The Simulation & Improvements of the RTP

Barry Soesanto, Ray Martinez, Swarna Bhuvanapalli, Timothy A Rock
INFS 612 Project
Professor Yih-Feng Hwang

Abstract

The main goal of our project is to create a tool in Java to simulate the limitations of the best-effort service (jitter, packet loss, & delay) used by any UDP-based protocol in real-time, one-way, stored multimedia streaming. The visuals & reports produced by this tool will be useful for scientists to analyze & experiment with their own multimedia streaming protocol. To put this tool into practice, we chose to experiment with the RTP. In order to accomplish this, we implemented the RTP on top of the tool using the Java's standard UDP API. Another goal of the project is to research existing improvements of the best-effort service in the transport layer especially on top of RTP. Thus, in the future, these improvements might be implemented on top of our tool's platform to assess their tolerant against jitter, packet loss, & delay.

1. Introduction

Since the main goal of this project is in developing the tool to implement & test an RTP-based protocol, the research portion has been focused on a wide variety of improvements so the reader will already have various options to be implemented & tested against network affects, loss, and potentially jitter and delay, hopefully, using our tool. The architecture of this tool is explained in section 4.

RTP is a UDP-based protocol that implements an RTP Packet to stream a multimedia in real-time. An RTP Packet consists of a UDP datagram plus several RTP information such as sequence number, timestamp, & the type of the media encoding being streamed [3]. This portion of the info is called the RTP Header.

2. Research Problem

Judging from the info provided by RTP Header, it's obvious that RTP alone, like UDP, only provides a best-effort service. In other words, streaming in RTP may suffer from jitter, delay, & packet loss. We believe that packet loss is the most significant affect as this results in congested network elements. Therefore our work has focused first on packet loss.

The popular approaches to solve these problems are the adaptive playout delay, forward error correction, & interleaving [3]. However, these approaches can be considered superficial because they will work only to some degree of loss, delay, or jitter.

Another different approach is to fix the unreliable, best-effort nature of the network layer by means of Intserv & Diffserv [3]. Although this approach promises a quality of service as reliable as TCP, it has faced a deployment issue if applied to all existing routers in the network core [3].

Thus, we are interested in researching solutions that stand between these two ends of approaches. However, we'd like to avoid streaming the media over TCP due to its jerky drop of rate in its multiplicative decrease algorithm. In addition, we'd like to avoid its mandatory retransmission of packets.

3. Related Research Works

3.1. TFRC

One of the improvements is to address the congestion problem in the router queue. Under this approach, jitter, delay, & packet loss are really due to congested router queues. They become congested because multimedia transmission keeps bursting packets without any congestion control [3]. This lack of control is obviously the inherent nature UDP or any multimedia protocol on top of it.

The forward error correction, adaptive playout delay, & interleaving only remedy the symptoms of the root problem of congested router queues superficially, but they don't address the root problem itself. Those superficial solutions are definitely effective most of the time; however, there's a limit that they might break down as long as the routers remain congested.

A straightforward solution might be to stream the

multimedia over TCP that has a congestion control plus much more. However, the sudden & jerky drop of rate in the AIMD algorithm might impact the filling of the buffer at the application layer at the receiver part. Of course, this won't be a problem if the streaming simply needs to burst the packets into the media player as fast as it can. However, if the streaming has a fixed cadence & playout rate, this drop may impact the quality of the media when viewed by the user [5].

Thus, a better solution would be to smoothen the rate drop in the Congestion Avoidance in TCP congestion control & adopt that modified version of the control into a UDP-based protocol. Experts have been using all of those arguments above to justify the positive impact to implementing congestion control over a UDP-based protocol. They propose the TCP-Friendly Rate Control (TFRC) protocol that implements such control over multimedia streaming [5]. They then go further by specifying a whole new transport-layer protocol. This new protocol of course has the TFRC in it. This new protocol is called Datagram Congestion Control Protocol (DCCP) [7].

Surprisingly, this protocol is intended to be deployed between the UDP & the network layer instead of on top of the UDP. The experts argued that designing a new protocol on top of the UDP would raise a compatibility issue with the existing applications running over UDP-based protocols [7].

TRFC is not a whole new transport-layer protocol; rather, it's merely an algorithm of a TCP-like congestion control for multimedia streaming.

In a nutshell, TFRC works by adjusting the sending rate of packets (instead of adjusting the packet size) upon receiving the info from the receiver about loss packets. When a packet loss occurs, instead of halving the sending rate, the sender comes up with the new sending rate by feeding the info about loss packets, & other info such as packet size, round trip time, retransmission timeout value, into an equation [6]. Unlike the jerky zigzag-line chart depicted in the TCP AIMD algorithm, the chart for TFRC as a result of this equation will depict a sinusoidal line [6]. The detail description of how this algorithm works is described in section 4.

3.2 SELECTIVE RETRANSMISSION CAPABILITY

In a congested network this helps to further reduce congestion by not retransmitting frames that were not lost. Only those frames that are lost or corrupted are retransmitted. It Balances the extremes of TCP and UDP and retransmits a percentage of lost packets .The amount that is retransmitted depends on several Quality of Service (QoS) factors including current loss , round-trip time, network congestion, and is tuned to provide the best possible multimedia quality given the network conditions.

The play out buffer has a temporary storage for incoming packets. when a packet has not arrived at the receiver by an expected time of arrival (ETA), a round trip time for the data packet is computed. The round trip time is an estimate of a period beginning from the time a retransmission request is sent to from the receiver to the sender till the time a copy of the missing packet is received at the receiver from the sender in response to the retransmission request. This is obtained by adding a time stamp into the header and a request for an acknowledgement. When the received message is received and that header is detected with a request for reply, a reply message is sent back with the timestamp included from the original request. When the sender receives the acknowledgement it compares the current time against the sent time. This round trip time is weighted more heavily than older round trip values to come up with an estimated round trip time.

If the sender and the receiver knew about the window about how long a packet can usefully be retransmitted, it could hold transmissions around until they aged out. If the receiver accumulates some amount of data in the buffer in before it starts playing the output. Assuming the round trip time to request and receive a retransmitted packet is less than the amount of time it has in the buffer, it will have enough time to detect a need for retransmission and retransmit the packet. This way it will always have time to retransmit and re-deliver a packet before it is needed. On the other hand if information about the round trip time is not received or if the round trip time to retrieve the missing packet is greater than the remaining buffer time then the request for the missing packet is discarded and the buffer proceeds to play out the next packet available.

4. Solutions/ Analysis

4.1. Analysis of TFRC

Each time the sender sends a packet, the receiver will sends a feedback packet in return. Through this mechanism, the round trip time can be calculated [6].

4.1.1 The receiver protocol of TFRC

Each time the receiver receives a packet, it recalculates a loss event rate. In this concept of loss event rate, a packet is considered lost after 3 consecutives packets with sequence numbers after the lost packet have arrived [6]. When a packet is lost, obviously the receiver doesn't have the arrival time of that packet unlike the other arriving packets. The receiver then calculates an estimated time T_loss when the lost packet were not lost & could have arrived with the following equation [6].

T_loss = T_before + ((T_after - T_before) * (S_loss - S_before) / (S_after - S_before));

where:

S_loss is the sequence number of a lost packet.

S_before is the sequence number of the last packet to arrive with sequence number before S_loss.

S_after is the sequence number of the first packet to arrive with sequence number after S_loss.

T_before is the reception time of S_before.

T_after is the reception time of S_after.

If this T_loss <= T_old + the round trip time, where T_old is the T_loss of the previous packet loss, then this current loss is grouped into the same previous loss event. Otherwise, it's grouped into a new loss event. Thus, an interval of each loss event Ii can be calculated by means of the first & the last T_loss within that event. TFRC keeps track of a maximum of 8 lost events [6]. The loss events before them are discarded from the history.

Then the receiver calculates a weighted average I_{mean} of all Ii where i is from 0 to 7. This weighted average puts more weight on the recent intervals. Finally, the loss event rate p = 1 / I_{mean} [6]. This value p is then sent to the sender as part of the feedback message.

4.1.2 The sender protocol of TFRC

Initially the sender will Slow Start until the loss event rate returned by the receiver is greater than 0 [6]. When such value occurs, the sender doesn't go into the Congestion Avoidance like in TCP. Rather, the sender determines the new sending rate by feeding the loss event rate into a TCP-like throughput equation [6].

$$X = \frac{s}{\{ R*\sqrt{2*b*p/3} + (t\_RTO * (3*\sqrt{3*b*p/8} * p * (1+32*p^2))) \}}$$

X is the transmit rate in bytes/second.

s is the packet size in bytes.

R is the round trip time in seconds.

p is the loss event rate, between 0 to 1.

t_RTO is the TCP retransmission timeout value in seconds.

b is the number of packets acknowledged by a single TCP acknowledgement.

The round trip time R above is updated every time the feedback message is received by means of the following equation [6]:

R = q*R + (1-q)*R_sample, or if there's no feedback message before, then R = R_sample.

Where:

q = 0.9 & R_sample = (t_now - t_recvdata) - t_delay

t_now is the arrival time of the feedback packet.

t_recvdata is the timestamp of the last data received by the receiver. In other words, it's the time when that last data departed from the sender. This info is included in the feedback message.

t_delay is the duration between the receipt of the last packet at the receiver & the generation of the feedback report. This info is included in the feedback message.

When p is back to zero, the sender will back to Slow Start.

When a feedback message is not received until the no-feedback-timer times out, the sender then halves its sending rate then returns to Slow Start [4]. The expiration time of such timer is calculated by the sender each time a feedback message is received as the max of 4*R or 2*s/X [6].

However, using TFRC is not without any problem. The equation in the TFRC is only sensitive to packet loss. It's not obvious if using this equation to adjust the sending rate will eventually reduce jitter as well.

4.1.3. Possible improvements to TFRC

If a packet loss can occur because of the congested router queues, then the straightforward conclusion is that variance in packet delays (jitter) is due to a less severe congestion of router queues. In other words, when the queue remains congested for a significant period of time, it's called a persistent congestion [2], and this is when an incoming packet cannot find any empty slot in the queue at all. On the other hand, if the queue is almost full but still has a few empty slots for incoming packets & that it's in the process of draining, then one of those packets will experience jitter. Such congestion is called a transient congestion [2].

Thus, an improvement would be to incorporate an additional response to this jitter into the TFRC sender protocol. Such response will prevent the congestion from becoming more severe, & thus reduce the loss event rate.

In order to accomplish this, an additional algorithm is needed at the receiver side to sense the jitter. Such algorithm is already available in the RTCP protocol. In fact, the TFRC-J protocol has been developed to extend the TFRC by adding the RTCP's jitter calculation algorithm [4].

RTCP is a protocol complementary to RTP. RTCP governs a feedback message sent by the RTP receiver once every a certain period to report important statistics such as packet loss, jitter, & delay [3].

The algorithm to calculate the jitter in RTCP basically works by observing the difference of timestamps between two consecutive received packets every time a new packet arrives at the receiver. Then a weighted average of such differences is calculated [2].

First, we calculate D(i,j)=(Rj-Ri)-(Sj-Si)=(Rj-Sj)-(Ri-Si)

where:

Ri and Rj are the receipt time of packet i & j at the receiver.

Si & Sj are the departure time of packet i & j from the sender

Then, a weighted average of Ds is calculated:

$Ji = J_{i-1} * (1 - a) + D_{(i, i-1)} * a.$

Where Ji is the current average, Ji-1 is the previous average calculated, & a equals to 1/16 in the RTCP's RFC [2].

This J value is what considered to be a jitter, & sent back from the receiver to the sender as part of the feedback message.

Originally, the sender only operates in either Slow Start when the loss event rate is zero, or using the TCP throughput equation when the loss event rate is not zero. Now, after the jitter feedback is introduced, when the loss event rate is zero but the jitter J is greater than a certain threshold, the sender will enter a TCP Congestion Avoidance rate instead of Slow Start [4]. The sender will enter Slow Start instead if the loss event rate is zero and the jitter is less than the threshold [4]. The magnitude of the threshold is 1/10 of the round trip time [4].

Thus, after all these changes the sending behavior will basically become [4]:

```
If (loss event rate > 0) {
    Use the TCP throughput equation to determine the
    new sending rate. Using this equation, unlike
    TCP, the rate won't abruptly drop.
} else if (jitter > threshold) {
    enter the CA phase;
} else {
    enter the Slow Start phase;
}
```

Another potential improvement of TFRC might be to incorporate the info about packet loss defined by RTCP into the TFRC's feedback message & to have the sender respond accordingly.

The advantage of using the packet loss information in RTCP style is that the RTCP feedback report only comes periodically instead of once for every packet sent. RTCP defines an algorithm to determine the delay between two consecutive reports [2]. In this way, the sender doesn't have to be busy expecting the feedback message from the receiver for every packet sent. Another advantage is that the calculation of packet loss by RTCP is much simpler than by TFRC. But the trade-off is that the calculation is not as accurate and sensitive as the original loss event rate variable in TFRC. This is because, unlike the loss event rate, the packet loss calculation by RTCP doesn't use history of loss events & weighted average of the event intervals.

However, we can still use this RTCP's packet loss calculation when the TFRC sender is in the Slow Start phase or in the Congestion Avoidance phase. In these phase, the frequent TFRC feedback about the loss event rate is not very useful in determining the sending rate except to detect exactly when the loss event rate becomes greater than zero.

When the sender notices a packet loss from the RTCP feedback report, it then switches to the TCP-throughput-equation phase. Within this phase, the TFRC then starts expecting the loss event rate again for every packet sent.

The RTCP calculation of the packet loss is [2]

```
expected_interval = expected - expected_prior;
expected_prior = expected;
received_interval = received - received_prior;
received_prior = received;
lost_interval = expected_interval - received_interval;
if (expected_interval == 0 || lost_interval <= 0) fraction = 0;
else fraction = (lost_interval << 8) / expected_interval;
```

where:

expected = The expected number of received packets. This can be calculated by taking the difference between the last packet's sequence number & the first packet's.

expected_prior = The expected number of received packets when the previous RTCP feedback report was sent.

Received = The number of packets actually received.

Received_prior = the number of packets actually received when the previous RTCP feedback report was sent.

The fraction here is the one sent to the sender by means of the feedback report.

It can be seen from that algorithm that the fraction variable is similar to the TFRC's loss event rate in that they both range from 0 to 1.

4.2. The architecture of the tool

The figure below depicts the architecture of the tool. There are 4 modules:

4.2.1. The main class (experiment.java)

The main class of this tool is experiment.java. This contains the main method and takes the following arguments:

- G – launch the Generator thread
- E – launch the Extractor thread
- N – launch the Network simulator thread
- C – launch the Controller thread

Experiment.java can be invoked with all arguments and a single PC will house all 4 applications. Alternatively, 4 computers could each invoke one of these functions and run separately.

Each non-Controller thread will obtain their start up information from a txt file that is named respectively from their argument (G.txt, E.txt or N.txt). This file tells the thread which UDP socket to communicate over for its commands and status.

The command-status element will be common to all applications although the specific command-status vocabulary will be specific to the module. No two modules can be listening on the same IP-Socket pair.

4.2.2. Generator

The Generator is the source of the stream of data. It will send to the test protocol a fixed number of datagrams of fixed size at a fixed interval.

Embedded in the datagram will be a sequence number and time stamp from the Generator. This information will be used by the simulator and extractor to perform their duties and to declare datagrams as having exceeded thresholds for network effects.

The Generator will listen for setup and initiation commands on its control-status port. When instructed to begin, it will start the generating of datagrams to the underlying protocol.

Upon completion of the generation, it will status the Controller with the completion and statistics, including any overruns on the sending.

4.2.3. Extractor

The Extractor will pull datagrams at the specified pace. It will also keep statistics at the datagram level, such as lost, out of sequence, and good datagrams. It will pull from a queue. A future improvement of threshold classifications will be made at the time that the datagram is being queued. This will determine the overall amount of delay and jitters experienced, and compare it against the parameters set for the experiment run.

The Extractor will listen on its control-status port

and will obtain thresholds, pre-fill queue levels and other parameters as commands from the Controller.

When the command run is complete, the Extractor will provide completion status and statistics back to the Controller for compilation.

4.2.4. Network Simulator

The Network Simulator (Simulator) will read datagrams sent by the Generator as soon as they arrive and using the effect parameters (jitter, loss and delay) calculate the scheduled departure (or droppage of the datagram if loss is required) to the Extractor.

The Simulator works totally at the UDP level and is agnostic to the higher level protocol being run on the Generator and Extractor.

It listens on its control-status port for parameters and when to start and stop the experiment. It has no knowledge of the number of datagrams being sent, so it has to be told when to stop.

4.2.5. Controller

The Controller is the heart of this tool. It pulls from its C.txt file the control status address/port pairs for each of the other modules. It then parses specific experiment runs from this file.

For each experiment run, it sends the appropriate commands to the modules in the proper sequence (Extractor, Network Simulator, and then Generator).

It watches for the completion statuses, compiles the statistics and outputs the results in a format that can be used for further analysis (comma separated values). It then continues with any subsequent experiment runs.

4.2.6. RTP Protocol

The RTP protocol is the target protocol being tested by the tool. This is a thread that is launched by either the Generator or Extractor to prepare and run. A class is defined to specify the enqueuing, dequeuing and statistics actions. The RTP protocol will inherit from this class definition to run on the tool.

Other users of this tool would substitute their target protocol here and reuse everything else as is.

4.3. Factors that need Consideration

The platform is intended to be used where each module can run on a different computer, and these modules could be geographically pretty far apart so a couple of factors that need to be considered as these might extend the experiment:

1) The time stamps or each module's perception of time is independent. The approach required the Generator to provide a time stamp and each down stream module needed to calibrate their time reference relative to the Generator's time stamp and the arrival time of the first packet.
2) Synchronization and timing the beginning, duration and end of an experimental pass needs to factor in long delays. We added a sleep command to the vocabulary of the controller to allow the user to set the duration of the experiment pass.
3) Running all modules on one computer sometimes overloaded it. This would be evidences with overruns on the Generator module. We added an over run statistic to capture this condition so the controller has the ability to subtract off the underruns from the statistics of the project. We also took advantage of the modularity of the platform. By distributing the modules across multiple computers, processing overhead was eliminated as a factor.

5. Summary & future works

Several improvements have been researched & reported in this project. However, we have not implemented one of them on top of RTP in our tool to be tested. Possible future works include comparing & analyzing these improvements to decide which one to be implemented.

TFRC itself is a growing research area. Several extensions of TFRC have to be researched further as they provide improvements when applied to a wireless network. There are also researches about incorporating various RTCP feedback calculations into the TFRC. The jitter calculation explained in this paper is just one of the examples.

The environment implements packet loss simulation. The next affect that should be added to the platform should be jitter. When the jitter delay reaches ½ of the packet transmission rate, out of sequence packet occur.

Selective retransmission capability is not implemented in this project and it was researched so that it could be considered as a future enhancement in the project experiment. There are quite a lot of

improvements that are suggested to RTP out of which this is considered most advantageous by many who have experimented with this selective retransmission by various other methods such as video data prefetch over a skipping function, retransmission over DCCP protocol and their performance has been evaluated to find that this capability is in deed very beneficial.

6. References

[1] H. Schulzrinne, A. Rao, R. Lanphier, "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998, http://www.rfc-editor.org/rfc/rfc1889.txt.

[2] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 1889, January 1996, http://www.rfc-editor.org/rfc/rfc1889.txt.

[3] J.F. Kurose & K.W. Ross, *Computer Networking: A Top Down Approach Featuring the Internet*, Pearson Education, Inc., 2005.Source Code:Chapter 7, http://media.pearsoncmg.com/aw/aw_kurose_network_3/labs/lab7/lab7.html

[4] Q. Li & D. Chen, "Analysis & Improvement of TFRC Congestion Control Mechanism", IEEE Xplore, September 2005.

[5] S. Floyd, M. Handley, J. Padhye, J. Widmer, "Equation-based Congestion Control for Unicast Applications", ACM Digital Library, August 2000.

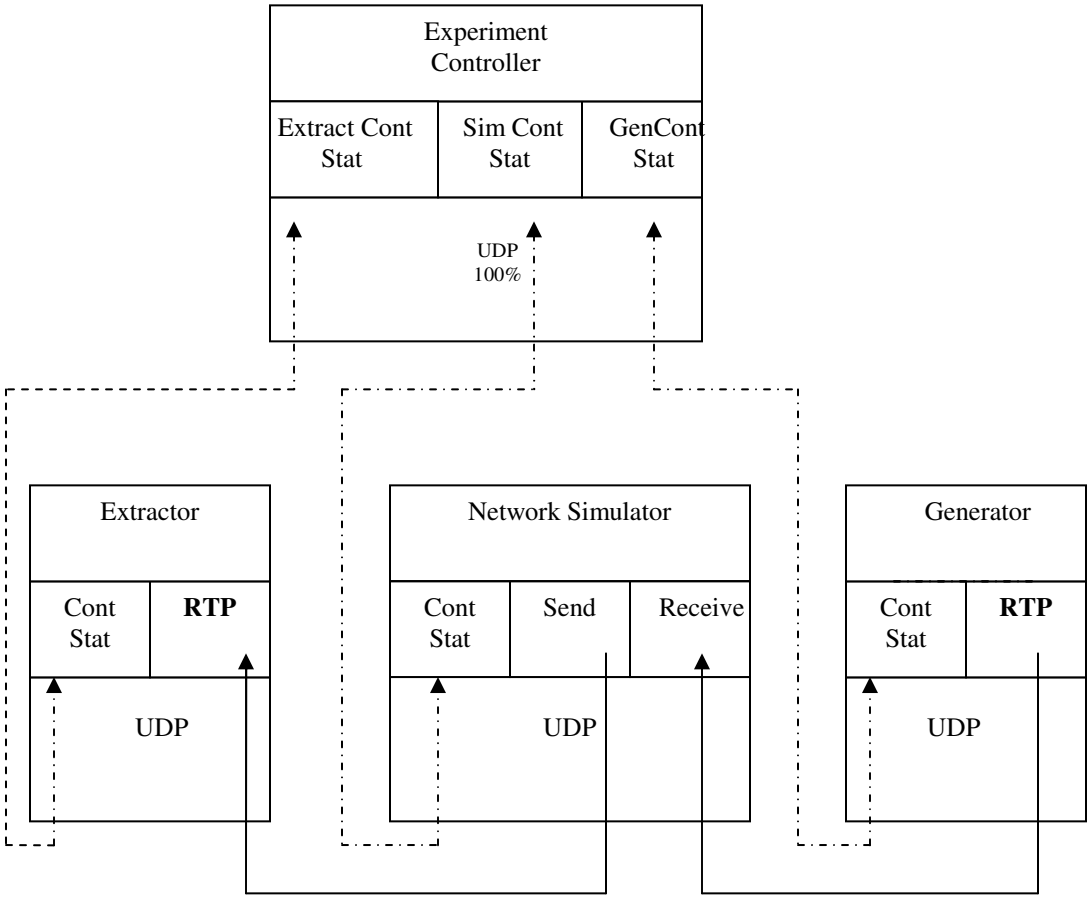
[6] S. Floyd, M. Handley, J. Padhye, J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification", RFC 3448, January 2003, http://www.rfc-editor.org/rfc/rfc3448.txt.

[7] T. Phelan, "Datagram Congestion Control Protocol (DCCP) User Guide", IETF Working Group, July 2004, http://www.read.cs.ucla.edu/dccp/draft-ietf-dccp-user-guide-02.txt.

[8] M.Piecuch, K.French, G.Oprica, M.Claypool "A Selective Retransmission Protocol for Multimedia on the Internet", http://web.cs.wpi.edu/~claypool/papers/srp/srp.pdf

APPENDIX:

ARCHITECTURE MODEL OF THE PROJECT:


Experiment Controller
Extract Cont Stat
Sim Cont Stat
GenCont Stat
UDP 100%
Extractor
Cont Stat
RTP
UDP
Network Simulator
Cont Stat
Send
Receive
UDP
Generator
Cont Stat
RTP
UDP