Real Time Extensions to TCP

Course Project
ECE371BW, Fall 2001

Srikanth Kandula [kandula@uiuc.edu]
Sankalp Singh [ssingh7@uiuc.edu]

December 15, 2001

Abstract

This independent project involved the design, testing and evaluation of Real Time extensions to TCP. Specifically, the TCP protocol stack was modified in order to update both the sender and the receiver windows whenever a deadline, implemented as a timer's timeout, occurs. Such a modification, attempts to alleviate the excessive jitter seen in TCP when the receiver has to wait for the re-transmission of a few lost packets while most other packets have been received. This report discusses the need for such a modification (Section 1), evaluates existing approaches to achieve similar results (Section 2), states the proposed solution (Section 3), enumerates the implementation details (Section 4) and provides experimental results to quantify the performance of these suggested modifications (Section 5) . Finally, we mention the yet unresolved limitations inherent to our approach and present possible methods of tackling these problems.

1 Introduction

This section defines the problem that this independent project attempts to tackle. Choice of an appropriate Transport Layer protocol for multimedia applications is a complex process due to limited real-time support in existing popular transport layer protocols. While UDP has the advantage of timely delivery of data, it does not guarantee delivery. Thus the use of UDP can lead to loss of several multimedia frames leading to a low SNR. On the other hand, reliable protocols such as TCP guarantee the in-sequence delivery of data, but could lead to excessive jitters. These jitters could be introduced due to differences in the network bandwidth perceived by each individual packet, but packet loss events greater jitter into the data stream. This is because if a packet has been lost, TCP waits until the lost packet can be re-transmitted before forwarding later packets to the application. Doing so, might cause late delivery of all the packets which would render them unusable by the application. Thus modifications to TCP that allow TCP to reduce the application perceived jitter would be very useful for multimedia applications. Specifically this would involve the implementation of certain mechanisms that would cause TCP not to wait indefinitely for lost packets.

2 Related Work

Traditionally transport layer support for real time network traffic has been performed using modified version of UDP or through QoS. Several new protocols such as RTCP, SCP [1] use UDP as the underlying framework to provide transport layer support for real time traffic. These protocols basically deal with integration of re-transmission based reliability mechanisms and congestion control mechanisms on top of UDP. However, the main drawback of such schemes is their inability to provide for effective congestion control/flow control. This would cause real time applications using these protocols to become in-sensitive to network congestion problems. Additionally, well-behaved TCP flows that respond to congestion might be denied service by such overly-aggressive traffic. Thus transport layer protocols using UDP are inherently flawed, not only from the application's point of view but also from the point of view of the network as a whole because fair (policy based) sharing of bandwidth can no longer be performed. Novel schemes to alleviate this problem have been proposed, such as the Congestion Manager approach by Balakrishnan et. al [5]. A Congestion Manager located on a host, is responsible for unified congestion control of all the transport connections that start or terminate at this host. Thus the sensitivity to network congestion of UDP based flows can be achieved, though not in to the degree desirable.

Other protocols such as RTP [3] that provide transport layer support for real time network traffic work through QoS provisioning. These protocols belong to the IntServ approach which assumes that several routers have the capability to maintain state information. Thus flows could be ' 'allocated" bandwidth, depending upon their requests (either sender driven or receiver driven such as RSVP [4]). Thus real time traffic is classified into different QoS levels, each of which is guaranteed different bandwidth, latency etc. A classification that allows real time traffic to have a higher priority than normal TCP traffic could be used to provide "usable/desired" support for multimedia applications. However such schemes have the major disadvantage that real time traffic is always assumed superior to normal traffic regardless of the content. This could be an inversion of the provisioning policies of the network managing authority. On the other hand several multimedia applications can adapt to changing network support. By causing a static assignment of resources to such multimedia applications, QoS provisioning schemes fail to utilize this functionality of the applications. Among schemes that perform QoS provisioning are two broad classes, those that rely on the IntServ framework such as RSVP [4] and those that rely on the DiffServ framework [2]. Of late, DiffServ based protocols are considered more useful as these would entail the deployment of a large number of fast routers (or switches) that store little state information and a small number of border nodes that store state and perform QoS provisioning. Another problem with these schemes is that they necessitate the large scale deployment of several protocols in order to achieve optimal transport layer support for real time applications.

A recent paper, TCP-RTM [6] attempts to obtain better support for real time applications than TCP with minimal modifications to the TCP stack. Additionally TCP-RTM is demonstrably fair when compared with normal TCP. The main idea can be described as below

- Whenever an out-of-order packet arrives, check whether the application that this packet is destined for is blocking to receive. If so, pass the out-of-order packet directly to the application and update the receiver window.
- When an application performs a blocking receive call and there is no data available in the sockets receive queue, the first contiguous sequence of packets from TCP's out of order queue are delivered to the application.

In addition, to the above TCP-RTM uses application level jitter buffers and disables buffered sends (Disables Nagle's algorithm) at the sender. Thus the modifications to TCP attempt to solve all excessive jitter caused by packet losses as these are reflected as out-of-order delivered packets. Application level jitter buffers are used to negate the jitter introduced by transients in the available network resources. Disabling Nagle's algorithm enforces that the sender not buffer frames to guarantee timely sending of data. In conjunction with this, a suitable MSS is set at the sender so as to enforce that only an integral $(1, 2, 3 \ldots)$ application-level frames can be transmitted in one transport layer packet. This scheme is essentially a good choice of solutions and performs well.

3 Proposed Solution

Our solution is motivated in part by TCP-RTM [6] which has been described above. We believe that the transport layer can efficiently negate the jitter in TCP caused due to packet losses, while the application is assumed to take care of network introduced jitter by using a jitter buffer. However our solution uses a unique approach to negate jitter caused due to packet losses, using timers at both the sender and the receiver. Each of these timers is guided by a soft deadline that is specified by the application and perform the following:

- **Timer at the Sender end**: The kernel is provided with application specified rate at which packets are to be sent, i.e. S bytes in T_S seconds. Our modification guarantees that the sender receives acknowledgments for S bytes in T_S seconds. Upon a timeout, the sender window is suitably adjusted (by updating $tp \rightarrow snd\_una$) to enforce this guarantee.
- **Timer at the Receiver end**: The kernel is again provided with values of R and T_R by the receiver. Using these values, the kernel guarantees that R bytes are received in T_R seconds. Upon a timeout, the receiver window is suitably

adjusted (by updating $tp \rightarrow rcv\_nxt$) to enforce this guarantee. The detailed implementation of these timers has been described in Section 4.

The sender timer was necessary in order to account for lost acks, which could cause the sender to freeze his sender window and the receiver to wait for a long time in order to receive data. We presume that the application performs a sequence number based framing and thus can identify and reconstruct lost packets without any help from the support. Additionally, the sender is also responsible for disabling the Nagle's algorithm for buffered sends and for setting an MSS that is an integral multiple of the frame size. Finally, we assume that the above-mentioned values, i.e. R, S, T_R and T_S are obtained in the following fashion.

- S and T_S are obtained based on the rate at which the sender attempts to send data.
- An application level handshake, using a different control connection (such as in FTP) is performed prior to the begin of data transmission. During this handshake the sender communicates its values of S and T_S to the sender. Using these values, the receiver initializes R and T_R suitably and passes these values to the kernel, through a setsockopt call on the receivers data socket.
- The actual method of initializing these values is mentioned in Section 4.

4 Implementation Details

Modifications to the applications, in order to disable Nagle's algorithm and establishing a suitable MSS, were trivially performed using the *setsockopt* system call. Non-trivial modifications that were performed during the course of this project are described below:

4.1 Choice of R, S, T_R and T_S

The sender chooses S and T_S based on the rate at which packets are being sent out onto the network. S is chosen as α times the size of each packet (not frame) where α is a small constant ($5 \leq \alpha \leq 10$). T_S is chosen to be α times the difference between the sending times of consecutive packets at the sender plus another small constant β. This is because the sender uses S and T_S to move $tp \rightarrow snd\_una$, i.e. these values are used to guarantee that S bytes would be acknowledged in T_S seconds. The small constant β is about RTT/2 to accommodate for the ack to be received at the sender.

The values of S and T_S are communicated to the receiver using a control channel prior to data transmission. Using these vales, the Receiver calculates R and T_R as $R = S$ and $T_R = T_S - \gamma$, where γ is another small constant. The idea is to have the receiver time out earlier than the sender so as to accommodate for new acknowledgment packets to be received by the receiver. The later sub sections detail the kernel modifications needed to implement these timers.

4.2 Moving the Sender window

The relevant modifications are presented in Appendix A and in Appendix B. The following major modifications were necessary:

Creating new socket options
Two new socket options TCP_SRTM_SIZE and TCP_SRTM_TM were defined in */include/linux/tcp.h*. The application used these socket options to pass the values of S and T_S to the kernel.

Reading application specified values
To record the arguments exercised by these options we modified *tcp_setsockopt* in *net/ipv4/tcp.c*. Specifically these arguments were stored in the *struct tcp_opt* associated with this socket and an additional field was set to denote that these options have been initialized.

Initialization of the timer
The sender timer was initialized upon the receipt of the first acknowledgment packet is received in the function *tcp_ack* in *net/ipv4/tcp_input.c*. Whenever a call to the above mentioned function occurs the sender timer is initialized if it is not already initialized. Initialization of the sender timer is done by a call to *tcp_init_rtm_snd_timer* which initializes a new timer $tp \rightarrow$ *rtm_snd_timer* using *init_timer* and adds this timer to the list of currently active timers using *mod_timer*. This timer is statically configured to timeout every T_S seconds based on the application specification. As would be discussed later, the use of a static timer is one of the limitations of the current implementation.

Actions to be taken whenever the sender timer expires
Whenever the sender timer expires a variable $tp \rightarrow$ *last_snd_una* stores the number of the first data byte that was un-acknowledged when the timer last expired. Using this value and the application specified value S, the kernel checks whether S bytes have been acknowledged since the last expiration of this timer. This is performed by comparing

the value of $tp \rightarrow snd\_una$ with that of $tp \rightarrow last\_snd\_una$. If enough data bytes have already been acknowledged, the value of $tp \rightarrow last\_snd\_una$ is updated to the current value of $tp \rightarrow snd\_una$ and the sender timer is re-initialized to T_S. If not, then we assume that S bytes after $tp \rightarrow last\_snd\_una$ have either already been received by the receiver and their acknowledgments have been lost, or these data bytes are no longer useful to the receiver and thus need not be resent. Hence we use a modified version of *Tc_clean_rtx_queue* to remove these data bytes off the write queue of this socket, where they have been stored for possible re-transmissions. After cleaning the re-transmit queue, we assume that the sender window size remains constant and check whether we could send out more data. This is done by calling the function *tcp_data_snd_check* after cleaning the re-transmit queue. The code for this event is present in *tcp_rtm_snd* in file *net/ipv4/tcp_input.c*.

4.3 Moving the Receiver window

As has been mentioned earlier, the relevant code is presented in Appendix A and in Appendix B. In the following, we describe the major changes necessary:

Creating new socket options
Two new socket options TCP_RRTM_SIZE and TCP_RRTM_TM were defined in */include/linux/tcp.h*. The receiver uses these socket options to pass the values of R and T_R to the kernel that are obtained after the application level handshake using a different control channel.

Reading application specified values
To record the arguments exercised by these options we modified *tcp_setsockopt* in *net/ipv4/tcp.c*. Specifically these arguments were stored in the *struct tcp_opt* associated with this socket and an additional field was set to denote that the receiver timer has been initialized. The actual initialization of the receiver timer is deferred till later.

Initialization of the timer
The receiver timer was initialized upon the receipt of the first data packet in *tpc_data_queue* or in *tcp_rcv_established* whichever occurs earlier. Whenever a call to either of the above mentioned functions occurs, the receiver timer is initialized if it is not already initialized. Initialization of the receiver timer is done by a call to *tcp_init_rtm_receiver_timer* which initializes a new timer $tp \rightarrow rtm\_receiver\_timer$ using *init_timer* and adds this timer to the list of currently active timers using *mod_timer*. This timer is statically configured to timeout every T_R seconds based on the application specification. As

would be discussed later, the use of a static timer at the receiver, is another limitation of the current implementation.

Actions to be taken whenever the receiver timer expires
Whenever the receiver timer expires a variable $tp \rightarrow last\_rcv\_nxt$ stores the number of the first data byte that the receiver was waiting for when the timer last expired. Using this value and the application specified value R, the kernel checks whether R bytes have been received by the receiver since the last expiration of this timer. This is performed by comparing the value of $tp \rightarrow rcv\_nxt$ with that of $tp \rightarrow last\_rcv\_nxt$. If enough data bytes have already been acknowledged, the value of $tp \rightarrow last\_rcv\_nxt$ is updated to the current value of $tp \rightarrow rcv\_nxt$ and the sender timer is re-initialized to T_S. If not, then we assume that R bytes after $tp \rightarrow last\_rcv\_nxt$ are no longer useful to the application if they are received at a later time. Thus as much of these R bytes that have been received and are present in the out of order queue of the receiver are moved into the sockets receive queue. While moving data out of the out-of-order-queue, we disregard any holes that might arise in the data. However we do not move the rcv_nxt pointer beyond the last data packet that has been received by the application. This guarantees that the sender and the receiver do not lose synchronization and is consistent with our objectives from this timer, specifically this timer provides us with a mechanism for TCP to recover from packet loss. Given the fact that the jitter introduced due to transient network conditions would be uniform across all transport layer protocols and is not specific to TCP, we assume that the application takes care of this jitter through the use of an application level jitter buffer. This is also consistent with the design rationale of [6]. The movement of out of order data into the sockets receive queue is performed using a modified version of *tcp_ofo_queue* and is present in *tcp_rtm_recv_ofo* in $net/ipv4/tcp_input.c$. After moving data out of the out-of-order queue, we schedule an ack by a call to *tcp_schedule_ack* if any data has at all been moved out of the queue. Subsequently, we cause the receiver to check whether it can send out any fresh data to the sender, as an acknowledgment could be piggybacked on this data packet and also cause it to send an acknowledgment if necessary. These are done by issuing calls to *tcp_data_snd_check* and *tcp_ack_snd_check*. The kernel does not help the application in identifying holes in the out-of-order queue, i.e. out-of-order packets are merely appended to the sockets receive queue. The application uses an application level sequence number to identify the packet loss event.

4.4 Porting to the host kernel

The above changes were first made in UML and were then ported to the host kernel. Porting involved a simple copy of the modified files into the host kernel and a re-compilation. The only notable features are that the timeout value of each timer is calculated of the order of 1/HZ, where HZ is an architecture specific parameter. On the UML kernel the timers had a resolution of 50mSec, while the timers had a resolution of 10mSec on the source kernel. This necessitated us to make a change in the specification of the timeout value. Further it has been noticed that the RTT, delay and jitter are significantly smaller when the applications run on the host kernel rather than when they run on the UML. This is possibly because the UML itself runs in user space and will need to be scheduled by the kernel.

5 Results

We observed that the use of remote TCP echo servers leads to spurious results. This is because the sender socket of the remote TCP echo server usually does not have Nagle's algorithm disabled. This leads to un-wanted buffering of the packets before they are returned to the application. Thus we have tested our modified TCP, TCP realtime, using TCP encapsulated in UDP packets. We also experimented with TCP Echo one way, i.e. ACKS are forwarded locally, and TCP Echo two way, i.e. ACKS are also send out to the remote UDP echo server. The following are the results with varying time constraints.

5.1 Internet Tests

Three applications, the "sender", the "receiver" and the "player" are all present on the local machine. The sender sends an audio stream to the receiver which is replayed off a remote UDP echo server (TCP Encapsulated in UDP). The receiver forwards the packets to the player as soon as they are received. The sender is sending 528 bytes periodically. The value of S was chosen to be $528 * \frac{1}{T_S(in seconds)}$ and the value of T_S(in tens of milliseconds) is varied from 10 to 1000. The values presented below are the averages over 10,000 packets. The loss percentage is calculated as the $\frac{number of packets lost * 100}{number of packets sent}$.

Site	Time Constraint	Delay in mSec	Jitter in mSec	Loss Percentage
202.141.127.131 (www.iitkgp.ernet.in)	10	3.978	29.926	0.6849%
202.141.127.131 (www.iitkgp.ernet.in)	50	27.672	44.645	0.2819%
202.141.127.131 (www.iitkgp.ernet.in)	100	30.672	50.045	0.0981%
202.141.127.131 (www.iitkgp.ernet.in)	1000	35.672	55.045	0.0087%

We observe that with increasing granularity of the timeout deadline, the efficiency of the timer decreases. This is because a significant amount of jitter is still passed onto the application because the timer times out only once every 1000 packets say. Hence a small value of T_S roughly of the order of the inter-packet transmission delay at the sender was found to be a good choice. As can be see above, the jitter in the case of smaller timeout values is sufficiently low. We believe that this remaining amount of jitter is caused primarily due to the transient conditions of the network and is to be resolved at the application (possibly by using application level jitter buffers). Finally, since the loss rates on the Internet have been found to be significantly small, we believed that testing TCP realtime on the Internet does not constitute sufficient stress testing of the software. Thus we performed an experiment in the lab, which is described next.

5.2 Experimental Setup

As before, three applications the "sender", "receiver" and "player" are all located on a single host. The remote UDP echo server however is substituted with a local server "students.uiuc.edu". However the path from the host running the applications to the server was caused to lead through a PPP link which was heavily congested as large UDP packets using the program "scok" were transferred between the end-points of the PPP link. The topology is as follows.

$$HostA \rightarrow HostB \rightarrow HostC \rightarrow HostD \rightarrow HostE$$

The hosts were as follows, 192.168.0.15 (A), 192.168.0.9 (B), 192.168.0.7(C), 192.168.0.1(D) and 128.174.5.39 (Host E). The reverse path was symmetric. We obtained the delay, jitter and SNR values for the base case of normal TCP, TCP realtime one way and TCP realtime two way. The results have been graphically presented.

Figure 1 plots the jitter and delay of normal TCP versus those of TCP realtime. It can be clearly seen that both the delay and the jitter have been halved. We believe that this is because TCP realtime reduces the jitter introduced due to packet losses and re-transmissions in TCP to zero. However the application still perceives jitter due to transients in the available network resources. As stated earlier, we believe that it is the responsibility of the application to smooth out these jitters, as these jitters would be caused in any transport protocol, not just TCP. Thus, it is not the responsibility of TCP real time to take care of jitter due to transients in network resources.

Figure 2 denotes the same values as before but for TCP two way. Again the jitters and delays are significantly reduced by TCP realtime.

Figure 3 shows the SNR of the resultant output stream for TCP realtime vs that for normal TCP. It can be seen that both the realtime versions perform almost similarly. The SNR for realtime case is significantly smaller as this is a contrived network, which has a high proportion of losses and thereby a high proportion of timeouts.

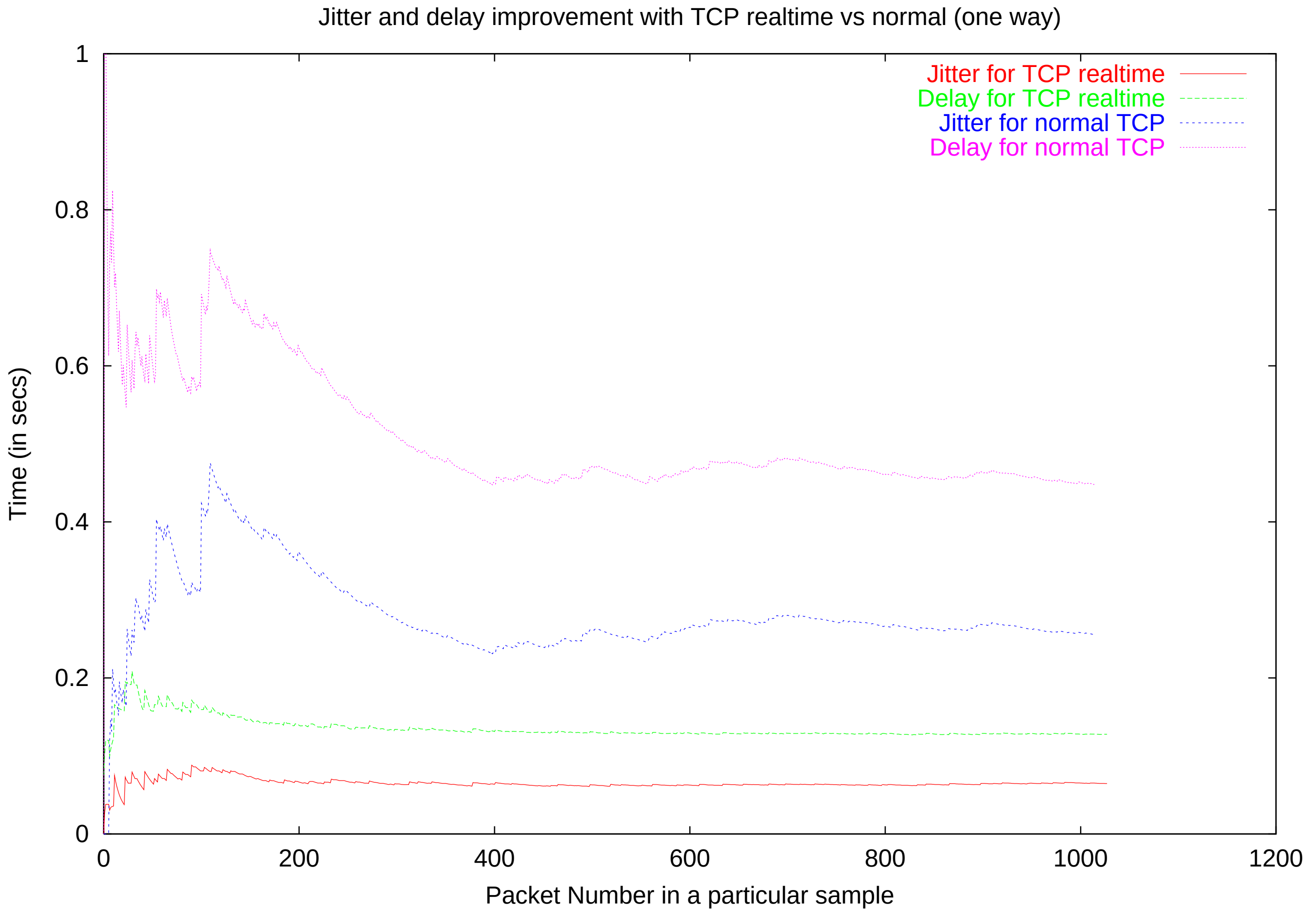

Figure 1: Jitter and Delay improvement with TCP realtime vs Normal TCP (one way)

This graph plots the number of lost packets vs the number of data packets actually transmitted. As has been mentioned before, since this is a contrived network, in which TCP has to compete with an un-responsive UDP flow, a very high proportion of packets have been dropped. The fact that these losses are visible to the application (by sequence numbers in application level frames) is proof of the correctness of TCP realtime.

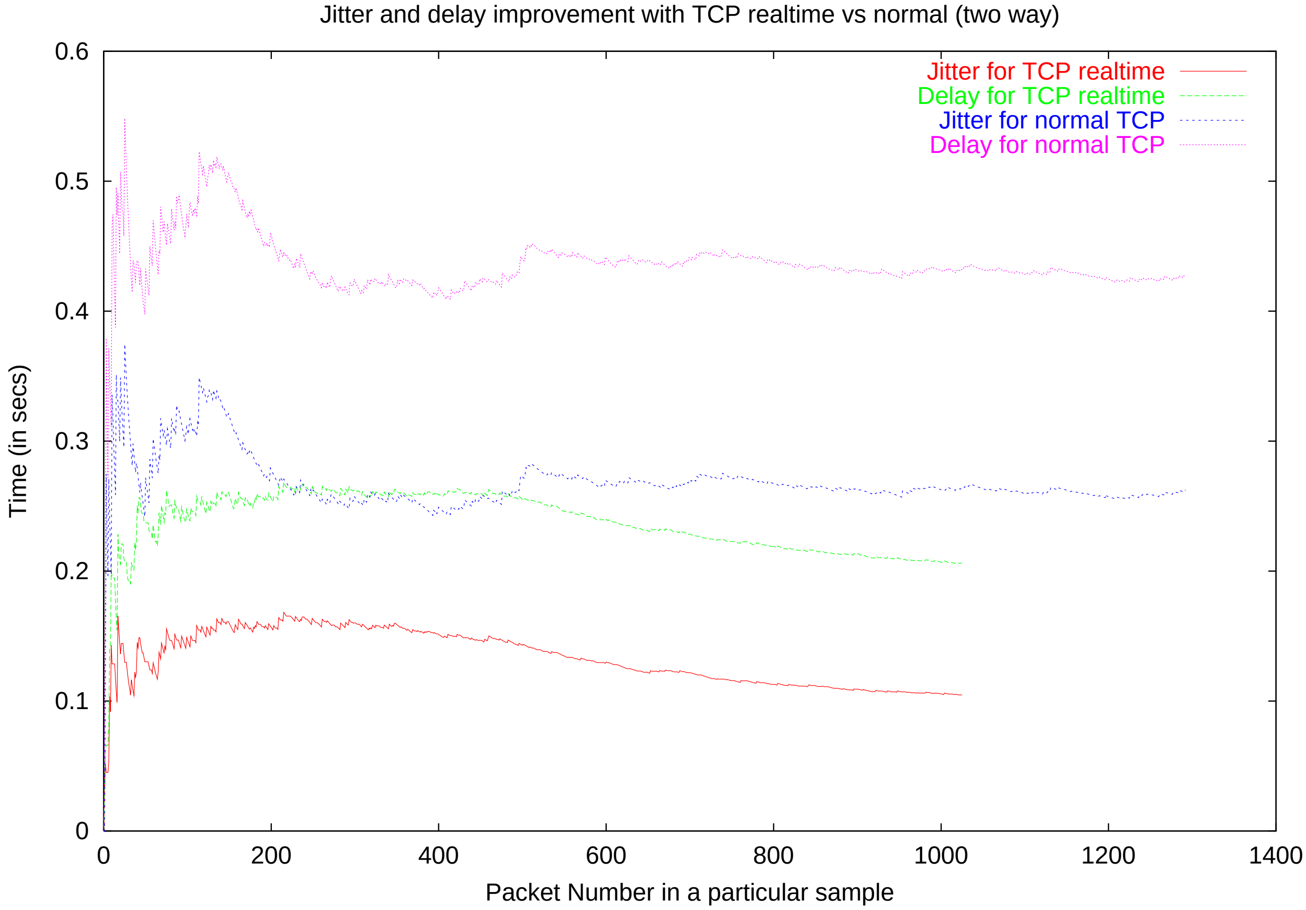

Figure 2: Jitter and Delay improvement with TCP realtime vs Normal TCP (two way)

6 Experiences, Limitations/Improvements to our approach

Kernel modifications required were simple to perform but required a significant amount of understanding of the Linux kernel code. A significantly larger amount of time was spent in understanding the code and in testing the final implementation than in performing actual kernel modifications. Coding the applications in order to test TCP real time also took considerable time and effort. We identified the spuriousness of runs that use TCP echo from remote TCP echo servers since the remote TCP echo server buffered packets prior to echoing back to the sender. Several limitations to our approach have been presented during the demonstration. They are

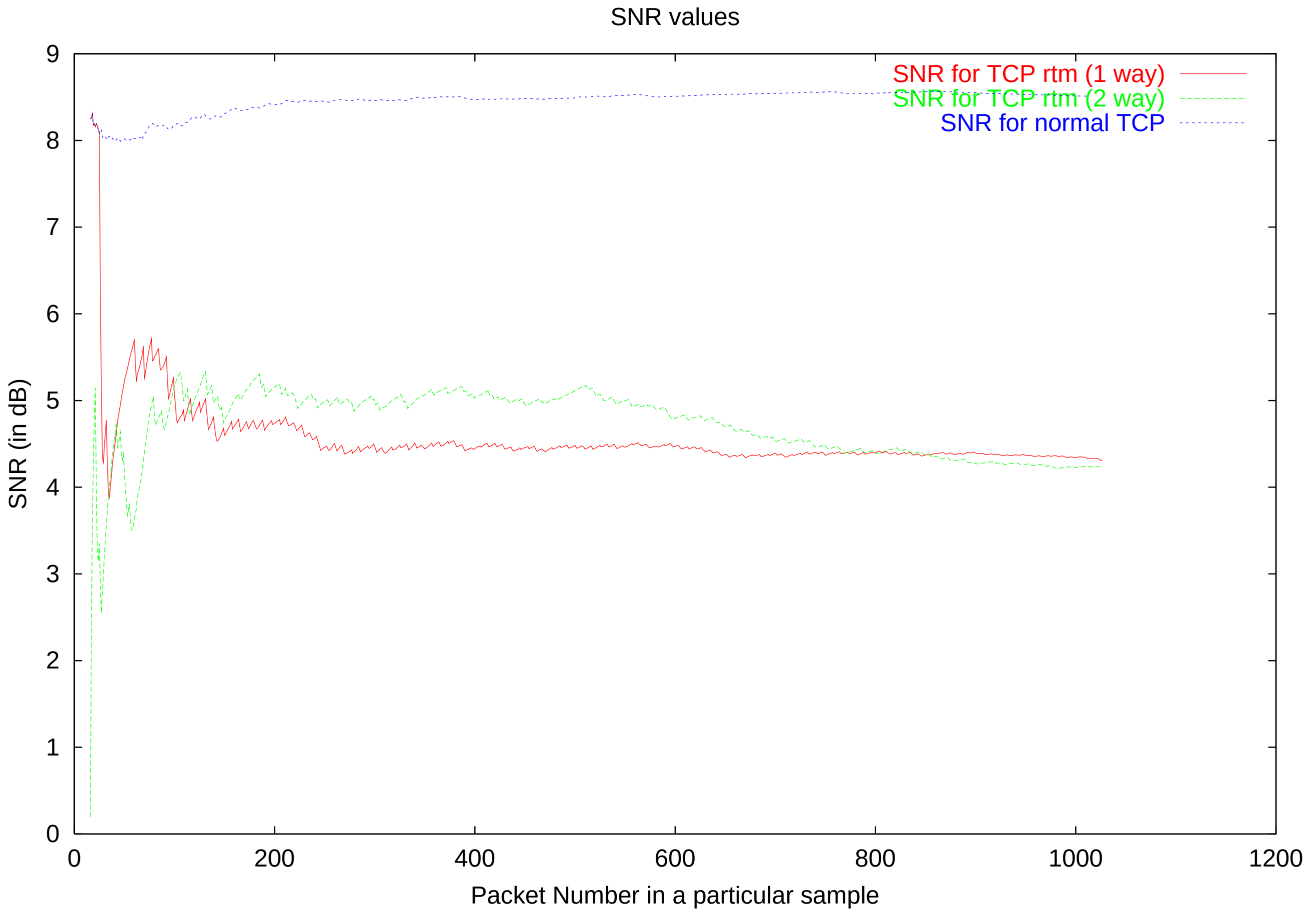

Figure 3: SNR for TCP realtime vs Normal TCP

- If more number of data bytes (acks) arrive in one timeout interval than in the other, we would update $tp \rightarrow last\_rcv\_nxt$ to the current value of $tp \rightarrow rcv\_nxt$. This would cause problems if lesser number of data bytes arrive in the next timeout interval. Instead we should have either updated $tp \rightarrow last\_rcv\_nxt$ by R whenever more than R data bytes are received in one timeout interval, or reduce the number of data bytes that would need to be received in the next timeout interval as we have received more than our fair share in this interval. A similar modification could also be made on the sender side. Thus the timeout interval and or the amount of data that we expect to receive should change dynamically corresponding to feedback form earlier timeout intervals.
- Initialization of the timers is a problem as the presence of excessive jitter in the beginning of a connection would lead to the setting up of incorrect timeout

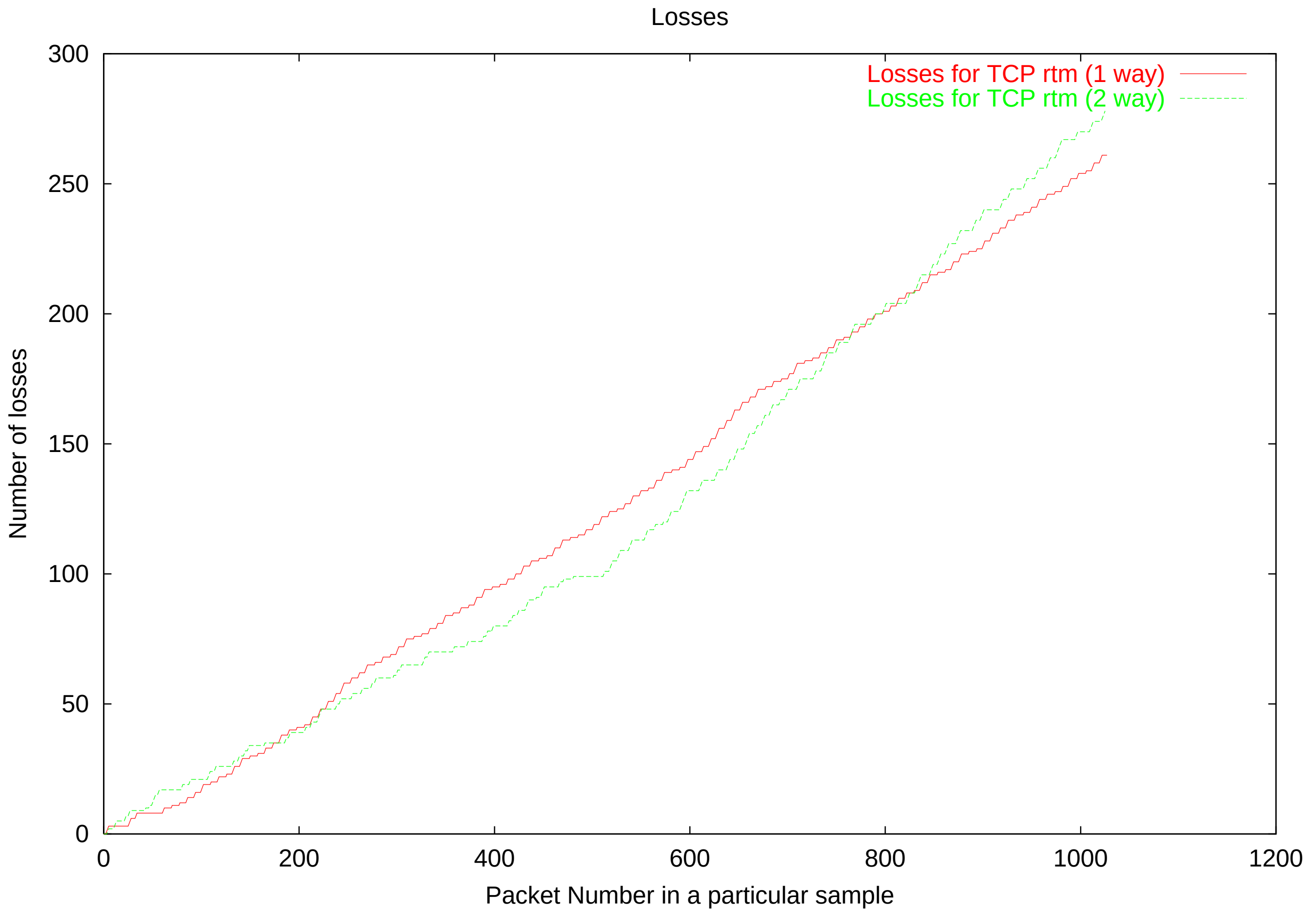

Figure 4: Loss values for TCP realtime

values. Thus our timer interval needs to be modified based on at least a few initial packets or dynamically. In short, the application specified value should only be used as a directive, but the timeout value should adapt to the network transients.

We learned a lot from this project, especially how to understand, design and build solutions. The key idea was that no solution is foolproof and would need to be carefully analysed to detect possible limitations.

7 Conclusion

We have demonstrated that by suitably modifying the sender and receiver windows, one could reduce to zero, the jitter present in a TCP real-time stream that is caused due to packet losses. We believe that the residual jitter which is due to transients in the network resources is common across all transport layer protocols and is not specific to TCP alone. Thus this residual jitter has to be taken care of by the application possibly by the use of an application level jitter buffer. On the whole, the idea of moving the sender and receiver windows to achieve better real time guarantees in a TCP stream has been demonstrated to work.

References

[1] S. Cen et al. *Flow and congestion control for Internet Media Streaming Applications*, Proc. of Multimedia Computing and Networking, 1998.

[2] Kathleen Nichols, Van Jacobson, and Lixia Zhang, *A Two-bit Differentiated Services Architecture for the Internet,"* IETF Internet draft (draft-nichols-diff-svc-arch-00.txt), November, 1997.

[3] H. Schulzrinne et al. *RTP: A transport protocol for Real-Time Applications.* IETF RFC 1889.

[4] R. Braden et al. *Resource ReSerVation Protocol (RSVP)* IETF RFC 2205.

[5] H. Balakrishnan et. al. *An integrated Congestion Management Architecture for Internet Hosts*, SigCOMM 1999.

[6] S. Liang et. al *TCP_RTM: Using TCP for Real Time Applications*, InfoCOM 2001.

A Code fragment from *tcp_input.c*

Attached along with this report.

B Code fragment from *tcp_timer.c*

Attached as a separate document along with this report.