1 Introduction

TinyControl is a simple congestion control protocol developed on top of UDP (User Datagram Protocol). This mechanism works like a simplified version of TCP congestion control, but it does not retransmit any application data. Thus, TinyControl is used in a situation where some number of packet losses are tolerable, such as audio and video streaming service. In addition, TinyControl does not abruptly changes the transmission rate as TCP does, which halves its rate when it comes to packet loss. This is so because the stream receiver would not want to deliver the discontinuous data to its application program, which is caused by a sudden rate change. By implementing TinyControl, students will be able to learn the following:

- Help you learn about TCP-friendly congestion control mechanism.
- Help you learn to develop applications that handle stream on top of UDP.

2 Architecture

The system consists of two main components: TinyControl server and client (server and client in short). The TinyControl server is placed below an application program and on top of UDP. On the other hand, the client, which also sits in-between an application and UDP, receives data packets and sends feedback packets back to the server as in Figure 1. As the TinyControl server and client are developed on top of UDP (User Datagram Protocol), they should implement a mechanism to deal with packet losses due to the network congestion.

Specifically, whenever a TinyControl server gets started, it waits for the connection from a client. After a client connects to it, the server reads blocks from a file and starts to send the blocks to the clients along with header information (e.g. sequence number and RTT). Initially, it slow-starts as TCP does—it doubles the rate whenever a feedback packet is received before a loss event happens. Once the loss has happened, it decreases its sending rate based on the information obtained from feedback packets. After the single loss
TinyControl provides applications congestion control, but does not provide reliability. Thus, probable application will be stream services, which tolerate some level of packet loss, and playback buffer to show received data consistently to a user. The formats of Data Packet and Feedback Packet in this figure are defined in Section 2.1.1 and Section 2.1.2 respectively. UDP data packet is simply a data packet prepended by a UDP header.

occurs, subsequent sending rate is updated with the feedbacks from the client. On the other hand, the client receives data packets, updates data receive rate, sends any loss information back to the server so that the server utilizes the loss information to recalculate its sending rate. We describe the method of updating the sending rate in Section 3.1.2 in more detail.

### 2.1 Packets

TinyControl has two types of packets: data packet and feedback packet, sent by a server and a client, respectively. The payload (data delivered from application) of the data packet is fixed as 1000 bytes. The feedback packet does not contain real data, but it does contain useful information for the sender to update its sending rate. The () next to each field name stands for the size of the field in bytes.

#### 2.1.1 Data Packets

Each data packet sent by the data sender contains the following information:

+-------------------+---------------+--------+-----+-------------------------------------+
| Sequence Number (4)| Timestamp (4) | RTT (4)| Payload (1000) |
+-------------------+---------------+--------+-----+-------------------------------------+

- **Sequence number**: this number must be incremented by one for each data packet transmitted. The field must be sufficiently large that it does not wrap causing two different packets with the same sequence number to be in the receiver’s recent packet history at the same time.

- **Timestamp**: this indicates when the packet is sent. We denote by $t_{si}$ the timestamp of the packet with sequence number $i$. The resolution of the timestamp should typically be measured in milliseconds.

- **Sender’s current estimate of RTT**: the estimate reported in packet $i$ is denoted by $R_i$. The RTT estimate is used by the receiver, along with the timestamp, to determine when multiple losses belong to the
same loss event. The RTT is also used by the receiver to determine the interval to use for calculating the receive rate and to determine when to send feedback packets.

- **Payload**: this field contains the block of a file delivered from an application. It is fixed as 1000 bytes, but for the last packet, it may not be 1000 bytes.

### 2.1.2 Feedback Packets

Each feedback packet sent by the data receiver contains the following information:

```
+----------------+---------------+---------------+------------------+
| Timestamp(4)   | Elapsed Time(4)| Receive Rate(4)| Loss Event Rate(4) |
+----------------+---------------+---------------+------------------+
```

- **Timestamp**: the timestamp of the last data packet received, denoted by $t_{\text{recvdata}}$. If the last packet received at the receiver has sequence number i, then $t_{\text{recvdata}} = t_{\text{recvdata}}$. This timestamp is used by the sender to estimate RTT.

- **Elapsed Time**: the amount of time elapsed between the receipt of the last data packet at the receiver and the generation of this feedback report. We denote this by $t_{\text{delay}}$.

- **Receive Rate**: the rate at which the receiver estimates that data was received in the previous RTT, denoted by $X_{\text{recv}}$.

- **Loss Event Rate**: the receiver’s current estimate of the loss event rate $p$.

### 3 Implementation

For implementation, you are going to develop TinyControl Server and TinyControl Client as in Figure [1](#). For the server and client, we describe protocols in Section 3.1 and Section 3.2 respectively. For testing purpose, you also need to write a simple sender and receiver program. The sender reads data from a file and delivers it to TinyControl server, whereas a receiver reads data from a TinyControl client. This is denoted as the provider and subscriber in Figure [1](#) but it does not necessarily be a stream service—it can be any sort of reader and writer. However, the TinyControl server should be able to handle multiple clients’ requests (e.g. by using thread or select).

#### 3.1 Server Protocol

The TinyControl server sends a stream of data packets to the TinyControl client at a controlled rate. When a feedback packet is received from the client, the server changes its sending rate based on the information contained in the feedback packet. If the server does not receive a feedback report for four RTT, then the server cuts its sending rate in half. This is achieved by means of a timer called the nofeedback timer.

#### 3.1.1 Initialization

If the server is ready to send data when it does not yet have a round-trip sample, the value of $X$ (sending rate) is set to $s$ bytes per second. The $s$ stands for the payload size in bytes, which is fixed as 1000 bytes as we mentioned before. Thus, it sends one packet per second. Additionally, the nofeedback timer is set to expire
Algorithm 1 Sender Rate Update

**procedure** \textsc{RateUpdate}(\textit{interval})

\begin{itemize}
\item Update \(X_{\text{recvset}}\)
\item \(\text{recv\_limit} = 2 \times \max(X_{\text{recvset}})\)
\item if \(p > 0\) then
\item \quad Calculate \(X_{\text{Bps}}\) using the TCP throughput equation 3.1.4
\item \quad \(X = \max(\min(X_{\text{Bps}}, \text{recv\_limit}), s / \text{tmbi})\)
\item else if \(t_{\text{now}} - \text{tld} \geq R\) then
\item \quad \(X = \max(\min(2 \times X, \text{recv\_limit}), \text{initial\_rate})\)
\item \quad \text{tld} = t_{\text{now}}
\item end if
\end{itemize}

end **procedure**

after two seconds. Upon receiving the first RTT measurement, \(\text{tld}\) (Time Last Doubled during slow-start) is set to the current time, and the allowed transmit rate, \(X\), is set to the initial rate:

\[\text{initial\_rate} = \frac{W_{\text{init}}}{R}, \quad W_{\text{init}} = \min(4 \times s, \max(2 \times s, 4380))\]

3.1.2 TinyControl Server Behavior When a Feedback Packet Is Received

The server knows its current allowed sending rate, \(X\), and maintains an estimate of the current round-trip time \(R\). The server also maintains \(X_{\text{recvset}}\), which contains two \(X_{\text{recv}}\) every time it receives a feedback packet. The set is first initialized to contain a single item, with value infinity (2^{32}). When a feedback packet is received by the sender at time \(t_{\text{now}}\), the current time in seconds, the following actions MUST be performed.

1) Calculate a new round-trip sample:

\[R_{\text{sample}} = (t_{\text{now}} - t_{\text{recv\_data}}) - t_{\text{delay}}\]

2) Update the round-trip time estimate:

\begin{itemize}
\item If no feedback has been received before{
\item \quad \(R = R_{\text{sample}}\)
\item } else {
\item \quad \(R = q \times R + (1-q) \times R_{\text{sample}}\)
\item \}, for the filter constant \(q\), we use 0.9.
\end{itemize}

3) Update the timeout interval, which is used for setting nofeedback timer.

\(\text{RTO} = \max(4 \times R, 2 \times s / X)\)

4) Update the allowed sending rate by following Algorithm 1. This algorithm uses the variable \(\text{tmbi}\) and \(\text{recv\_limit}\):

\begin{itemize}
\item \(\text{tmbi}\): the maximum backoff interval of 64 seconds.
\item \(\text{recv\_limit}\): the limit on the sending rate computed from \(X_{\text{recvset}}\).
\end{itemize}

The procedure Update \(X_{\text{recvset}}\) in the first line of the algorithm is simple. It tries to delete the values older than two round-trip times, and add \(X_{\text{recv}}\) to set \(X_{\text{recvset}}\).

5) Reset the nofeedback timer to expire after \(\text{RTO}\) seconds.
Algorithm 2 Nofeedback Timer expired

procedure NOFEEDBACK EXPIRED
    \( X_{\text{recv}} = \max(X_{\text{recvset}}) \)
    if server does not have an RTT sample, has not received any feedback from client
    \( X = \max(X/2, s/t_{\text{mbi}}) \)
    else if \( p = 0 \) then
    \( X = \max(X/2, s/t_{\text{mbi}}) \)
    else if \( X_{Bps} > 2s \times X_{\text{recv}} \) then
    Update-limits(\( X_{\text{recv}} \))
    else
    Update-limits(\( X_{Bps}/2 \))
end if
end procedure

Note that when \( p = 0 \), the server has not yet learned of any loss events, and the server is in initial slow-start phase. In this case, the server can approximately double the sending rate each round-trip time until a loss occurs. The initial rate gives a minimum allowed sending rate during slow-start of the initial allowed sending rate.

3.1.3 Expiration of Nofeedback Timer

This section specifies the server’s response to a nofeedback timer. The nofeedback timer could expire because of an idle period or because of data or feedback packets dropped in the network. In this project, we assume there is no idle period. If the nofeedback timer expires, the server must perform the following actions:

1) Cut the allowed sending rate in half: If the nofeedback timer expires when the server has had at least one RTT measurement, the allowed sending rate is reduced by modifying \( X_{\text{recvset}} \) as described in the pseudocode Algorithm 2. The procedure Update-limits() uses the variable timer for the limit on the sending rate computed from the expiration of the nofeedback timer, as follows:

Update-limits(timer-limit):
    If (timer-limit < s/t_{\text{mbi}})
    timer-limit = s/t_{\text{mbi}}
    Replace \( X_{\text{recvset}} \) contents with the single item timer-limit/2
    Recalculate X using Algorithm 1

2) Restart the nofeedback timer to expire after \( \max(4 \times R, 2 \times s / X) \) seconds.

3.1.4 Throughput

The throughput equation \( X_{Bps} \) in Bytes per second is defined as follows:

\[
X_{Bps} = \frac{s}{R \times \sqrt{2 \times p/3 + 12 \times \sqrt{3 \times p/8} \times p \times (1 + 32 \times p^2)}}
\]

- \( s \): the segment size in bytes.
- \( R \): is the round-trip time in seconds (Section 3.1.2).
- \( p \): loss event rate (0 ≤ p ≤ 1) (Section 3.2.5).
This throughput equation can also be expressed in terms of $X_{pps}$, the sending rate in packets per second, with

$$X_{pps} = \frac{X_{Bps}}{s}$$

The parameter $s$, $p$ and $R$ need to be measured or calculated. The measurement of $R$ is specified in Section 3.1.2 and the measurement of $p$ is specified in Section 3.2.5. In the rest of this document, data rates are measured in bytes per second unless otherwise specified.

### 3.2 Client Protocol

The client periodically sends feedback messages to the sender. Feedback packets should normally be sent at least once per RTT, unless the sender is sending at a rate of less than one packet per RTT, in which case a feedback packet should be sent for every data packet received. A feedback packet should also be sent whenever a new loss event is detected without waiting for the end of an RTT, and whenever an out-of-order data packet is received that removes a loss event from the history. We first introduce a step-by-step procedure when a client received a data packet from a server in the next Section, and then explain each step in more detail.

#### 3.2.1 Client Behavior When a Data Packet Is Received

When a data packet is received, a client performs the following procedure:

- Add the packet to the packet history.
- Check if done: If the new packet results in the detection of a new loss event (Section 3.2.3), or if no feedback packet was sent when the feedback timer last expired, go to next step; otherwise, exit this procedure
- Calculate loss event rate $p$: Calculate the new value of $p$ as described in Section 3.2.5
- Expire feedback timer: Let the previous value of $p$ be $p_{prev}$. If $p > p_{prev}$, cause the timer to expire and perform the actions described in Section 3.2.2. When $p \leq p_{prev}$ and a feedback packet was sent when the feedback timer last expired, no action need to be performed.

#### 3.2.2 Expiration of Feedback Timer

When the feedback timer at the receiver expires, the action to be taken depends on whether data packets have been received since the last feedback was sent.

For the $m^{th}$ expiration of the feedback timer, let the maximum sequence number of a packet at the receiver, so far, be $S_m$ and the value of RTT measurement included in the packet be $R_m$. If data packets have been received since the previous feedback was sent, the receiver performs the following steps:

- Calculate the average loss event rate using the algorithm described in Section 3.2.5
- Calculate the measured received rate, $X_{recv}$, based on the packets received within the previous $R_{m-1}$ seconds. This is performed whether the feedback timer expired at its normal state or expired early due to a new lost packet. In the typical case, when the receiver is sending only one feedback packet per RTT and the feedback timer did not expire due to a new lost packet, then the time interval since the feedback timer last expired would be $R_{m-1}$ seconds.
• Prepare and send a feedback packet containing the information described in Section 3.2.2.

• Restart the timer to expire $R_m$ seconds.

If no data packet have been received since the last feedback was sent, then no feedback packet is sent, and
the timer is restarted to expire after $R_m$ seconds.

### 3.2.3 Detection of the Lost Packets

Obtaining an accurate and stable measurement of the loss event rate is of primary importance for Tiny-
Control. Loss rate measurement, performed at the receiver, based on the detection of lost packets from the
sequence numbers of arriving packets. If the receiver has not yet detected a lost packet, then the receiver
does not calculate the rate, but report a loss event rate of zero.

TinyControl assumes that all packets contain a sequence number that is incremented by one for each
packet that is sent. By using this, the client maintains a data structure that keeps track of which packets have
been arrived and which are missing, called a history.

The loss of a packet is detected by the arrival of at least NDUPACK packets with a higher sequence
number than the lost packet, for NDUPACK set to 3. The requirement for NDUPACK subsequent packets is
to make TinyControl more robust in the presence of reordering (note that UDP does not handle ordering). If
a packet arrives late in TinyControl, the late packet can fill the hole in its reception record, and the receiver
can recalculate the loss event rate.

Once it detects packet losses, the client needs to map the packet loss history into a loss event record,
where a loss event is one or more packet lost in an RTT. To perform this mapping, the client needs to know
the RTT to use, and this is supplied periodically by the server.

To determine whether a lost packet should start a new loss event or be counted as part of an existing loss
event, we need to compare the sequence numbers and timestamps of the packets that arrived at the receiver.
Assume the following:

- $S_{\text{loss}}$: the sequence number of a lost packet.
- $S_{\text{before}}$: the sequence number of the last packet to arrive, before any packet arrivals with a sequence
  number above $S_{\text{loss}}$, with a sequence number below $S_{\text{loss}}$.
- $S_{\text{after}}$: the sequence number of the first packet to arrive after $S_{\text{before}}$ with a sequence number above
  $S_{\text{loss}}$.
- $S_{\text{max}}$: the largest sequence number. Therefore, $S_{\text{before}} < S_{\text{loss}} < S_{\text{after}} \leq S_{\text{max}}$
- $T_{\text{loss}}$: the nominal estimated arrival time for the lost packet.
- $T_{\text{before}}$: the reception time of $S_{\text{before}}$.
- $T_{\text{after}}$: the reception time of $S_{\text{after}}$; thus, $T_{\text{before}} < T_{\text{after}}$.

For a lost packet, $S_{\text{loss}}$, we can interpolate its nominal arrival time at the receiver from the arrival time
of $S_{\text{before}}$ and $S_{\text{after}}$.

\[
T_{\text{loss}} = T_{\text{before}} + \frac{(T_{\text{after}} - T_{\text{before}}) \times \text{Dist}(S_{\text{loss}}, S_{\text{before}})}{\text{Dist}(S_{\text{after}}, S_{\text{before}})},
\]
where $\text{Dist}(S_A, S_B) = (S_A + S_{MAX} - S_B)\% S_{MAX}$ ($S_{MAX} = 2^{32}$).

If the lost packet $S_{old}$ was determined to have started the previous loss event, and we have just determined that $S_{new}$ has been lost, then we interpolate the nominal arrival time of $S_{old}$ and $S_{new}$, called $T_{old}$ and $T_{new}$, respectively. If $T_{old} + R \geq T_{new}$, then $S_{new}$ is part of the existing loss event. Otherwise, $S_{new}$ is the first packet in a new loss event.

### 3.2.4 The Size of a Loss Internal

After the detection of the first loss event, the receiver divides the sequence number space into loss intervals. If a loss interval, $A$ is determined to have started with sequence number $S_A$ and the next loss interval, $B$, started with $S_B$, the number of packets in loss interval $A$ is given by $(S_B - S_A)$. Thus, loss interval $A$ contains all of the packets transmitted by the server starting with $S_A$, but not including $S_B$.

We call current loss interval $I_0$, which is defined as the loss interval containing the most recent loss event. If that loss event started with packet sequence number $S_A$, and $S_C$ is the highest received sequence number so far, then the size of $I_0 = S_C - S_A + 1$.

The following example is a packet sequence explaining about loss event and loss interval. In the following figure, the symbol * stands for a packet loss. $T_{old}$ and $T_{new}$ are the times that the respective packet losses (indicated by *) are happened. After the first loss occurs at the beginning of the sequence, a few more losses continues. Those losses belong to the Interval 1 because there are within the boundary of $T_{old} + RTT$. However, the loss at $T_{new}$ starts new loss event because the arrival time of the loss at $T_{new}$ is larger than $T_{old} + RTT$.

As shown in the figure, the length of an interval can be larger than one RTT. This is so in this example that all data packets after the third loss in Interval 2 are received without loss.

### 3.2.5 Average Loss Interval and Loss Event Rate

To calculate the loss event rate, $p$, we first need to calculate the average loss interval. This is done using a filter that weights the $n$ most recent loss event intervals in such a way that the measured loss event rate changes smoothly. If the receiver has not yet seen a lost packet, then the receiver does not calculate the average loss interval.

Suppose there are $n$ loss intervals($0 \sim n-1$), where 0 stands for the most recent one. The corresponding weights $w_0$ to $w_{n-1}$ are calculated as:

If ($i < n/2$) $w_i = 1$ otherwise $w_i = 2 \times (n-i)/(n+2)$

In our implementation, we set $n=8$, which gives the values of $w_0$ to $w_7$ as 1.0, 1.0, 1.0, 1.0, 0.8, 0.6, 0.4, 0.2, respectively. Let the most recent loss intervals be $I_0$ to $I_k$, where $I_0$ is the current loss interval. We calculate the average loss interval $I_{mean}$ as follows:

\[
I_{tot} = \sum_{i=0}^{k} i \times w_i, \quad W_{tot} = \sum_{i=0}^{k} w_i, \quad I_{mean} = \frac{I_{tot}}{W_{tot}}
\]
The loss event rate, \( p \), is then, \( p = 1/I_{\text{mean}} \).

### 3.2.6 Receiver Initialization

The receiver is initialized by the first data packet that arrives at the receiver. Let the sequence number of this packet be \( i \). When the first packet is received:

- Set \( p = 0 \)
- Set \( X_{\text{recv}} = 0 \)
- Prepare and send a feedback packet
- Set the feedback timer to expire after \( R_{t} \) seconds.

### 3.2.7 Initializing the Loss History after the First Loss Event

The number of packets until the first loss cannot be used to compute the allowed sending rate directly, as the sending rate changes rapidly during this time. We assume that the correct data rate after the first loss is half of the maximum sending rate before the loss occurred. Client approximates this target rate, \( X_{\text{target}} \), by the maximum value of \( X_{\text{recv}} \) so far.

After the first loss, instead of initializing the first loss interval to the number of packets sent until the first loss, the client calculates the loss interval that would be required to produce the data rate \( X_{\text{target}} \), and uses this synthetic loss interval to seed the loss history mechanism.

The client does this by finding some value, \( p \), for which the throughput equation in Section 3.1.4 gives a sending rate within 5\% of \( X_{\text{target}} \), given the RTT, and the first loss interval is then set to \( 1/p \). For this purpose, we assume that the client knows the throughput equation 3.1.4.

### 4 Testing

If you test your program in your local machine (using IP 127.0.0.1), it is hard to experience packet losses. On the other hand, saturating University network to generate packet losses could be a problematic. Instead, you can intentionally generate packet losses in TinyControl server program. For example, if you have 10 packets to send, first pick a packet/packets to drop, and do not send the data. While sending data to receiver, you also record the starting sequence number and lost packet numbers. Make a graph showing rate adaptation according to packet losses. X-axis is sequence number and y-axis will be sender’s sending rate. Mark lost packets on the graph along with rate changes.

For data to send to receiver, you can use any file, such as text or binary files. As this protocol does not retransmit lost packets, you may want to use streaming data, such as mp3 file or movie file, which is tolerable with a few packet losses. In addition, for the UDP port number for server, you can freely choose your own.

### 5 Deliverables

You must deliver the source code that you have developed for this project, including a Makefile if necessary. You must also include a README.txt file that includes: (1) the names of the people in your team (including whether you take the 344 or the 444 version of the class), (2) instructions on how
to run your code and, if your work is not complete, (3) the missing parts. Do not include any binaries or files that you used to test your implementation. You should create a single compressed archive containing all your files. The name of the file should be: groupNumber_CS344Spring10.tar.gz (.zip) or groupNumber_CS444Spring10.tar.gz (.zip). You must send this file by email to: cs344hw@hinrg.cs.jhu.edu.