay of 92 ms. At the same number of connections, the talkspurt traffic shows an average delay of 90 ms while the compressed traffic shows a delay of 87 ms. These results are consistent with the assumption made about the performance of the scheme with the three kinds of traffic.

5.2 Admission Behavior

The admission protocol also needs to be evaluated in term of the probability of accepting for new calls when the capacity of the route is not reach. It should be pointed that even the route can accommodate more calls; the short--term variation of network congestion can still reject a new call. Thus, a better indication of the admission behavior is the probability of acceptance at the first call attempt.

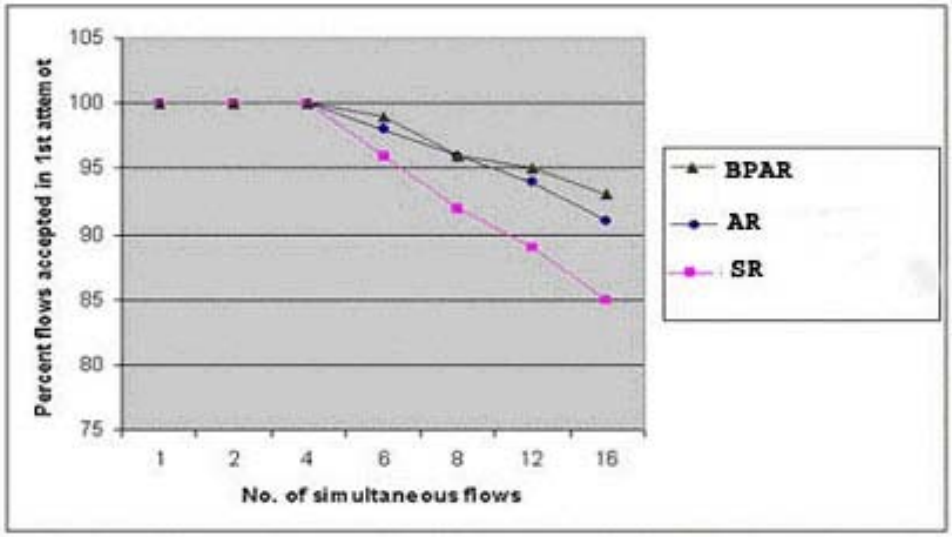

Figure 5-4 Percent flows accepted in first attempt for flat--rate CBR voice traffic.

The figure above illustrates the number of flows that are accepted in the first attempt. When the system load is low (less than 4 calls), 100% of the new calls are admitted. The number of flows accepted starts decreasing after 4 connections for all routing schemes. However for the SR scheme, the percentage of admitted calls drop down by nearly 10% at 12 calls. At 16 connections, the percentage of flows admitted at the first attempt is 85%. The AR scheme has a rate of 93% and 91 % for 12 and 16 connections while the BPAR schemes are observed to have a rate of 95% and 93% on first attempt with 12 and 16 connections. Thus the BPAR scheme has a higher acceptance rate than that of the AR scheme, which in turn performs better than the SR. This suggests that the alternate routes and dynamic rerouting effectively ensures that fewer amounts of flows are rejected the first time and correspondingly lesser amounts of flows are tried for a second time.

Another measurement of admission performance is the overall probability of acceptance. We will show the percentage of acceptance for the three voice traffic types in the rest of this subsection. Whereas figure 13 is useful to point out the efficiency of the scheme at first attempt, it is not the complete indicator of the overall performance of the scheme by itself. To get that, we need to take a look at the overall connection admission performance of the three schemes. Figure 14 analyzes the connection admission probabilities of the three schemes considering the overall admittance rate, not just the first attempt.

The figure 5-5 illustrates the effects of increasing the number of flows on the number of flows admitted in total. The SR scheme shows an admission rate of 100 %

when the number of simultaneous flows is less than 5. However we observe a steady decline in the percentage as the number of flows increases.

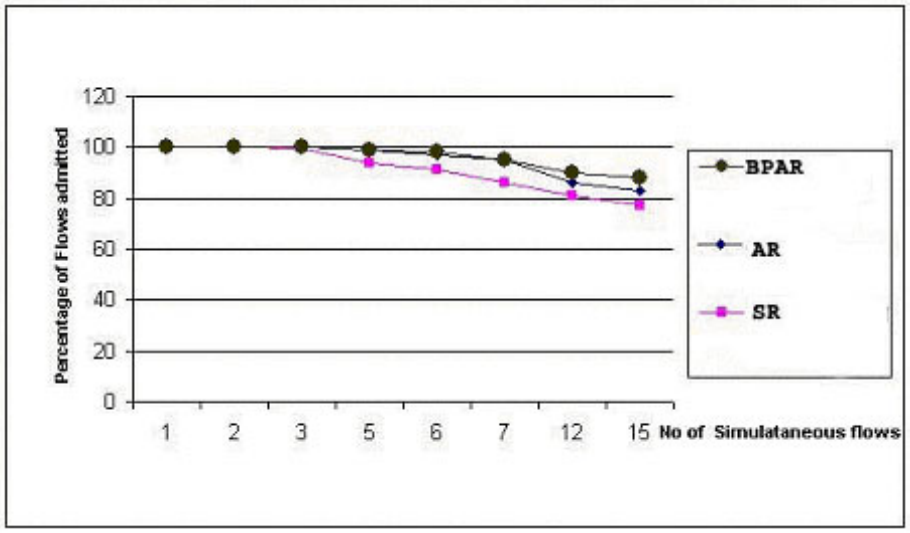

Figure 5-5 Approximate percentage of new flows with varying number of flows for the fixed--rate CBR voice traffic

At 7 simultaneous flows, it has an admission rate of 83% and at 15 connections it shows an admission rate of only 79%. The decrease of the admission rate for the new incoming flows with the SR routing scheme is caused by the increasing congestion of the existing route, where the new flows are not considered for transmission along other paths.

With the AR scheme, the new calls can be routed on an alternate route in the case of network congestion. As can be observed in figure 5-5, the admission rate for AR scheme remains at 100% up to 7 flows. It also shows an admission rate of 86% and 82 % for 12 and 15 connections respectively, which is higher than the SR scheme. The BPAR scheme shows the highest admission rate. With 12 and 15 connections, 90 % and 88 % of the new flows are accepted. This is a 13% increase over the SR routing and 6 % over the AR scheme. This suggests that with the BPAR scheme, the flows that are already admitted are rerouted when number of flows increase. Thus there is a more balanced load for the feasible path, which results in a greater chance for the new flows to be accepted.

Table 2 Percentage of packets taking alternate routes

No. of Simultaneous Flows	% Through First Route	% Through Alternate routes
1	94	6
2	84	16
4	81.5	18.5
8	69	31
15	64	36

The effect of alternate rerouting is also verified by the percentage of packets that take alternate routes for the BPAR scheme, shown in Table 2. As the number of simultaneous connections goes on increasing, the percentage of the packets taking the alternate routes also increases. At 8 connections, 69 % of the packets follow the originally chosen route whereas at 15 connections nearly 36 % of the packets did not follow the original chosen route.

A similar performance trend for the admission can be found when the underlying voice traffic is encoded with talkspurt modeling and compression. Figure 5-6 shows the number of sessions admitted with the different types routing tried and the first of the talkspurt models used. Again the performance with the rerouting is seen to remain consistent with the observations from the experiments with the baseline continuous data that was used in the earlier experiments. The admittance rates for the BPAR scheme with 15 simultaneous flows shows an improved performance by about 5 % over the AR scheme and 11 % over the SR routing scheme. Thus these observations prove that the results with the baseline type of data were consistent and reflective of the general trend of results that have been obtained with the BPAR scheme.

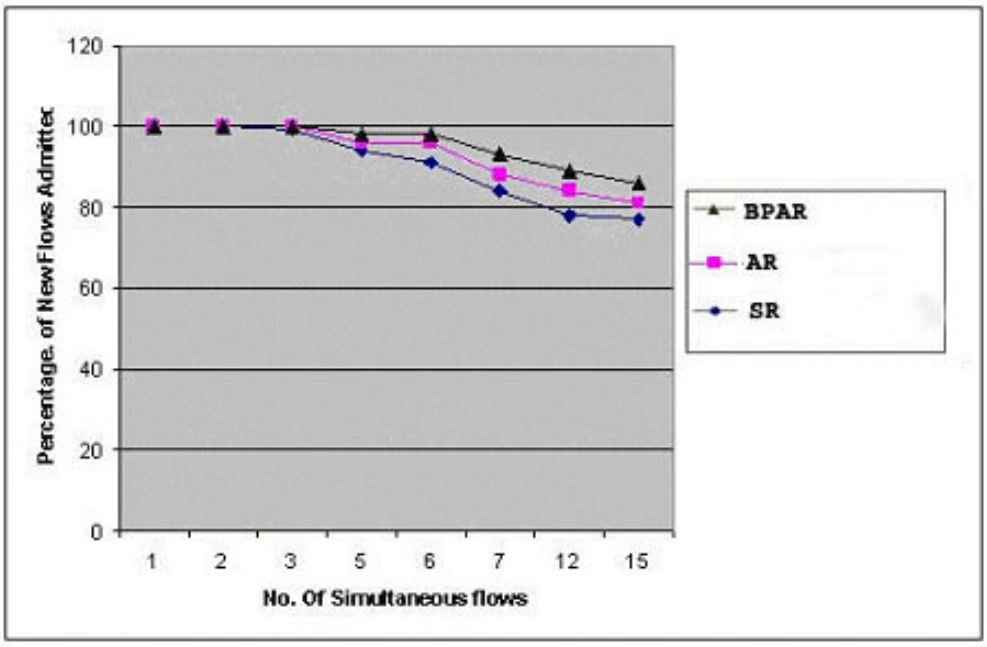

Figure 5-6 Percentage of new flows admitted with varying number of active flows, talkspurt trace 1.

Figure 5-7 shows the average number of sessions admitted when all the voice traffic are compressed. The trend that was observed with the baseline traffic and the talkspurt type traffic is also seen in the results obtained with the simulated compressed real--time data traffic. In this case with the data rate being less, we allow a higher number of simultaneous flows. When the total number of connections is 15 the SR admitted about 90% of the total flows, while the AR scheme admitted 94% of the total flows. With a higher number of flows the SR performance shows a steeper decline and allows only 82% of the flows whereas the AR scheme allows 87%. In contrast to both these schemes, the BPAR scheme allows 97% of traffic for 15 connections and 93% for 28 connections. This is a gain of nearly 9% over the AR scheme and a 16% increase over the SR method.

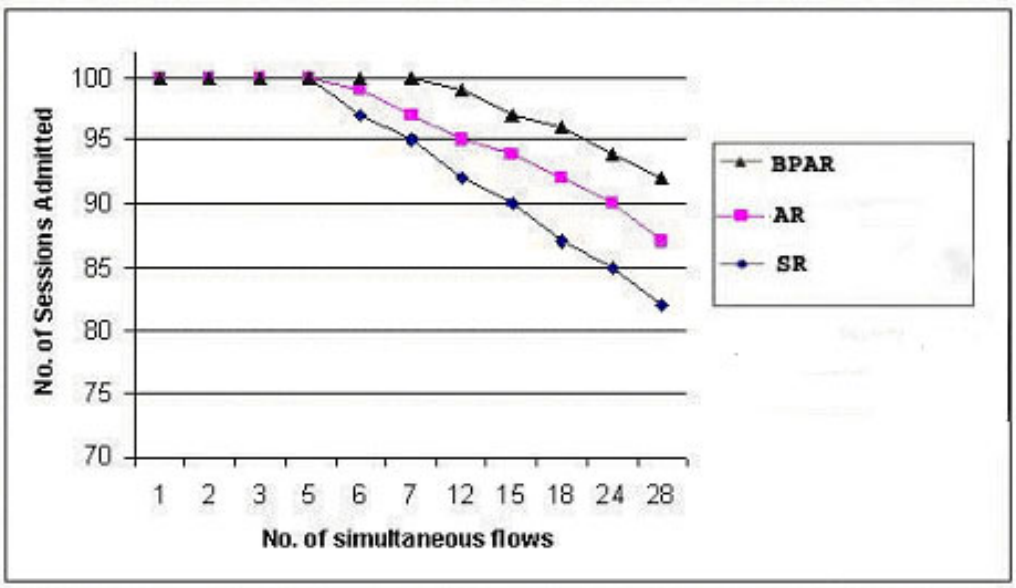

Figure 5-7 Average number of sessions admitted with variable number of flows for compressed voice traffic.

To carry out a comparison of the performance of the BPAR scheme with the three types of traffic used, we consider the case with 12 connections. The baseline traffic results show that 90 % of the incoming connections get admitted with the BPAR scheme. With the talkspurt traffic, this figure is at 91 %, while for the compressed traffic, as many as 97 % of the connections are admitted. Thus the BPAR scheme is seen to have a better performance with the compressed type of traffic.

Based on the results obtained for all three types of simulated traffic, the BPAR scheme is seen to have a better performance than the other two schemes, when the total number of sessions that are admitted into the network are considered.

5.3 Rerouting Observations and Buffer Requirements

Another important observation is the number of flows that actually get rerouted during call time. This would reflect whether the amount of rerouting actually affects the network throughput and performance. Table 3 provides the percentage of rerouted connections for the three routing schemes. For example, 20% means that out of the five

times that the flow was used on that path, it was rerouted once and at other times, it continued to use the initially chosen path till it concluded.

As can be seen from the table, the new incoming flows have to be rerouted onto alternate routes, so that they can at least go through rather than suffer complete rejection. As the number of active connections becomes greater, the chance of rerouting for the flow occurring is greater.

Table 3 Rerouting statistics

Flow No	1	6	8	10	15
Rerouting	0%	20%	40%	60%	60%

An important factor that needs to be considered is the actual time required for the rerouting. This time includes the time required for buffering, re--probing and setting a new route for the connection. Assuming the worst--case scenario, the BPAR scheme always buffers the packets for the duration of probing the alternate paths, and then for the time when actually the route is switched.

Table 4 Approximate rerouting times

No. of Simultaneous Flows	Approx. Rerouting time (ms)
1	1470
2	1491
3	1522
5	1573
8	1616
12	1637
16	1698

The table shows emulation results for the rerouting time. The average rerouting time from these statistics is around 1574 ms. As can be seen; the number of simultaneous flows has a positive effects to the rerouting time. With one connection, the time required

to switch the call from one route onto the other is approximately 1500 ms. There is no considerable increase until the number of simultaneous flows is more than 5. When the number of connections reaches to 15, the rerouting takes 1700 ms.

A higher average rerouting time indicates a higher buffer requirement at the source. Before the rerouting of the flow can be actually done, there will be a need for some additional buffer space in this scheme. This happens because during the time that the source decides to switch the route and actually does it, it might have to buffer the packets for a short period, ranging from a few milliseconds to few seconds. However, the voice traffic is still continuously delivered along the original connection with the achievable quality until the new connection can be setup.

Using the formula and the rerouting statistics provided in the previous chapter, we calculated the estimated buffer requirements. The actual buffer required using the emulations was plotted as a comparison with the estimated values. This is shown in the figure below.

Figure 5-8 shows a maximum estimated buffer requirement of 670 Kbytes when there are 15 connections and a minimum of 50 Kbytes when the number of connections is just 1. The actual buffering observed is slightly higher than the estimated requirement. At 15 connections, the actual requirement is nearly 80 Kbytes higher than the estimated requirement. The trend for any number of connections is the same with the actual requirement showing a higher value than the calculated buffer space. The actual requirement varies from the calculated one because it is very difficult to correctly estimate how much time the source will need to reroute a flow.

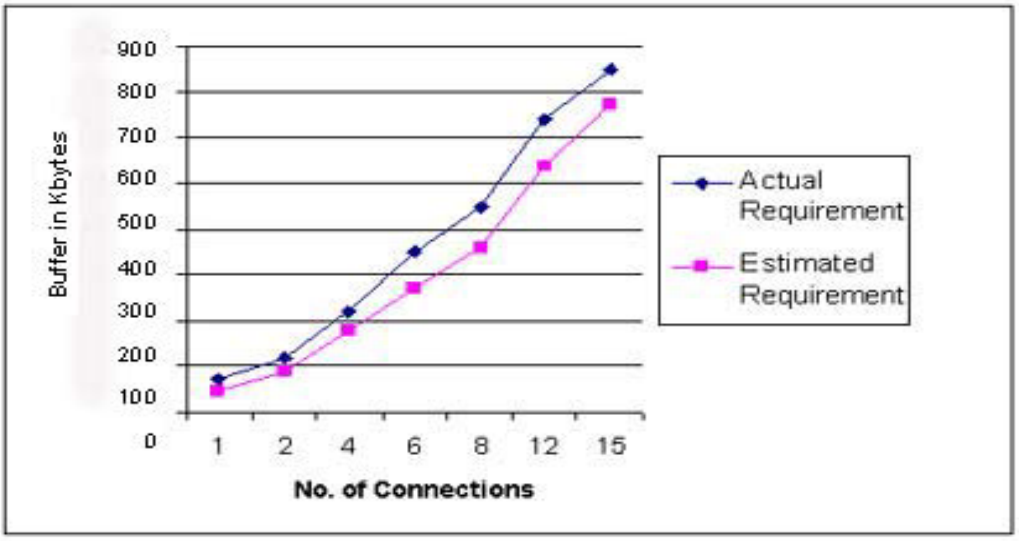

Figure 5-8. Estimated and actual buffer requirements.

The estimation is based on the average rerouting time of all the connections. In practice this rerouting time varies for each flow. It is therefore possible that some of the connections end up buffering for more time than estimated. Hence the amount of buffering that is needed varies from the actual obtained values.

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Conclusion

The widespread use of the Internet has spilled over onto the transfer of multimedia over the Internet. More and more real--time applications are being designed for use on the Internet. In the current scenario, because of the sharing of the networks with the non real--time best--effort traffic, it is impossible to guarantee a single path for routing and as such it is very difficult to assure a certain level of quality to the users even with reservations along the links. The routing of the real--time flows is very important in ensuring that the desired quality levels are attained.

The proposed rerouting scheme ensures that the flows get the optimal but available path, which might not be the shortest or even the best. However it ensures that the chosen path delivers the required levels of Quality of Service. It reduces a lot of overhead for the intermediate nodes by reducing the amount of decision making that they have to do. With compressed traffic, the performance is better by approximately 12% in terms of the delays and up to 16% in the number of connections admitted to the network when compared to the simple and AR schemes without rerouting. The documented results show that the performance of the rerouting scheme is encouraging enough to be tested on a larger scale. Importantly, we have to give consideration to the fact that as people realize the immense usefulness of using the Internet for such real--time applications, it is very feasible to think in terms of a parallel network for real--time

traffic. It would not only reduce a great amount of load from the current infrastructure, but also would create great avenues for improving the quality of the real--time applications in use currently.

6.2 Future Work

A very desirable enhancement to the application would be to incorporate flow labels in the packets that could increase throughput. Also the application currently assumes the knowledge of the network nodes. A route discovery algorithm would be a very welcome addition to the source node's capabilities. It is also recommended that the facility to create some reservation scheme also along the path be incorporated. This could help the system to increase the overall performance. These are enhancements to the core structure and can be done according to user needs.

APPENDIX
RELATED SOURCE CODE

```
/**** udpcli.c***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <math.h>
#include <netdb.h>
#include <sys/times.h>

#define MYPORT 4950    // the port users will be connecting to

  int probe(char host[]){
    struct sockaddr_in their_addr1; // connector's address information
    struct hostent *he1;
    struct timeb tp1,tp2;
    time_t t1,t2;
    int sockfd1;
    int port=2000;
    int n;
    char buf[200];
    int addr_len,numbytes;
    if ((he1=gethostbyname(host)) = = NULL) {  // get the host info
      perror("gethostbyname");
      exit(1);
    }
    if ((sockfd1 = socket(AF_INET, SOCK_DGRAM, 0)) = = -1) {
      perror("socket");
      exit(1);
    }
    their_addr1.sin_family = AF_INET;
    their_addr1.sin_port = htons(port);
    their_addr1.sin_addr = *((struct in_addr *)he1->h_addr);
    memset(&(their_addr1.sin_zero), '\0', 8);
```

```
  ftime(&tp1);
  t1= time(NULL);
          if ((numbytes=sendto(sockfd1,send_buf,strlen(send_buf),0,(struct sockaddr

              *)&their_addr1, sizeof(struct sockaddr))) = = -1) {

    perror("sendto");
    exit(1);
  }
  system(sleep(1));
          if ((numbytes=recvfrom(sockfd1, buf, strlen(send_buf) , 0,(struct sockaddr

              *)&their_addr1, &addr_len)) = = -1) {

    perror("recvfrom");
    exit(1);
  }
  printf("HI");
  t2= time(NULL);
  ftime(&tp2);
  close(sockfd1);
  if (strcmp(host,"bog.cise.ufl.edu")= =0)time1 =tp1.millitm-tp2.millitm;
  if (strcmp(host,"bog.cise.ufl.edu")= =0) time1=10;
  else if (strcmp(host,"leaf.cise.ufl.edu")= =0)time2 =tp1.millitm-tp2.millitm;
  if (strcmp(host,"leaf.cise.ufl.edu")= =0) time2=13;
  if (strcmp(host,"sun114-21")= =0)
    return 10;
  if (strcmp(host,"leaf")= =0)
    return 12;
}
int main(int argc, char *argv[])
{
  int sockfd,i,tmp,tmp2,k;
  int time1,time2;
  struct sockaddr_in their_addr; // connector's address information
  struct hostent *he;
  int numbytes;
  struct tms t1,t2;
  int start_count=1,send=1;
  int  * st, * init, * init2;
  int  * bf;
  char ch;
  int bf2,j;
  int size;
  FILE * fp;
  int filesize;
```

```
if (argc != 3) {
  fprintf(stderr,"usage: cli hostname1 hostname2 \n");
  exit(1);
}
time1=probe(argv[1]);
time2=probe(argv[2]);
if (time1<=time2){
  if ((he=gethostbyname(argv[1])) = = NULL) {  // get the host info
    perror("gethostbyname");
    exit(1);
  }
}
else{
  if ((he=gethostbyname(argv[2])) = = NULL) {  // get the host info
    perror("gethostbyname");
    exit(1);
  }
}
if ((fp= fopen("BT0A0A7.WAV","rb" )) = = NULL)
  fprintf(stderr, "Cannot open file");
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) = = -1) {
  perror("socket");
  exit(1);
}
st = (unsigned int *) malloc(200*sizeof(int));
bf = (unsigned int *) malloc(200*sizeof(int));
tmp = (int)st;
init = st;
init2 = bf;
tmp2= (int)bf;
for (i=1;i<290;i=i+1){
 fgets(bf,200,fp);
 strcat(st,bf);
 realloc(st,(100*sizeof(char)));
}
fseek (fp,0,2);
bf2=fgetc(fp);
filesize= ftell(fp);
fseek(fp,0,0);
bf2= fgetc(fp);
for (i=0;i<=(filesize/4);i=i+1){
  bf2 = fgetc(fp);
  fseek(fp,0,i);
  * st = bf2;
  st++;
}
```

```
    tmp = ((int)st-tmp-1);
    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(MYPORT); // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8);  // zero the rest of the struct
    st = init;
    for(;;){
      if (start_count = =1){
        times(&t1);
        start_count=0;
      }
      if (send= =1){
        for(i=0;i<=60;i++){
          for (j=0;j<=100;j++){
            *bf = *st;
            st++;
            bf++;
            }
          tmp2= ((int)bf-tmp2);
        if ((numbytes=sendto(sockfd, bf,tmp2, 0,(struct sockaddr *)&their_addr,
        sizeof(struct sockaddr))) = = -1) {
            perror("sendto");
            exit(1);
          }
          system(sleep(0.2));
          bf = init2;
          tmp2=(int)bf;
          }
      }
      times(&t2);
      if(t2.tms_stime - t1.tms_stime= =16){
        start_count =1;
        send = 0;
      }
    if((t2.tms_stime - t1.tms_stime= =2)&& send = = 0){
        start_count =1;
        send = 1;
        system(sleep(2));
      }
      st=init;
    }
    close(sockfd);
    return 0;
  }

/***probe.c**/
```

```
 Original Author: James Hall (jch1003@cl.cam.ac.uk)*//
/** Modified by Narasinha Kamat ********/

#ifdef linux
#include "linux-tweaks.h"
#endif
#include <pthread.h>
#include <stdio.h>
#include <pwd.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <getopt.h>
#include "list.h"
#include "jroute.h"
#include "jroute_threads.h"

char *prog;
char errbuf[132];
int verbose = 0;
int simple_output = 0;
int sort = 0;
int naddr = 0;                  /* # addresses if in input preamble */
int naddr_done = 0;             /* #addresses actually getrouted */
int s;                         /* (icmp) socket file descriptor */
int main_pid;
int probe_type = PROBE_TCP;
int (*get)();                   /* input function */
void usage(){
fprintf(stderr, "%s [-v(erboser)] [-q(uieter)] [-Q(uite like traceroute)] [-t(hreads)]\n\t [-
m(ax-ttl)] [-f(irst-ttl)] [-r(oute)] [-o(utput file] [-s(ort)] \n\t[-S(et src address)] [-I set
interface] [-g(ive up after n timeouts)] \n\t[-b generate probes and replies at most
nKb/s]\n\t[-B generate src routed packets at most n/s] [-h(elp)] <file/host>\n", prog);
}

void help(){
  usage();
  printf("\n");
```

```
  printf("Input lines have form \"(addr1+addr2+...+)host(/f/m$label) <CR>\"\n");
  printf("\twhere:\taddrs form a list of loose source routed requirements\n");
  printf("\t\tf/m = min/max probe ttl to employ\n");
  printf("\t\tand optional label will be quoted with result\n");
  printf("\t\tlsr list, either/both of /f /m and label are optional\n");
  printf("\tif any option is not supplied the current default is used\n\n");

  printf("flags \"-vqmfrbBh<CR>\" may be inserted anywhere in the input stream and will
\n\tact/set current defaults from that point\n");
  printf("\t (if preceeded by a double \"--\" will take effect immediately)\n");
  printf("Current defaults can be cleared by entering flag followed by \"-\"\n");
  printf("\ti.e. to set source routing from current point in input:\n");
  printf("\t\t\"-raddr1+addr2+...\"\n");
  printf("\tand to clear it:\n");
  printf("\t\t\"-r-\"\n");
  printf("for more look at the source\n\n");
  printf("\n");
  return;
}
void opterror(char *msg){
  fprintf(stderr, "%s: %s\n", prog, msg);
  usage();
  exit (1);
}
void error(char *msg){
  fprintf(stderr, "%s: ", prog);
  if (errno)
    perror(msg);
  else
    fprintf(stderr, "%s\n", msg);
  exit (1);
}
void error2(char *msg1, char *msg2){
  fprintf(stderr, "%s: %s: ", prog, msg1);
  if (errno)
    perror(msg2);
  else
    fprintf(stderr, "%s\n", msg2);
  exit (1);
}
FILE *openf(char *fnm){
  FILE *f;
  if ((f = fopen(fnm, "r")) = = NULL)
    error2(fnm, "input file - open error");
  return f;
}
```

```
void command(char *comm, char *allowed){
  int i;
  TRC(fprintf(stderr, "command %s\n", comm);)
  if (*comm = = '-')                /* double dash - already done */
    return;
  while (strlen(comm))
    {
     if (!strchr(allowed, *comm))
          {
           comm++;
           continue;
          }
     switch (*comm)
          {
          case 'v':
            if (nthreads = = 1)
              {
                verbose++;
                simple_output = 0;
              }
            comm++;
            break;
          case 'q':
            if (nthreads = = 1)
              {
                verbose = MAX(0, verbose-1);
                simple_output = 0;
              }
            comm++;
            break;
          case 'Q':
            if (nthreads = = 1)
              {
                verbose = 0;
                simple_output = 1;
                }
            comm++;
            break;
          case 'm':
            con_def.ttl_max = strtol(comm+1, &comm, 0);
            con_def.ttl_max = MIN(con_def.ttl_max, TTL_MAX);
            if (!con_def.ttl_max)
              con_def.ttl_max = TTL_MAX_DEF;
            break;
          case 'f':
            con_def.ttl_min = strtol(comm+1, &comm, 0);
```

```
            con_def.ttl_min = MAX(con_def.ttl_min, TTL_MIN_DEF);
            if (!con_def.ttl_min)
              con_def.ttl_min = TTL_MIN_DEF;
            break;
          case 'g':
            giveup = strtol(comm+1, &comm, 0);
            break;
          case 'r':
            comm++;
            if (*comm = = '-')
              {
               con_def.route.num = 0;
              }
            else
              {
               if (!parseroute(&con_def.route, comm))
                    {
                     fprintf(stderr, "# ignoring default lsr setting: %s\n",
                              comm);
                     con_def.route.num = 0;
                    }
               while (*comm != ' ')
                    comm++;
              }
            break;
          case 'b': maxrate = strtol(comm+1, &comm, 0); break;
          case 'B': maxrate = -strtol(comm+1, &comm, 0); break;
                    case 'h': help(); comm++; break;
          }
    }
  return;
}
int get_stdio(char *buf, FILE *i){
  char *nlp;
  if (feof(i)){
      return 0;
    }
  else{
      if (fgets(buf, MAX_INDATA_SZ, i) = = NULL){
            if (feof(i)){
               return 0;
              }
            else{
               fprintf(stderr, "%s: input read error\n", prog);
               exit (1);
              }
```

```
            }
        nlp = strchr(buf, '\n');
        *nlp = '\0';
      }
    return 1;
}
int tread(int f, char *buf, int in, int out, int block){
    int i;
    int to_read;
    int nread;
    int got = 0;
    int end = TNET_INBUF_SZ;
    int mark = in;
    int on = 1;
    static state_t state = STATE_NORMAL;
    if (!block && ioctl(f, FIONBIO, (char*)&on) < 0)
      error("FIONBIO unblock");
    to_read = in < out ? out - in : end - in;
    nread  = read(f, &buf[in], to_read);
    if (nread < 0){
        if (! block && errno = = EWOULDBLOCK)
            nread = 0;
        else
            error("read");
      }
    else if (nread = = 0){}
    else{
        got += nread;
        in = (in + nread)%TNET_INBUF_SZ;
      }
    on = 0;
    if (!block && ioctl(f, FIONBIO, (char*)&on) < 0)
      error("FIONBIO block");
    for (i = got; i > 0; mark = (++mark)%TNET_INBUF_SZ, i--){
        unsigned char c = buf[mark];
        switch(state){
            case STATE_NORMAL:
              if (c != IAC){
                  TRC(printf("%02x ", c);)
                  fflush(stdout);
                  /* check for commands to apply immediately */
                  if (c = = '-' && buf[mark+1] = = '-'){
                        char cbuf[TNET_INBUF_SZ], *cp = cbuf;
                        int cindx = mark+2; /* start after dashes */
                        while (buf[cindx] != '\x0d')
                          *(cp++) = buf[cindx++];
```

```
            *cp = '\0';
            command(cbuf, "mfvqQrbBhg");
          }
    }
   else{
      /* shift into the command state */
      state = STATE_IAC;
    }
   break;

 case STATE_IAC:
   switch(c){
     case IP:
       exit(0);
       break;

     case SE:
     case NOP:
     case DMARK:
     case BRK:
     case AO:
     case AYT:
     case EC:
     case EL:
     case GA:
     case SB:
       printf("ignoring option %02x\n", c);
       break;

     case WILL:
       printf("WILL ");
       state = STATE_OPT;
       break;
     case WONT:
       printf("WONT ");
       state = STATE_OPT;
            break;
     case DO:
       printf("DO   ");
       state = STATE_OPT;
       break;
     case DONT:
       printf("DONT  ");
       state = STATE_OPT;
       break;
```

```
            default:
              printf("unhandled IAC %02x\n", c);
              break;
            }
           if (state = = STATE_IAC)
            state = STATE_NORMAL;
           break;
          case STATE_OPT:
           switch(c){
            case TMARK:
             {
                  unsigned char reply[] = {IAC, WILL, TMARK};
                  /* send WILL TMARK */
                  write(2, reply, sizeof(reply));
                  printf("tmark\n");
                  state = STATE_NORMAL;
                  break;
             }
            default:
             printf("ignoring option code %02x\n", c);
             state = STATE_NORMAL;
             break;
            }
          }
        }
  return got;
}
int get_tnet(char *obuf, FILE *infile){
  int i = 0;
  int nread;
  int gotline = 0;
  int block = BLOCK;
  static unsigned char ibuf[TNET_INBUF_SZ];
  static int in = 0;
  static int out = 0;
  static int bytes = 0;
  do{
      for (; bytes; out = (++out)%TNET_INBUF_SZ, --bytes)
          {
           switch (ibuf[out]){
            case '\x0a':   /* LF */
              gotline++;
              obuf[i] = '\0';
            case '\x0d':  /* DROP THROUGH - clear CR */
              break;
            case '\x04':             /* EOF */
```

```
            printf("#Closing connection\n");
            return 0;
            break;
          default:
            obuf[i++] = ibuf[out];
            if (i = = MAX_INDATA_SZ)
                error("telnet input line too long");
            break;
          }
        if (gotline){
            out = (++out)%TNET_INBUF_SZ;
            --bytes;
            break;
          }
        }
    nread = tread(0, ibuf, in, out, gotline ? NO_BLOCK : BLOCK);
    bytes += nread;
    in = (in + nread)%TNET_INBUF_SZ;
  } while (!gotline);
 return 1;
}
int main(int argc, char **argv){
 char *fnm;
 char *cp, opt;
 FILE *infile = NULL;
 char inbuf[MAX_INDATA_SZ];
 char ofilenm[PATH_MAX];
 int n, i;
 char *source = NULL, *device = NULL;
 struct sockaddr_in wherefrom;        /* Who we are */
 struct ifaddrlist *al;
 struct hostinfo *hi, *hi_ss;
 uint32 *ap;
 addr_un_t to;
 src_route_t route_try;
 char clbuf[CL_BUFSZ], *p = clbuf;
 struct passwd *pwd;
 time_t tm;
 char hname[50];
 struct timeval finish;
 struct protoent *pe;
 if ((cp = strrchr(argv[0], '/')) != NULL)
   prog = cp + 1;
 else
   prog = argv[0];
 main_pid = getpid();
```

```
*p = '\0';
for (i = 1; i < argc; i++)
  p+= sprintf(p, "%s ", argv[i]);
printf("# %s: invoked %s\n", prog, clbuf);
pwd = getpwuid(getuid());
if (pwd){
    printf("# by %s ", pwd->pw_gecos);
     if (!gethostname(hname, 50))
          {
            printf("@%s ", hname);
          }
   }
else{
    printf("# ");
  }
  if ((tm = time(NULL)) = = -1)
  error("time error");
printf("%s", ctime(&tm));
opterr = 0;
route.num = 0U;
get = get_stdio;
ofile = stdout;                    /* default */
while ((opt = getopt(argc, argv, "vqQsDXt:m:f:r:o:S:I:g:b:B:h")) != EOF)
  switch (opt)
    {
    case 'v': verbose++; simple_output = 0; break;
    case 'q': verbose--; simple_output = 0; break;
    case 'Q': simple_output = 1; break;
    case 's': sort++; break;
    case 't': nthreads = atoi(optarg); break;
    case 'm': con_def.ttl_max = atoi(optarg); break;
    case 'f': con_def.ttl_min = atoi(optarg); break;
    case 'r':
         if (!parseroute(&con_def.route, optarg))
         error2("lsr parse fail", optarg);
         break;
    case 'o': strcpy(ofilenm, optarg); ofile = open_ofile(ofilenm, 0); break;
    case 'g': giveup = atoi(optarg); break;
    case 'X':                      /* drop through */
          giveup = 1;

    case 'D': get = get_tnet; break;
    case 'S':
         source = optarg;
         break;
    case 'I':
```

```
        device = optarg;
        break;
    case 'b': maxrate = atoi(optarg); break;
    case 'B': maxrate = -atoi(optarg); break;
    case 'h': help(); return 0; break;
    case '?': opterror("Unknown option"); break;
    case ':': opterror("Missing parameter"); break;
    default: opterror(""); break;
    }
if (getuid() && geteuid())
  error("Sorry can't help you - must have root privileges.");
if ((pe = getprotobyname("icmp")) = = NULL)
  error("icmp protocol unknown!");
if ((s = socket(AF_INET, SOCK_RAW, pe->p_proto)) < 0)
  error("icmp socket");
if (argc  = = optind)
  error("no input file or host specified");
strncpy(inbuf, argv[optind], MAX_INDATA_SZ);
cp = strchr(inbuf, ':');
if (cp)
  *cp = '\0';
if (!parseroute(&route_try, inbuf)) /* input must be a file */ {
    fnm = argv[optind];
    if (*fnm = = '-')
        {
          fprintf(stderr, "# - using std in\n");
          infile = stdin;
        }
    else
        {
          fprintf(stderr, "# - assuming \"%s\" is an input file\n", inbuf);
          infile = openf(fnm);
        }
  }
else/* one dest. input as c.l. arg. */    {
    naddr = 1;
    nthreads = 1;
    if (simple_output)
        verbose = 0;
    else
        verbose = MAX(verbose, 1);
    if (route_try.num = = 1)
        {
          to.addrint = route_try.rt[0].addrint;
          route_try.num = 0;
        }
```

```
    else
        {
         to.addrint = route_try.rt[route_try.num-1].addrint;
         route_try.rt[route_try.num-1].addrint = 0U;
         route_try.num--;
         if (route_try.num > con_def.route.num)
           con_def.route = route_try;
        }
  }
/* don't produce gibberish */
if (verbose || simple_output)
  nthreads = 1;
if (con_def.ttl_min < TTL_MIN_DEF || con_def.ttl_min > TTL_MAX){
    fprintf(stderr, "%s: minimum ttl %d specified\n",
            prog, con_def.ttl_min);
    return 1;
  }
if (con_def.ttl_max > TTL_MAX || con_def.ttl_max < TTL_MIN_DEF){
    fprintf(stderr, "%s: maximum ttl %d specified - max is %d\n",
            prog, con_def.ttl_max, TTL_MAX);
    return 1;
  }
if (nthreads > N_THREADS_MAX){
    fprintf(stderr, "%s: %d threads specified - max is %d\n",
            prog, nthreads, N_THREADS_MAX);
    return 1;
  }
else if (nthreads <= 0){
    fprintf(stderr, "%s: %d threads specified\n",
            prog, nthreads);
    return 1;
  }
/* Get the interface address list */
n = ifaddrlist(&al, errbuf);
if (n < 0)
  error2("ifaddrlist", errbuf);
if (n = = 0)
  error("Can't find any network interfaces");
/* Look for a specific device */
if (device != NULL){
    for (i = n; i > 0; --i, ++al)
        if (strcmp(device, al->device) = = 0)
          break;
    if (i <= 0)
        error2("Can't find interface", device);
  }
```

```
/* Determine our source address */
if (source = = NULL){
   setsin(&wherefrom, al->addr);
   if (n > 1 && device = = NULL){
         fprintf(stderr,
                  "# %s: Warning: Multiple interfaces found; using %s @ %s\n",
                  prog, inet_ntoa(wherefrom.sin_addr), al->device);
        }
  }
else{
   hi = gethostinfo(source);
   if (hi = = NULL)
        error2("no such host as", source);
   source = hi->name;
   hi->name = NULL;
   if (device = = NULL){
         setsin(&wherefrom, hi->addrs[0]);
         if (hi->n > 1)
           fprintf(stderr,
                    "%s: Warning: %s has multiple addresses; using %s\n",
                    prog, source, inet_ntoa(wherefrom.sin_addr));
        }
   else{
         for (i = hi->n, ap = hi->addrs; i > 0; --i, ++ap)
           if (*ap = = al->addr)
             break;
         if (i <= 0){
            fprintf(stderr,
                      "%s: %s is not on interface %s\n",
                      prog, source, device);
            exit(1);
           }
         setsin(&wherefrom, *ap);
        }
   freehostinfo(hi);
  }
fflush(stdout);
setlinebuf(stdout);
if (gettimeofday(&start, NULL) != 0)
  error("gettimeofday - start");
if (infile){
   if (sort)
        do_sorted(&wherefrom, infile, nthreads);
   else
        do_unsorted(&wherefrom, infile, nthreads);
  }
```

```
  else{
     do_singleshot(&wherefrom, to, cp, nthreads);
    }

  if (gettimeofday(&finish, NULL) != 0)
    error("gettimeofday - finish");
  if (ofile != stdout){
     char b[MAX_INDATA_SZ];
     fclose(ofile);
     ofile = open_ofile(ofilenm, 1);
     while (get_stdio(b, ofile)){
           printf("%s\n", b);
          }
    }
  report(&start, &finish);
  fflush(stdout);
  return 0;
}
int do_singleshot(struct sockaddr_in *wherefrom, addr_un_t to, char *cp, int nthreads){
  addr_train_t at;
  all_addrs = &at;
  train_bufsz = 1;
  at.dest.addrint = to.addrint;
  if (cp = = NULL){
     fprintf(stderr, "%s: No port specified - default to 80\n",
               prog);
     cp = ":80";
    }
  at.port = atoi(cp+1);
  at.con = con_def;
  at.label[0] = '\0';
  sort = 1;
  threads_init(wherefrom, nthreads, NULL);
  rthread_inform();
  threads_finish(nthreads);
  return 0;
}

int do_unsorted(struct sockaddr_in *wherefrom, FILE *infile, int nthreads){
  int read;
  int started = 0;
  uint line = 0;
  char inbuf[MAX_INDATA_SZ];
  train_bufsz = nthreads*ADDR_POOL_SZ_FACT;
  if ((all_addrs = (addr_train_t *)malloc(train_bufsz*sizeof(addr_train_t))) = = NULL)
    error("addr train malloc");
```

```
  threads_init(wherefrom, nthreads, infile);
  while (read = get(inbuf, infile)){
     line++;
     if (!strlen(inbuf))
          continue;
     if (*inbuf = = '#'){
           fprintf(ofile, "%s\n", inbuf);
           continue;
          }
     else if (*inbuf = = '-'){
           command(inbuf+1, "vqQmfrbBhg");
           TRC(fprintf(stderr, "# command \"%s\" line %u\n", inbuf+1, line);)
           continue;
          }
     else if (strstr(inbuf, "addresses")){
           fprintf(stderr, "# %s\n", inbuf);
           if (naddr)
           fprintf(stderr,"# another notification of #addresses found (was %d now %d) line
     %u\n", naddr, naddr + atoi(inbuf), line);
           naddr += atoi(inbuf);
           continue;
          }
     else if (add_addr(inbuf, line)){
           started = 1;
          }
     else{
           fprintf(stderr, "# ignored input \"%s\" line %u\n", inbuf, line);
          }
    }
  threads_finish(nthreads);
  return 0;
}

int do_sorted(struct sockaddr_in *wherefrom, FILE *infile, int nthreads){
  char *cp, opt;
  char inbuf[MAX_INDATA_SZ];
  int n, i;
  int naddr_in = 0;
  int started = 0;
  int port;
  while (get(inbuf, infile)){
     if (*inbuf = = '#'){
           if (!started)
             fprintf(ofile, "%s", inbuf);
           continue;
          }
```

```
if (!isdigit(*inbuf)){
        if (*inbuf = = '-')
          command(inbuf+1, "vqbBq");
        else if (!started)
          fprintf(ofile, "#%s", inbuf);
        continue;
      }
if (strstr(inbuf, "addresses")){
        if (naddr){
           error("two notifications of #addresses found");
          }
        else{
           naddr = atoi(inbuf);
           train_bufsz = naddr;
           nthreads = MIN(nthreads, naddr);
           fprintf(ofile, "# %d addresses\n", naddr);
           if ((all_addrs = (addr_train_t *)malloc(train_bufsz*sizeof(addr_train_t)))
                 = = NULL)
               error("addr train malloc");
           threads_init(wherefrom, nthreads, NULL);
           continue;
          }
      }
else{
        char *pp;
        started = 1;
        if (all_addrs = = NULL)
          error("no address count found");
        pp = strchr(inbuf, ':');
        if (pp){
           all_addrs[addr_indx].port = atoi(pp+1);
           *pp = '\0';
          }
        else
          all_addrs[addr_indx].port = 80;
        if ((all_addrs[addr_indx].dest.addrint = inet_addr(inbuf)) = = INADDR_NONE)
          {
           fprintf(stderr, "can't resolve address %s\n", inbuf);
           continue;
          }
        all_addrs[addr_indx].con = con_def;
        addr_indx++;
        if (++naddr_in > naddr)
          break;
        rthread_inform();
        }
```

```
    }
  threads_finish(nthreads);

  fprintf(stderr, "sorting ");
  qsort(all_addrs, naddr, sizeof(addr_train_t), cmp_train);
  fprintf(stderr, "\n");
  /*resolve_to(all_addrs, naddr);*/
  pretty_print_trains(stdout, all_addrs, naddr, sort);
  return;
}

/********udppro.c*****/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <netdb.h>
#define MYPORT 4950    // the port users will be connecting to
#define MAXBUFLEN 10000

   char str[20];
   int timer=0;
   int flag_sun114_21=1,flag_bark=2,flag_leaf=2;
   void sendData(int buf[] , char str1[],int bytes)
   {
      int sockfd;
      struct sockaddr_in their_addr; // connector's address information
      struct hostent *he;
      int numbytes;
      int tmp;
      printf("packet sent to \"%s\"\n",str1);
      if ((he=gethostbyname(str1)) = = NULL) {  // get the host info
        perror("gethostbyname");
        exit(1);
      }
      if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) = = -1) {
        perror("socket");
```

```
    exit(1);
  }

  their_addr.sin_family = AF_INET;     // host byte order
  their_addr.sin_port = htons(MYPORT); // short, network byte order
  their_addr.sin_addr = *((struct in_addr *)he->h_addr);
  memset(&(their_addr.sin_zero), '\0', 8);  // zero the rest of the struct

  if ((numbytes=sendto(sockfd,buf,bytes,0,(struct sockaddr *)&their_addr, sizeof(struct
                                                                          sockaddr))) = = -1){
    perror("sendto");
    exit(1);
  }
  close(sockfd);
}
int delay_probe(){
  int delay_time;
  if ((timer < 10)||(timer>=30)){
    system(sleep(0.01));
    printf("HII %d\n",timer);
    delay_time=0;
  }
  else if ((timer>10) || (timer<30)){
    system(sleep(1));
    printf("HIIII %d\n",timer);
    delay_time=1;
  }
  timer++;
  return delay_time;
}
int probe(char * hostName,int flag){
   int sockfd1,sockfd2;
  struct sockaddr_in their_addr;
  struct sockaddr_in their_addr2;
  struct hostent *he;
  int probe_time;
  struct hostent *he1;
  int numbytes,delay_time;
  char dest_host1[20]="";
  char dest_host2[20]="";
  int port=2000;
  struct timeb tp,tp1,tp2;
  int addr_len;
  char buf[100];
  char recvbuf[100];
```

```
if (strcmp(hostName,"sun114-21")= =0){
  if (flag= =0)strcpy(dest_host1,"bark.cise.ufl.edu");
  else strcpy(dest_host1,"leaf.cise.ufl.edu");

  if ((he=gethostbyname(dest_host1)) = = NULL) {
    perror("gethostbyname1");
    exit(1);
  }
  if (strcmp(hostName,"bark")= =0){
    strcpy(dest_host1,"rain.cise.ufl.edu");
    if ((he=gethostbyname(dest_host1)) = = NULL) {
      perror("gethostbyname1");
      exit(1);
    }
  }
  if (strcmp(hostName,"leaf")= =0){
    strcpy(dest_host1,"mango.cise.ufl.edu");
    if ((he=gethostbyname(dest_host1)) = = NULL) {
      perror("gethostbyname1");
      exit(1);
    }
  }
  if (strcmp(hostName,"mango")= =0){
    strcpy(dest_host1,"sun114-21.cise.ufl.edu");
    if ((he=gethostbyname(dest_host1)) = = NULL) {
      perror("gethostbyname1");
      exit(1);
    }
  }
  if ((sockfd1 = socket(AF_INET, SOCK_DGRAM, 0)) = = -1) {
    perror("socket");
    exit(1);
  }
  their_addr.sin_family = AF_INET;
  their_addr.sin_port = htons(port);
  their_addr.sin_addr = *((struct in_addr *)he->h_addr);
  memset(&(their_addr.sin_zero), '\0', 8);
  ftime(&tp);
  if ((numbytes=sendto(sockfd1,buf,strlen(buf),0,(struct sockaddr *)&their_addr,
                                                    sizeof(struct sockaddr))) = = -1) {
    perror("sendto");
    exit(1);
  }
  delay_time=delay_probe();
  if ((numbytes=recvfrom(sockfd1, recvbuf, strlen(buf) , 0,
                  (struct sockaddr *)&their_addr, &addr_len)) = = -1) {
```

```
        perror("recvfrom");
        exit(1);
      }

      ftime(&tp1);
      printf("%%%time in s = %d\n",tp1.time-tp.time);
      close(sockfd1);
      if (delay_time>0){
        probe_time=12;
      }
      else probe_time = tp1.millitm-tp.millitm;
      return probe_time;
    }
}
void selectroute(char *addr, char *hostName){
      struct hostent *he;
   if ((he=gethostbyname(addr)) = = NULL) {  // get the host info
      perror("gethostbyname");
      exit(1);
    }
   if (strcmp(hostName,"sun114-21")= =0){
      if (strcmp(he->h_name,"128.227.205.18")= =0 && flag_sun114_21 = = 1){
        strcpy(str,"bark.cise.ufl.edu");
        printf("Entered selectroute-SUN114-21_1\n");
      }
      else if (strcmp(he->h_name,"128.227.205.18")= =0 && flag_sun114_21 = =2){
        strcpy(str,"leaf.cise.ufl.edu");
        printf("Entered selectroute-SUN114-21_2\n");
      }
      else strcpy(str,"bark.cise.ufl.edu");
    }
   if (strcmp(hostName,"bark")= =0){
      if (strcmp(he->h_name,"128.227.248.172")= =0 && flag_bark = = 2){
        strcpy(str,"rain.cise.ufl.edu");
        printf("Entered selectroute-BARK_2\n");
      }
      else if (strcmp(he->h_name,"128.227.205.136")= =0 && flag_bark = = 3){
        strcpy(str,"rain.cise.ufl.edu");
        printf("Entered selectroute-BARK_3\n");
      }
      else strcpy(str,"rain.cise.ufl.edu");
    }
   if (strcmp(hostName,"leaf")= =0){
      if (strcmp(he->h_name,"128.227.205.18")= =0 && flag_leaf = = 2){
        strcpy(str,"mango.cise.ufl.edu");
        printf("Entered selectroute-LEAF_2\n");
```

```
  }
  else if (strcmp(he->h_name,"128.227.248.172")= =0 && flag_leaf = = 3){
    strcpy(str,"mango.cise.ufl.edu");
    printf("Entered selectroute-LEAF_3\n");
  }
  else strcpy(str,"mango.cise.ufl.edu");
}
if (strcmp(hostName,"mango")= =0){
  if (strcmp(he->h_name,"128.227.248.58")= =0){
    strcpy(str,"rain.cise.ufl.edu");
    printf("Entered selectroute-MANGO_2\n");
  }
  else strcpy(str,"mango.cise.ufl.edu");
}
if (strcmp(he->h_name,"128.227.205.21")= =0){
  strcpy(str,"rain.cise.ufl.edu");
  printf("Entered selectroute-ECLIPSE\n");
}
else if (strcmp(he->h_name,"128.227.205.18")= =0){
  strcpy(str,"rain.cise.ufl.edu");
  system(sleep(1));
  printf("Entered selectroute-SAND\n");
}
else if (strcmp(he->h_name,"128.227.205.19")= =0){
  strcpy(str,"eclipse.cise.ufl.edu");
  printf("Entered selectroute-RAIN\n");
}
else if (strcmp(he->h_name,"128.227.248.58")= =0){
  strcpy(str,"bark.cise.ufl.edu");
  printf("Entered selectroute-LEAF\n");
}
else if (strcmp(he->h_name,"128.227.248.57")= =0){
  strcpy(str,"sand.cise.ufl.edu");
  printf("Entered selectroute-BARK\n");
}
else if (strcmp(he->h_name,"128.227.35.30")= =0){
  strcpy(str,"sand.cise.ufl.edu");
  printf("Entered selectroute-TICK\n");
}
else if (strcmp(he->h_name,"128.227.205.136")= =0){
  strcpy(str,"leaf.cise.ufl.edu");
  printf("Entered selectroute-MANGO\n");
}
else if (strcmp(he->h_name,"128.227.248.172")= =0){
  strcpy(str,"bark.cise.ufl.edu");
  printf("Entered selectroute-BOG\n");
```

```
  }

}

int main(void){
  int sockfd,t,count,route_change=0;
  struct sockaddr_in my_addr;    // my address information
  struct sockaddr_in their_addr; // connector's address information
  int addr_len, numbytes,totbytes;
  int buf[MAXBUFLEN];
  int * buf1;
  int flag=0;
  int probe_time;
  char hostName[20];
  int i,tmp;
  struct timeb tp1,tp2;

  i = gethostname(hostName,80);
  printf("HOSTNAME IS : %s \n",hostName);
  if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) = = -1) {
    perror("socket");
    exit(1);
  }
  buf1= (unsigned int *) malloc(200*sizeof(int));
  my_addr.sin_family = AF_INET;         // host byte order
  my_addr.sin_port = htons(MYPORT);     // short, network byte order
  my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
  memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
  if (bind(sockfd, (struct sockaddr *)&my_addr,
        sizeof(struct sockaddr)) = = -1) {
    perror("bind");
    exit(1);
  }
  addr_len = sizeof(struct sockaddr);
  while(1){
    if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
                    (struct sockaddr *)&their_addr, &addr_len)) = = -1) {
      perror("recvfrom");
      exit(1);
    }
    else{
      i = gethostname(hostName,80);
      selectroute(inet_ntoa(their_addr.sin_addr),hostName);
      count=t;
      count=count+1;
      probe_time=probe(hostName,flag);
```

```
        if (probe_time>11){
          ftime(&tp1);
          probe_time=probe(hostName,flag);
          if(probe_time>11){
            route_change =1;
            ftime(&tp2);
            printf("route_change ---%d \n",route_change);
            printf("time: %d \n",tp2.millitm-tp1.millitm) ;
          }
          else{
            route_change=0;
            ftime(&tp2);
            printf("route_change ---%d \n",route_change);
            printf("Time: %d \n",tp2.millitm-tp1.millitm) ;
          }
        }
        printf("time in ms = %d\n",probe_time);
        printf("route_change ---$$$%d \n",route_change);

        if (strcmp(hostName,"sun114-21")){
          flag=1;
          probe(hostName,flag);
        }
         if (route_change= =1) {
          flag_sun114_21 =2;
          flag_bark=3;
          flag_leaf=3;
        }
        else{
          flag_sun114_21=1;
          flag_bark =2;
          flag_leaf=2;
        }
        printf("got packet %d  from %s\n",count,inet_ntoa(their_addr.sin_addr));
        totbytes=totbytes+numbytes;
        t= count;
        sendData(buf,str,numbytes);
        flag=0;
      }
    }

    return 0;
}
```

LIST OF REFERENCES

[1] M. Baldi, and F. Risso, "Efficiency of packet voice with deterministic delay," in *IEEE Communications Magazine*, vol. 38, pp. 170–177, May 2000.

[2] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "RSVP: A new resource reservation protocol," in *IEEE Network,* vol. 7, pp. 8-18, Sept. 1993.

[3] L. Breslau, S. Jamin, and S. Shenker, "Comments on the performance of measurement-based admission control algorithms," in *Proc. IEEE Infocom 2000*, vol.3, pp. 1233–1242, Tel-Aviv, Israel, 2000.

[4] L. Breslau, E. Knightly, S. Shenker, I. Stoica, and H. Zhang; "Endpoint admission control: architectural issues and performance," in *Proc. ACM SIGCOMM 2000*, pp. 57–69, Stockholm, Sweden, 2000.

[5] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network," in *Proc. ACM Sigcomm 1992*, pp. 14–26, Baltimore, USA, 1992.

[6] M. Conlon, F. Cao, and H. Fang, "Performance analysis of measurement based call admission control on Voice Gateways," in *Proc. The 2nd IP-Telephony Workshop*, pp. 67–77, Columbia University, New York City, USA, 2001.

[7] S. Deng, "Traffic characteristics of packet voice," in *IEEE International Conference on Communications*, vol. 3, pp. 1369–1374, Seattle, USA, 1995.

[8] V. Elek, G. Karlsson and R. Ronngren, "Admission control based on end-to-end requirements," in *Proc. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 623–630, Tel Aviv, Israel, 2000

[9] P. Ferguson and G. Huston, *Quality of service: delivering QoS on the Internet and in corporate networks.* New York: John Wiley & Sons, Inc., 1998.

[10] K. Iida, K. Kawahara, T. Takine, and Y. Oie, "Performance evaluation of the architecture for end-to-end quality of service provisioning," in *IEEE Communications Magazine*, vol. 38, pp. 76–81, Apr. 2000.

[11] S. Jamin, P. Danzig, S. J. Shenker, and L. Zhang, "A measurement-based admission control algorithm for integrated services packet networks," in *IEEE/ACM Transactions on Networking,* vol. 25, pp. 56–70,1997.

[12] B. Keepence, "Quality of service for voice over IP," *IEEE Colloquium on Services Over the Internet-What Does Quality Cost*, pp. 4/1–4/4., London, UK, 1999.

[13] B. Li, M. Hamdi, D. Jiang, and X. Ciao, "QoS-enabled voice support in the next generation Internet: issues, existing approaches and challenges," in *IEEE Communications Magazine*, vol. 38, pp. 54–61, Apr. 2000.

[14] Y.J. Lin, J. Man, J. C. L. Liu, and H.A. Latchman, "Performance Study of VAT/RTP for supporting VOIP," in *Proc. The 6th World Conference on Systemics, Cybernetics and Informatics*, vol. 4, pp. 289–294, Orlando, USA, 2002.

[15] J. Pinto and K. J. Christensen "An algorithm for playout of packet voice based on adaptive adjustment of talkspurt silence periods," in *Proc. IEEE Conference on Local Computer Networks*, pp. 224–231, Lowell, USA, 1999.

[16] E. W. M. Wong, A. K. M. Chan, and T. P. Yum, "A taxonomy of rerouting in circuit-switched networks," in *IEEE Communications Magazine*, vol. 37, pp. 116–122, Nov. 1999.

[17] E. W. M. Wong, and A. K. M. Chan, "Analysis of rerouting in circuit switched networks," in *IEEE/ACM Transactions on Networking*, vol. 8, No. 3,pp. 419–427, 2000.

BIOGRAPHICAL SKETCH

Narasinha Kamat was born on 21st December 1978 in Mumbai, India. He graduated from Fr. Conceicao Rodrigues College of Engineering at the University of Mumbai with a Bachelor of Engineering in computer science in August 2000. He came to Gainesville, FL, to pursue higher studies in his chosen field. He graduated with a Master of Science in computer engineering from the Computer and Information Sciences and Engineering Department at the University of Florida in December 2002. His research interests are network technologies and Internet routing for real-time traffic.