Protocol Alchemy: Designing a TCP-Based Approach to Support Large-Scale Deployment of VoIP on the Internet

Amit Kumar Dutta , Mart Molle
{adutta,mart}@cs.ucr.edu
Department of Computer Science and Engineering
University of California, Riverside
CA 92521, USA

Trivikram Phatak
t.phatak@tcs.com
TATA Consultancy Services
1650 Iowa Ave. Ste#120
Riverside, CA 92521, USA

Abstract

None of the standard protocols in the TCP/IP suite provides the ideal type of service to support real-time voice transport over the public Internet. Since the development of a new protocol specifically tailored for this application would be very unrealistic, we propose an alternative strategy with minimal impact on the installed base. The starting point for our approach is TCP, the most commonly used Internet transport protocol. However, instead of changing the behavior of the protocol —in terms of when to transmit a (possibly duplicate copy of) some particular data segment—we change the mechansim for assigning payloads to those transmitted data segments. This approach allows us to escape from the negative effects of TCP's insistence on absolute reliability, while continuing to take advange of its compatibility with firewalls, and the preferential service given to TCP traffic at congested IP routers because of its responsiveness to congestion. For compatibility with existing VoIP applications, we develop a proxy-based architecture for managing long distance voice transport between LAN "islands". Voice traffic travels between proxies through TCP-based tunnels, which are optimized to support real-time traffic. In particular, the proxies aggregate data from unrelated voice sessions into each TCP segment to reduce overhead, allowing them to send redundant copies of the data in different TCP segments to counteract packet loss and/or delay jitter. To demonstrate the effectiveness and performance benefits of our approach, we present experimental results from both a detailed simulation model and a prototype Linux implementation.

1 Introduction

TCP has become the de facto standard protocol for data transport over the public Internet. TCP was originally designed to provide applications with a reliable transport mechanism on top of an unreliable IP-based network infrastructure. However, TCP has evolved to include sophisticated congestion control algorithms that respond to the active queue management (AQM) algorithms in IP routers (such as RED [10]). These algorithms promote effi cient sharing of Internet resources among competing applications when congestion is present, and adapt to give high throughput by an individual application when it is not. Thus, TCP flows are classifi ed as *responsive traffic* by AQM algorithms because they respond to congestion signals by reducing their transmission rate. As a result of the "good citizen" behavior, *TCP flows may receive preferential treatment at a congested router* in comparison to non-responsive traffi c like UDP.

TCP also has a signifi cant advantage over UDP as tighter network security measures are deployed. This is because many fi rewalls block all non-TCP traffi c [14]. Since TCP is connection-oriented, the fi rewall can make a policy determination about a particular flow once at setup time, and then cache the result for duration of the flow. On the other hand, UDP is a connectionless protocol so there is no setup to inspect and no way to determine whether later packets carrying similar address/port numbers were generated by the same application.

Now consider the type of service requirements for a real-time multimedia application such as Voice-over-IP (VoIP). VoIP is a method for transporting real-time voice over IP networks [25]. In VoIP, the sending application converts the voice signals into a compressed digital format, for transmisssion over the network as a packet stream. The receiving application converts this incoming packet stream back into normal voice format for real time playback. VoIP works well with a "best effort" data transport service, in which a high proportion of its data is delivered in a timely fashion, but the remaining data can be lost with little impact on the quality of the voice conversation. In addtion, the VoIP application has several adjustable parameters available (such as voice encoding rate, packetization delay, playback buffer size, etc.), throught which it can make trade offs between voice quality and traffi c requirements in response to different network conditions.

Studies have shown that the voice quality degrades if the elapsed time between speaker and the listener becomes too large, leading to such problems as talker overlap and echo. Delays below 150 ms are considered good, while delays greater than 400 ms are considered unacceptable [3, 5, 19]. Most receivers have a playback buffer to compensate the delay and jitter [20]. Similarly, packet loss rates greater than 10% are generally not tolerable, although interpolation and Forward Error Correction (FEC) mechanisms can be used to compensate for some losses [12].

Given this background information, which protocol from the Internet suite should the VoIP application writer choose for transporting voice streams? Although TCP receives preferential service at congested routers, its insistence on absolute reliability represents a very serious problem. Thus, most VoIP applications currently run RTP over UDP to provide end-to-end delivery services [12, 21]. In this paper, however, we consider a new answer to this dilemma: developing a strategy for solving the negative aspects of TCP for VoIP traffi c without changing its observable behavior on the network. The key idea is to change the rules by which the payload is assigned to each data segment. These changes reduce overhead (i.e., aggregating traffi c across multiple sessions to create "car pools"), mask the effects of lost and/or delayed segments (i.e., speculatively sending redundancy copies of data without waiting for an acknowledgement), and turn previously-useless TCP retransmissions into gold by replacing the worthless repetition of expired data by a new payload.

2 Related Work

Liang et al. [14] propose some modifi cations to TCP to make it more suitable for real time applications. Their approach, known as TCP-RTM, allows the receiving application to read out-of-order packets rather than waiting for lost data to be retransmitted. In addition, the receiver is allowed to tell the sender to move forward without bothering to retransmit obsolete missing data by advancing its advertised receive window in conjuncion with a selective negative acknowledgement (SNACK) mechanism which is used by the sender's congestion control to trigger a rate decrease.

Badrinath et al. [6] propose a mechanism for aggregating small packets to reduce router congestion and improve TCP throughput. They study the pros and cons of aggregating small-sized TCP packets.

Lee et al. [13] describe a system of TCP tunnels to carry multimedia traffi c between endpoints. Each tunnel consists of a single long-lived, high-bandwidth TCP session between proxies. Their main idea is to encapsulate each UDP packet into a separate TCP packet belonging to the tunnel session, which allows the proxies to set the priority for each flow.

Bolot et al. [9] address the problem of optimizing Forward Error Correction (FEC) parameters to provide the receiver with the best quality at any given time by sending redundant copies of the data, subject to varying network conditions and rate constraints imposed by congestion control algorithms.

3 Overview of our Approach

3.1 Aggregate payload from multiple sources

Carrying more data in each packet increases effi ciency and improves performance. A typical packet generated by a VoIP application contains no more than $D = 80 - 160$ bytes of payload.[1] But consider the overhead associated with sending this payload as a single Ethernet frame, which adds: *20 bytes* of framing overhead between packets to account for the minimum-size 96-bit Inter-Frame Gap and 64-bit preamble/start-frame delimiter; *18 bytes* of packet overhead, to account for a minimum-size 14-byte Ethernet header and 32-bit CRC; *20 bytes* of packet overhead, to account for a minimum-size IPv4 header; *8 bytes* of packet overhead, to account for the UDP header; and *12 bytes* of packet overhead, to account for a minimum-size RTP header. Thus, transmitting D bytes of voice payload occupies the physical link for enough time to transmit $D + 78$ bytes. Depending on the effi ciency of the encoder, this represents a combined overhead of between 30% (if $D = 160$ bytes) and 80% (if $D = 16$ bytes).

Clearly, network effi ciency would greatly increase if we could increase the amount of payload carried by each packet! Moreover, if we increase the payload size by a factor of k, then we also reduce the total number of packets sent by a similar amount. As discussed in [7], changing the traffi c mix to include $1/k$th as many voice packets, each carrying k times more payload has several additional benefi ts beyond the overhead reduction.

[1] For example, if the voice application uses a simple uncompressed 64Kbps PCM encoder, it will generate 160 bytes of payload every 20 ms. However, much lower rate encoders are available [2], such as ITU-T recommendations G.726, G.728 and G.729, which can reduce the payload size by an order of magnitude.

IP routers benefit from this payload-size increase in two ways. First, larger packets reduce the processing load at the routers. This is because all packets incur similar processing requirements at the router independent of their size, i.e., a routing table lookup and possibly its classification according to some set of access control and/or priority based policy rules. However, a larger packet size increases the minimum time between packet arrivals, giving the router more time to process each packet. Second, larger packet sizes can reduce packet losses at the routers because the architecture of many routers places a limit on *the total number of queued packets*, rather than the *the combined size of all queued packets*. Thus, by combining the payloads from k small packets into one large packet we free up $k-1$ slots in the router queue and hence reduce the probability of queue overflow.[2]

The source and destination nodes would also benefit from reduced operating system overhead if the increase in payload size were carried out by the VoIP application programs. On a modern RISC processor, each interrupt for transmitting or receiving a packet entails considerable overhead.

Unfortunately, increasing the payload size would also have some serious negative side effects *if the additional data was obtained from the same voice call*. First, the *packetization delay* would need to increase by a factor of k. In other words, if the voice application originally collected data from the voice encoder for 20 ms (say) before transmitting a packet, then we would now need to collect data from the voice encoder for $20k$ ms before we have accumulated enough data to fill the payload of the larger packet. Clearly, the first piece of voice data added to the packet must wait at the sender for $20k$ ms before the final piece of voice data arrives. Thus, in comparison to the original application, this change increases the delay at the sender by an additional $20(k-1)$ ms, and hence reduce the likelihood that the data will be able to reach the destination before its scheduled playback time. Second, in the event of a single *lost packet*, the receiving application will now be faced with a gap in the voice stream that is k times larger than before. Thus, even if it is easy for the receiving application to conceal a small loss of data, we are now asking it to handle the equivalent of k consecutive packet losses, which will surely have a larger effect on voice quality.

In the case of our proxy-based architecture however, we have the option of aggregating data "in parallel" (i.e., generated by different voice sessions at the same time) rather than "in series" (i.e., generated by the same voice session at different times). For example, suppose we combine a single voice payload (of size D bytes) from k different voice sessions to create an heterogeneous aggregate payload. Clearly, we will incur some additional overhead in comparison to the "series" approach because we must be able to demultiplex the k voice streams at the receiving proxy. However, the "parallel" approach imposes no increase in the packetization delay because each

[2]For example, Cisco routers[4] allocate memory from a structure that reserves a fixed-size block for each packet. For efficiency, routers store packet headers (which must be examined during the routing/classification process, and whose sizes are independent of packet sizes) in fast memory, whereas the packet payload is stored in ordinary slow memory. Thus, the amount of expensive fast memory available determines the maximum queue size.

of the k streams is generating new data at the same. Furthermore, in the event of a single packet loss, each of the k voice streams would suffer an easily concealed small losses of data rather than requiring one voice stream to handle a much larger loss of data.[3]

3.2 Retransmit without waiting for ACKs

Because payload aggregation signifi cantly reduces the cost of transporting voice over the public Internet (in terms of both the total number of packets sent and the normalized overhead per payload byte), we can afford to invest some of this saved bandwidth in sending redundant data to improve quality of service(QoS). Rather than introducing some new computationally expensive, non-standard voice encoding scheme that supports forward error correction, we propose a much-simpler approach that is based on transmitting duplicate copies of the existing voice data in different packets.

The key idea is that the sending proxy will transmit duplicate copies of the voice data *speculatively*, without waiting to hear about the fate of the original copy from the receiving proxy through some feedback mechanism that arrives at least one round-trip time (RTT) later. This allows the receiver to recover from a lost packet in less than one RTT.

If typical network transit delays are quite close to the tolerable delay for a VoIP session, then any acknowledgement-based retransmission mechanism would be too slow to be useful. However, the sending proxy can still base its duplicate scheduling decisions on general information about current conditions along the path (i.e., how many recent packets have been lost or suffered excessive network transit delays). Thus, if packet loss rates rise, the sending proxy can compensate by adjusting the number of duplicate copies sent for each future voice sample.

3.3 TCP, rather than UDP, for voice transport

Despite its overall popularity, however, TCP is generally not used for multimedia applications because it is optimized to provide *reliable delivery* of *all* data, rather than *timely delivery* of *sufficient* data. Real time voice applications are more tolerant of some lost data (which can be hidden by various interpolation schemes) rather than high jitter. Moreover, data that TCP retransmits after detecting lost packets is likely to miss the deadline for using it at the destination. Meanwhile, new data being generated by the application is held at the sender until the previous data as been delivered, causing it to miss its playback deadline as well.

The dynamic behavior of TCP's congestion control mechanism [11] presents a further challenge to the timely delivery of real-time voice. In contrast to the predictable *bandwidth requirements* of a voice encoder, which

[3]This is exactly the same risk-spreading principle that forms the basis of the insurance industry.

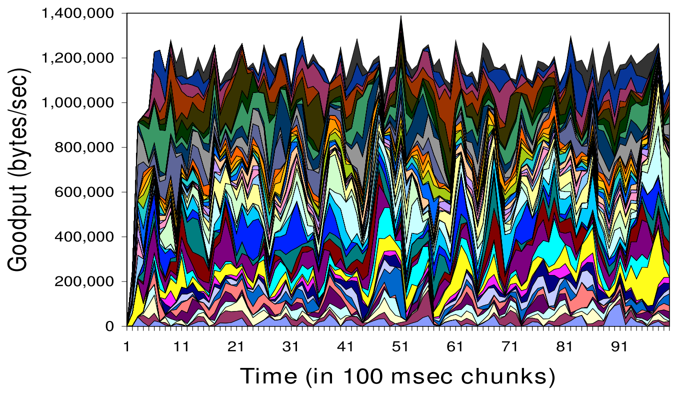

Figure 1: Dynamic bandwidth sharing by 40 TCP sessions over a common 10 Mbps bottleneck link

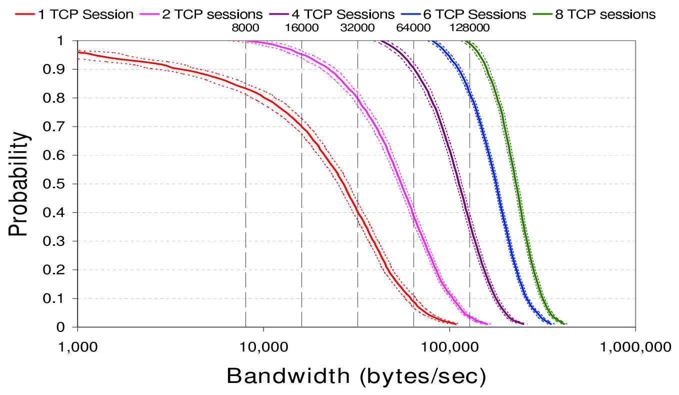

Figure 2: Complementary cumulative distribution functions for the combined bandwidth of 1, 2, 4, 6, and 8 parallel TCP sessions (solid curved lines) together with the associated 95% confidence intervals (dashed curved lines). Also shown (dashed vertical lines) are the bandwidth requirements for 1, 2, 4, 8, and 16 simultaneous 64Kbps voice conversations.

periodically generates a quantum of data with (approximately) fixed size, the *instantaneous bandwidth available* from a single TCP stream varies considerably.

Figure 1 shows a simulation experiment representing the first 10 seconds of a “bulk file transfer” benchmark. Here 40 simultaneous TCP flows are all passing though the same 10 Mbps bottleneck link, and we record the average goodput received by each flow over consecutive 100 ms intervals. The results are shown as a “stacked area” chart, so thickness of each color band represents the throughput for a particular flow, and their total height represents the aggregate goodput for all flows. Thus, in the ideal case, the graph would consist of 40 parallel bands, each with a thickness of about 30,000 bytes/s. However, the results are far from ideal!

The first 5 time steps clearly show the exponential increase in total goodput which is characteristic of TCP’s *slow start phase*. Slow start restricts the available bandwidth to the application during the initial period after establishment of a new TCP connection, or during the recovery period after the connection has stalled because of a timeout. It is important to note that this initial bandwidth restriction is a key feature of the TCP congestion control algorithm; it will be present even if we increase the size of TCP receiver's advertised window, the link speeds, or the available buffer space in the routers. Only decreasing the Round-Trip Time (RTT) can reduce this effect.

The remaining 95 time steps show the behavior of TCP during its *congestion avoidance phase*, where the available bandwidth for each TCP flow varies chaotically in response to packet losses (or other congestion signals). During this time, each TCP sender adjusts its congestion window according to the linear increase (if there are no packet losses) and multiplicative decrease (if a packet loss is detected) algorithm. Because of this dynamic window adjustment —and in particular, the large bandwidth reduction in response to packet losses —a TCP connection in "steady state" will experience continuous changes in the available bandwidth, quite possibly including periods of time where the rate falls below the modest requirements of the voice application. This result is clearly evident from the left-most curved line in Figure 2, which shows the complementary cumulative distribution function (CCDF) for bandwidth provided by one TCP session. Even though the average bandwidth is close to the predicted value of 30,000 bytes/s, its distribution is so highly skewed that only about 83% of the time the average bandwidth over a 100 ms interval is suffi cient to handle one uncompressed voice session (which requires only 64Kbps or 8,000 bytes/s). Moreover, even if we drastically reduced the bandwidth requirements to only 1,000 bytes/s, that single TCP session would still only provide suffi cient bandwidth for about 96% of the time.

Nevertheless, we will now demonstrate that these diffi culties for real time applications caused by TCP's insistence on providing reliability above all and its chaotically varying bandwidth can be easily solved, using an innovative combination of techniques that we will introduce in the following sections. Thus, in the fi nal analysis TCP easily triumphs over UDP because of its compatibility with fi rewalls and better treatment by routers.

3.4 Session diversity, but not path diversity

In addition to the CCDF for the bandwidth provided by one TCP session, Figure 2 also shows the CCDF for the aggregate bandwidth available to a single application if it used 2, 4, 6, or 8 TCP sessions in parallel. Notice that the aggregate bandwidth availability improves dramatically as we aggregate multiple TCP sessions in parallel — and all of those sessions go through the same bottleneck link. In particular, 2 parallel TCP sessions have suffi cient combined bandwidth to handle one uncompressed 64Kbps voice session virtually 100% of the time. Moreover, for the same 96% availability level at which one TCP session could only support an application data rate of about 1,000 bytes/s., two parallel TCP sessions can support an application data rate of about 16,000 bytes/s.

The bandwidth availability distribution continues its dramatic improvement as we aggregate more and more parallel TCP sessions to support a single application. For example, 8 parallel TCP sessions have suffi cient combined bandwidth to handle 16 uncompressed 64Kbps voice sessions virtually 100% of the time. The lesson here is that even though there are severe performance problems when we attempt to support one voice call using one TCP session, we can get very good performance under the same network conditions when we combine several TCP sessions to support multiple voice calls. This observation fits very well with our proxy-based architecture, which is designed to support multiple simultaneous voice calls between LAN “islands” of VoIP connectivity over the public Internet.

The next step beyond distributing traffic across multiple TCP sessions is *path diversity*, where each of these sessions follow a different path through the Internet. Although path diversity has been suggested as a means for improving QoS for real time traffic [15], it is much more difficult to establish than session diversity, and both of two major strategies for implementing this feature have significant drawbacks. Invoking the IP source routing option to force packets to follow different routes would impose considerable processing overhead on the proxies (which must maintain detailed routing tables) as well as the intermediate routers. In addition, IP source routing represents a known security vulnerability because a malicious host can use it to spoof its return address while establishing a TCP connection. Similar problems would result if the proxies tried to set up indirect paths in which the traffic is relayed by a third-party device in some remote location.

Fortunately, careful studies of Internet traffic [17] indicate that most links are quite lightly loaded, although a small fraction of the links are very heavily loaded. These “hot spots” tend to occur at critical choke points in the Internet topology, such as expensive trans-continental links, or the peering points between major Internet Service Providers. Thus, it seems unlikely that any of our alternate paths can avoid every hot spot, while at the same time their relative rarity makes it unlikely that a given path will encounter multiple hot spots. Thus we expect the most common case will be that each long distance path between proxies contains only one major bottleneck, whether or not we try to use path diversity. This is exactly the situation we considered previously in Figures 1 and 2.

3.5 Use TCP as a container shipping service

We have now reached the key idea in our approach, namely that the TCP sequence number is *not relevant* to the reliable delivery of application data. These sequence numbers are inspected by the TCP congestion control algorithm in the usual away, so our system will properly respond to packet losses (or lack thereof) by lowering (or raising) its transmission rate. However, acknowledgement of payload delivery is handled by a separate payload acknowledgement scheme (described below), which understands that each payload consists of data from multiple voice which was generated at different times. Moreover, the payload acknowledgement scheme also recognizes that when a particular TCP segment is lost, another segment with the same TCP sequence number will be transmitted but its payload will be replaced by something new.

4 Detailed Proxy Architecture

Initialization. When a connection between two proxies is first established, they open up a fixed number of TCP sessions with each other and exchange a series of training packets that contain no voice data. Instead, these training packets carry timestamps in their payload field, to allow the proxies to determine their relative clock offsets, and to estimate the one-way delay in both directions. (See below for more details.) The exchange of training packets also invokes the TCP slow start algorithm to increase the congestion window in anticipation of future bandwidth needs.

Payload assembly and distribution. The proxy system is completely transparent to the local VoIP clients it serves, except for the requirement to route their voice traffic (encapsulated in RTP/UDP/IP as usual) via the proxy, rather than sending directly to the destination VoIP client. Incoming packets from local VoIP clients are received by the **UDP Listener** process, which strips off the unnecessary headers (leaving only an RTP packet, together with the relevant identification information, such as the *source address and length*) and places it in a separate queue for each VoIP client. Thereafter, the RTP packet waits for the **Queue Reader** process to assign it to the payload of some TCP container(s). Whenever there are any TCP containers available and waiting RTP packets, the Queue Reader starts collecting RTP packets to form a TCP payload that multiplexes data from the different VoIP source queues according to a Deficit Round Robin Scheme [23] that guarantees fair bandwidth share to all VoIP clients. A fraction f of the payload is reserved for fresh data which has never been included in a previous payload. Once this quota is filled, or as soon as the oldest data collected so far is reaching its maximum age limit, the Queue Reader tries to fill up the remaining payload with redundant data —which has been previously sent fewer than r times, and no acknowledgement has yet been received. The scheduling of redundant data is based on the approach of Bolot et al. [9], who showed that to maximize the probability that at least one copy of the data reaches the destination before a maximum playback delay deadline expires (typically, we use $D^* = 250ms$), the transmission of these r copies should be equally spaced over the remaining slack in the delivery schedule, i.e., D^* minus the sum of the age of the data when it arrived at proxy and the estimated one-way network transit delay.

TCP container selection. The Queue Reader process uses a greedy algorithm for choosing an available TCP container to carry its newly-constructed payload. Each open TCP session maintains a list of its available containers —i.e., the block of consecutive segment IDs that is bounded below by the last cumulative ACK received and above by the minimum of the receiver's advertised window and the sender's congestion window —from which we subtract any segment IDs that have already been sent but not returned to the pool because of an assumed packet loss (as indicated by either the expiry of a timeout or the reception of multiple duplicate ACKs through TCP's fast retry algorithm). Note that the list entries for "used" segment IDs do *not* store any information about the associated

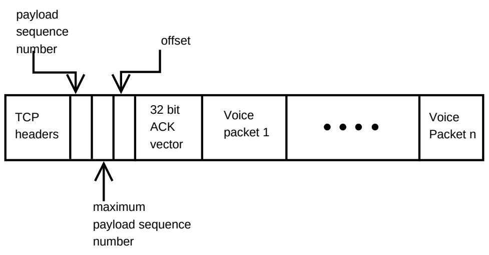

Figure 3: Payload structure inside a TCP container

voice payload, which would be ambiguous if the TCP segment has been declared lost and its segment ID returned to the pool.

Since its congestion window increases whenever a TCP flow receives an acknowledgement, our strategy is to grow the supply of available containers in the future. Thus, the first choice is to select the container from a TCP flow that is currently in the slow start (i.e., exponential increase) phase, because its acknowledgement will increase the number of available containers by one. The second choice is to select the container from a TCP flow that is currently in the congestion avoidance (i.e., linear increase) phase and has the *smallest* congestion window size, CW_{MIN}, because its acknowledgement will increase the number of available containers by the largest fraction, i.e., $1/CW_{MIN}$. Finally, the last choice is to select the container from a TCP flow for which the receiver's advertised window is less than or equal to the sender's congestion window, because its acknowledgement will not increase the available number of containers.

Payload acknowledgement scheme. Since the payload that is assigned to a given TCP segment ID will be changed if the TCP session decides that the segment was lost and needs to be retransmitted, we need to establish a separate acknowledgement mechanism to reliably keep track of payloads. Since the selection of the TCP session and segment ID to be used for transmitting a particular payload is rather arbitrary, we will establish a single proxy-level acknowledgement scheme that spans all TCP sessions, using meta-data carried within the payload field of the TCP containers, as shown in Figure 3. Meanwhile, the standard TCP segment IDs are still present and continue to be used the normal way (i.e., if a TCP segment with ID x is lost, the sender will transmit another TCP segment with the same ID), but in reality their only purpose is to allow the TCP congestion control algorithms to respond normally to congestion signaling from the routers.

The payload acknowledgement header consists of four fields: (i) the 32-bit *payload sequence number*, which identifies the payload being sent in this container; (ii) the 32-bit *maximum payload sequence number*, which indicates the highest payload sequence number received so far by this proxy; (iii) the 8-bit *offset*, which is used to selectively acknowledge out-of-order segment arrivals by indicating the relative position of the most recently received payload number in comparison to the highest payload number received; and (iv) a 32-bit Boolean *ACK-*

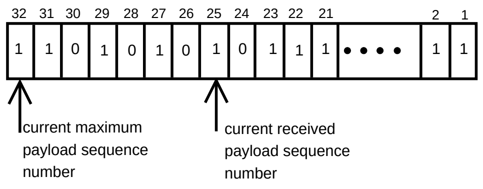
32 31 30 29 28 27 26 25 24 23 22 21 2 1
1 1 0 1 0 1 0 1 0 1 1 1 • • • • 1 1
current maximum payload sequence number
current received payload sequence number

Figure 4: Detailed structure of the ACK vector

vector that indicates which of the next 32 highest payload sequence numbers have been received by this proxy. Remember that these payload sequence numbers are common across all the TCP sessions, so a payload transmitted by a TCP container belonging to one session will be included in subsequent acknowledgements carried by TCP containers belonging to all TCP sessions.

While the Queue Reader is assembling each voice payload, it builds a record of the contents of that payload indexed by the associated payload sequence number, including the *client_id* and *sequence number* for all the voice packets encapsulated within it. If the proxy later receives an acknowledgement for this payload sequence number, it cancels any remaining duplicate transmissions for the associated voice packets and then discards the record. Otherwise, it simply discards the record if an acknowledgement is not received before the normal TCP retransmission timeout period expires.

The structure of the ACK vector is shown in Figure 4. The bit position 32 represents the maximum payload sequence number received so far, and hence is always 1. The remaining bits signify whether each of the next-highest 31 sequence numbers have also been received. The offset field is used to provide an explicit acknowledgement for the arrival of an out-of-order packet. In this example, the sequence number of the most recently received payload is below the maximum by 7, so bit position 25 is also set to 1. Conversely, if the sequence number of the current payload is greater than the maximum payload sequence number received so far, then the maximum is increased to the current received sequence number, and the rest of the ACK vector is right-shifted by the magnitude of this difference.

5 Finding one-way network delay

Paxson [18] showed that the Internet packet losses are usually uncorrelated between the forward and the reverse path. Similarly, we expect the one-way network transit delays in each direction to be different as well. Since this information is an important parameter for scheduling decisions made by the Queue Reader process, we would like a better estimate of the one-way delay than simply half the RTT.

Our approach is similar to the one used in the Network Time Protocol [16]. Initially, we assume that Alice and Bob have not synchronized their respective local clocks, and that Bob's clock reads Δ time units ahead of Alice's clock. Thus, the first step is to derive an estimate for the unknown clock offset value Δ. Alice sends a series of training messages $A_1,\ A_2,\ldots$ to Bob, each of which carries a local timestamp, $send(A_i)$, to indicate its departure time from Alice based on her local clock. When these messages reach Bob, he adds his own local timestamp, $recv(A_i)$, to indicate its arrival time based on his local clock. Similarly, Bob sends a series of training messages $B_1,\ B_2,\ldots$ to Alice, which receive Bob's local departure timestamp, $send(B_j)$, and later Alice's local arrival timestamp, $recv(B_j)$.

After defining the average of the two local clocks to be the "correct" time, we see that each training message gives us an estimate for the one-way network transit delay in terms of the (unknown) clock offset, Δ, namely $\tau_{AB}(i) = recv(A_i) - send(A_i) - \Delta$ from the ith training message sent from Alice to Bob and $\tau_{BA}(j) = recv(B_j) - send(B_j) + \Delta$ from the jth training message sent from Bob to Alice. These delay estimates can vary considerably between packets because of queueing effects. However, Δ is assumed to be constant and the majority of Internet paths are symmetrical, so the *minimum possible* one-way network transit delays in each direction are probably equal too. Notice that this common value for the minimum one-way network transit delay may be obtained from network delay measurements covering either direction. Moreover, even though both expressions still contain the unknown constant Δ, we can equate them to obtain a simple estimate for the clock offset:

$$\Delta \approx (min_i\{recv(A_i) - send(A_i)\} - min_j\{recv(B_j) - send(B_j)\})/2 \tag{1}$$

Once we have found Δ, it is easy for Alice (and, similarly, Bob) to maintain an exponential moving average calculations for the one-way network transit delay to Bob by carrying out the following updates after receiving delay information about the ith packet:

$$\begin{aligned} est(\tau_{AB}) &\approx 0.875 \cdot est(\tau_{AB}) + 0.125 \cdot \tau_{AB}(i) \\ dev(\tau_{AB}) &\approx 0.75 \cdot dev(\tau_{AB}) + 0.25 \cdot |\tau_{AB}(i) - est(\tau_{AB})| \\ timeout &= est(\tau_{AB}) + 2 \cdot dev(\tau_{AB}) \end{aligned}$$

As shown in Figure 5 timeout calculation based on the one-way delay estimation gives the most conservative upper bound on the actual delay suffered by the ongoing packet. The similar calculations based on RTT and RTT/2 do not give a correct estimate of the uni-directional delay.

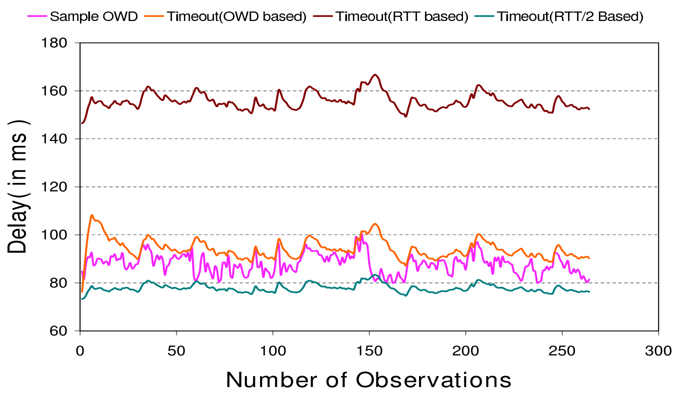

Figure 5: Timeouts based on one-way delay vs. RTT

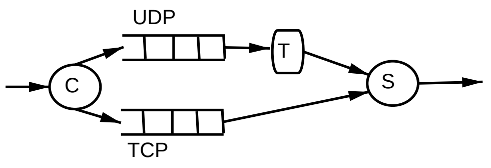

Figure 6: Virtual Router Implementation

6 Simulation Results

We developed a prototype simulation using CSIM simulation package [22]. Multiple *VoIP Clients* send voice packets to their destination, routed via the proxy servers. Different encodings were used for each client resulting in different sized payloads. Each client sends data at a fi xed gap of 20 ms. The maximum playback delay was set to 250 ms.

The router was implemented as described in [8] where the queueing components are designed to favor congestion-avoiding TCP flows over aggressive UDP traffi c. As shown in Figure 6, classifi er C inspects each incoming packet's protocol ID to separate TCP and UDP traffi c. The token bucket fi lter T causes the UDP queue to drop packets when a certain rate threshold is exceeded. The scheduler S reads packets from the TCP queue directly and from the UDP queue via the token bucket fi lter. Hence TCP flows get the full bandwidth in absence of UDP traffi c but UDP flows are always constrained to a certain quota. Study of Cisco router policies [1] also reflect the preferential treatment of responsive flows over non-responsive flows.

We compared the simulation results using three different schemes. In the **Simple UDP scheme** the voice packets were transmitted as simple UDP packets with no aggregation or redundancy. In the **Redundant UDP scheme** multiple UDP voice packets were combined together into single bigger UDP packets, stripping off unneeded headers coupled with duplicate copy transmissions. But no congestion control strategies were in place and the intermediate routers implementing a rate fi lter for UDP streams dropped packets once the specifi ed limit was

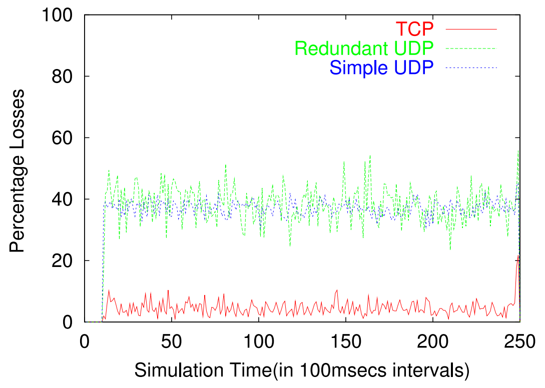

Figure 7: Percentage Losses per 100ms intervals using 20 VoIP sessions and 15 TCP sessions

reached. Absence of any acknowledgement scheme resulted in a constant number of duplicate copy transmissions. In the **TCP scheme** aggregation and redundancy was implemented using TCP as outlined in our earlier sections.

The simulations were run for different time periods with different network capacities. Experiments were conducted with a varying number of incoming VoIP sources and TCP sessions. Other than the VoIP UDP sources and shared TCP sessions, there were extra TCP and UDP sessions sending packets through the same bottleneck link to simulate real network traffi c (10 TCP sessions each between 3 host pairs doing bulk fi le transfer in 1Kbyte chunks and 10 UDP sessions between a single host pair sending 1Kbyte packets every 5 ms). The proxies were separated by 15 hops and the link speeds were typically 28-30 Mbps with the bottleneck link having a speed of 20 Mbps. The propagation delay was around 5-6 ms. At most 2 duplicate copies were sent for each voice packet and each TCP packet had at most 55% of it fi lled with fresh data.

Figure 7 shows the percentage losses per 100 ms interval of the simulation time. The results include both physically lost packets and late packets that missed their scheduled playback time. We see that both UDP schemes have high loss rates (about 40%), whereas the Redundant TCP scheme gives us a drastic reduction to about 4-5%. During the initial 1000ms, the training messages were being exchanged. After that the VoIP packet transfer began. Soon the UDP sources consumed their bandwidth limit leading to UDP packet drops. Even with Redundant UDP scheme, the results were not good since increased packet length led to longer queuing delays. Even duplicate transmissions were not suffi cient for ensuring timely delivery in moments of severe congestion.

Since consecutive packet losses are harder for the voice application to hide than isolated packet losses, Figures 8 plot the number of received packets in a 3-packet moving window for each of the different schemes. A y-value of 3 means that all three packets in the window reached the destination on time, whereas y-values of 2, 1 or 0 mean that some or all of the packets in that window were lost. We observe that in our Redundant TCP mechanism, packet losses of any kind were quite rare, and there were only two double packet losses during the entire experiment.

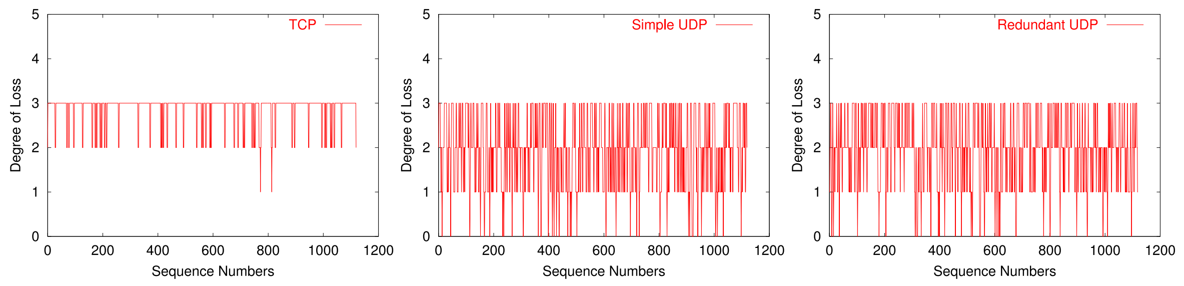

Figure 8: Loss Characteristics using the three schemes

Conversely, both single and double packet losses were quite frequent in both of the UDP mechanisms, and even triple packet loss events occurred dozens of times during the experiment. It is evident that the number and the burstiness of losses is much more in the UDP schemes. Bursty losses are more signifi cant than single losses because they are more likely to degrade the user's playback quality.

A comparison was made of the goodput achieved by increasing the number of TCP sessions for a given number (twenty) of VoIP sessions. The goodput in this experiment is a measure of the ratio of VoIP packets generated by all the sources and the VoIP packets successfully delivered in time. With around 8 TCP sessions, it was 85% with a gradual increase to around 98% with 10 sessions and thereafter remaining constant, whereas both the UDP schemes achieved around 68%. We also noted the number of duplicate copies needed for the Redundant UDP scheme to achieve a goodput as high as that of the TCP scheme – it was almost thrice the current count, i.e. approximately 6 duplicate transmissions.

7 Linux Testbed

We implemented the proposed architecture on a Linux testbed(kernel version 2.4.8-26 Mandrake Linux with TCP-Reno SACK enabled) to conduct real life experiments. The proxies were implemented as multithreaded applications. Multiple threads at the VoIP proxies served the queuing, dequeuing, multiplexing and demultiplexing of voice packets.

The Linux kernel TCP implementation maintains two receiver queues - *receive queue* and *out-of-order queue*. As the names suggest they store in-order and out-of-order packets respectively. Implementing the TCP container service abstraction required a signifi cant amount of code change to the *retransmission queue* implementation. Hence we chose a simpler approach by testing out our concepts using TCP-RTM[14].

The only changes to the Linux kernel were: (i) A system call implementation which on being passed a set of TCP session socket descriptors returned all their details. The application program then used this data to calculate

Table 1: Comparison of Simple UDP, Redundant UDP and TCP approaches

Scheme	Num Late	Jitter Buffer Size
S UDP	177	255ms
R UDP	52	202ms
TCP	3	180ms

the container availability and select one of the sessions. This required around 15 lines of code; (ii) The TCP-RTM implementation . Here if the application wanted to read more data and the *receive queue* was empty, then packets from *out-of-order queue* were passed to them and the *receive next* pointer was advanced to reflect the out-of-order data read. Also immediately two ACKs were sent to the sender with the SACK fi elds reflecting the out-of-order packets read. In this way unnecessary retransmissions were avoided and also the sender was informed of the losses . Their implementation was about 50 lines of code change to the TCP stack implementation.

We used the OpenH323 OhPhone application[24] to conduct a voice conversation experiment across two labs separated by 22 hops(as per *traceroute*) with a RTT delay on an average 50ms(as per *ping*) and separated by a 100Mbps link. The experiment was done on a business day at around 10:30am. All the voice traffi c was routed through source and destination proxies. The conversation was for about 5 minutes. We experimented all the three schemes in turn. Table 1 summarizes some of the results. Along with the jitter buffer size, the number of delayed packets reduced signifi cantly with the use of TCP scheme. In absence of severe congestion, the redundant UDP scheme performed a little better. But the increased delay inspite of using the same also similar sized packets and same number of duplicate transmissions(two) as that in the TCP scheme suggest the preferential treatment of TCP flows by routers. All these delayed packets were eventually discarded by the receiver playback buffer leading to intermittent gaps in speech.

8 Conclusions and Future Work

In this work, we propose a novel approach for real time voice transmission over the public Internet using TCP. Detailed simulation results and preliminary experiments over the Internet show that this scheme achieves much better quality and loss characteristics (while being a *responsive and well-behaved flow*) than the conventional RTP over UDP approach.

Future work includes (i) further experiments over the public Internet using trans-continental links; (ii) implementation and testing of our *TCP container service* notion; (iii) predicting number of duplicate transmissions needed based on Data Link layer characteristics; (iv) design issues with multiple proxies where acknowledgment and manageability of multiple TCP sessions become more complicated ; (v) dynamic adaptation to network conditions to open/close more number of shared TCP sessions; (vi) elimination of unneeded headers to further augment aggregation efficiency.

References

[1] Congestion Avoidance Overview, Cisco IOS Quality of Service Solutions Configuration Guide. www.cisco.com/univercd/dc/td/doc/product/software/ios122/122cgcr/fqvs-c/fqcprt3/qcfconav.pdf.

[2] G.7xx:Audio (Voice) Compression Protocols (CODEC) Overview. http://www.javvin.com/protocolG7xx.html.

[3] Reference removed for double blind reviewing.

[4] Understanding Buffers Misses and Failures. http://www.cisco.com/warp/public/650/41.html.

[5] Understanding Delay in Packet Voice Networks, Cisco Whitepaper. http://www.cisco.com/warp/customer/-788/voip/delay-details.html.

[6] B. Badrinath and P. Sudame. Car Pooling on the Net: Performance and Implications. *ACM Sigcomm, New Research Session*, September 1999.

[7] B. Badrinath and P. Sudame. Gathercast: The Design and Implementation of a Programmable Aggregation Mechanism for the Internet. *ICCCN*, October 2000.

[8] F. Baumgartner, T. Braun, and B. Bhargava. Virtual Routers: A Tool for Emulating IP Routers. *LCN*, November 2002.

[9] J. Bolot, S. Fosse-Parisis, and D. Towsley. Adaptive FEC-Based Error Control for Internet Telephony. *Proceedings of IEEE Infocom*, 1999.

[10] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[11] V. Jacobson and M. Karels. Congestion Avoidance and Control. *ACM Sigcomm*, August 1988.

[12] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach featuring the Internet.* Addison Wesley Longman, 1999.

[13] B. Lee, R. Balan, L. Jacob, W. Seah, and A. Ananda. Avoiding congestion collapse on the internet using TCP Tunnels. *Computer Networks*, 39(5):207–219, June 2002.

[14] S. Liang and D. Cheriton. TCP-RTM: Using TCP for Real Time Multimedia Applications. *ICNP*, 2002.

[15] Y. Liang, E. Steinbach, and B. Girod. Real-time Communication over the Internet Using Packet Path Diversity. *ACM Multimedia*, pages 431–440, 2001.

[16] D. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Trans. Comm.*, COM-39:1482–1493, October 1991.

[17] A. Odlyzko. Data Networks are Lightly Utilized, and will Stay that Way. *The Review of Network Economics*, 2(3):210–237, September 2003.

[18] V. Paxson. End to End Internet Packet Dynamics. *IEEE Trans. Networking*, 7(3):277–292, June 1999.

[19] A. Percy. Understanding Latency in IP Telephony. www.totaltele.com/whitepaper/docs/-Understanding_Latency.pdf.

[20] R. Ramjee, J. Kurose, D. Towsley, and H. Schulzrinne. Adaptive Playout Mechanisms for Packetized Audio Applications in Wide-Area Networks. *Proceedings of IEEE Infocom*, 1994.

[21] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889 - RTP: A Transport Protocol for Real-Time Applications. http://www.faqs.org/rfcs/rfc1889.html.

[22] H. Schwetman. CSIM: a C-based process-oriented simulation language. *Proceedings of the 18th conference on Winter simulation*, pages 387–396, 1986.

[23] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE Trans. Networking*, 4(3):375–385, June 1996.

[24] V. Toncar. Simple OpenH323 Tutorial. http://toncar.cz/openh323/tut/.

[25] U. Varshney, A. Snow, M. McGivern, and C. Howard. Voice Over IP. *Communications of the ACM*, 45(1):89–96, January 2002.