Reliable Transport Protocols

Haowei Zhou/ Shi Yan
haoweizh@buffalo.edu / shiyan@buffalo.edu

Contents

Academic Integrity

I，Haowei Zhou, have read and understood the course academic integrity policy.

Multiple logical timer on single hardware clock

Mathematical description

We only have one hardware clock but each packet in windows should have its own logical clock in selective-repeated algorithm, so I save the simulator time for each packet. The windows in SR protocol is slightly different from GBN protocol. In GBN protocol, I maintain a vector sorted by packet's sequence number. However, in SR protocol, I maintain a queue sorted by packet's sending time. So when simulator calls *A_timerinterrupt()*, I can ensure it is the first packet in this vector that times out. I just move it to the back of vector and reset its send time and recalculate the next packet by this formula:

Increment = increment - (now - A_windows.front().start_time)

In this way we can set multiple logical timer for each packets on single hardware clock.

Send

When a packet is sent, I buffer this packet in windows but not time for this packet because maybe other packets will be timed out earlier than this packet. What I do is save the sending time for this packet.

Receive

When a packet is received, I have to remove this packet from the queue, because we can only process the packet on either front of the queue or back of the queue, so I use vector to store all sent packets. So I can easily scan the vector and remove the packet with the same sequence number.

Timeout

When the simulator calls *A_timerinterrupt()*, It's definitely the first packet in this queue that times out because I sorted all sent packets by its sending time. So I just resend this timeout packet and move it to the back of the queue and reset the timeout according to the second timeout packet.

Data structure

Because I need to save the sending time with correspond packet, so I use this new data structure:

```
struct pkt_with_begin_time{
    pkt_with_begin_time(struct pkt pack,float start):packet(pack),start_time(start){}
    struct pkt packet;
    float start_time;
};
```

Timeout schema

In this project, I don't use fixed timeout for three protocols. I estimate the RTT time dynamically for each RTT by this formula:

$$EstimatedRTT = (1 - \alpha) \cdot EstimatedRTT + \alpha \cdot SampleRTT$$

EstimatedRTT is initialized as 20, also I set α to 0.125. Each time when the sender receives acknowledged packet, it will calculate the *SampleRTT*.

Different protocol uses different strategy to calculate *SampleRTT.* For ABT protocol and GBN protocol, we only need to record the simulator time when a packet was sent by A, when A receives acknowledgement from B, *SampleRTT* is current time minus send time. As for SR protocol, I record send time for each packet because each packet has its own logical time. So when A receives acknowledgement packet, we just need to minus correspond send time.

After we calculate *EstimatedRTT*, I use this formula to calculate timeout:

$$TimeoutInterval = EstimatedRTT + 4 \cdot DevRTT$$

And the *DevRTT* is calculated by this formula:

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot | SampleRTT - EstimatedRTT |$$

DevRTT is initialized as 0 and I set β as 0.25. According to the result, when there is no loss or corruption, the *EstimatedRTT* is around 20, that's why I initialized it as 20. When loss rate and corruption rate is not 0, the *EstimatedRTT* will fluctuate from 20 to 100. I think this method can set reasonable timeout and adjust it dynamically in unstable networking environment.

Experiments

Experiments 1

Incremental loss probability based on two different window sizes

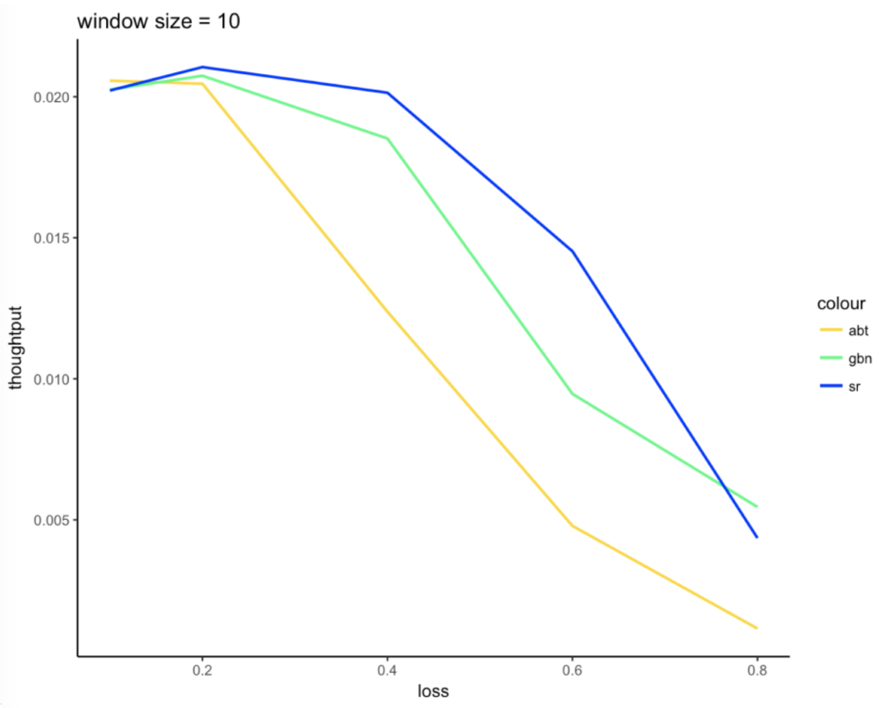
Protocol	Message	Corruption	Loss	Window
Alternative Bit Protocol	1000	0.2	0.1, 0.2, 0.4, 0.6, 0.8	10, 50
Go Back End	1000	0.2	0.1, 0.2, 0.4, 0.6, 0.8	10, 50
Selective	1000	0.2	0.1, 0.2, 0.4, 0.6, 0.8	10, 50

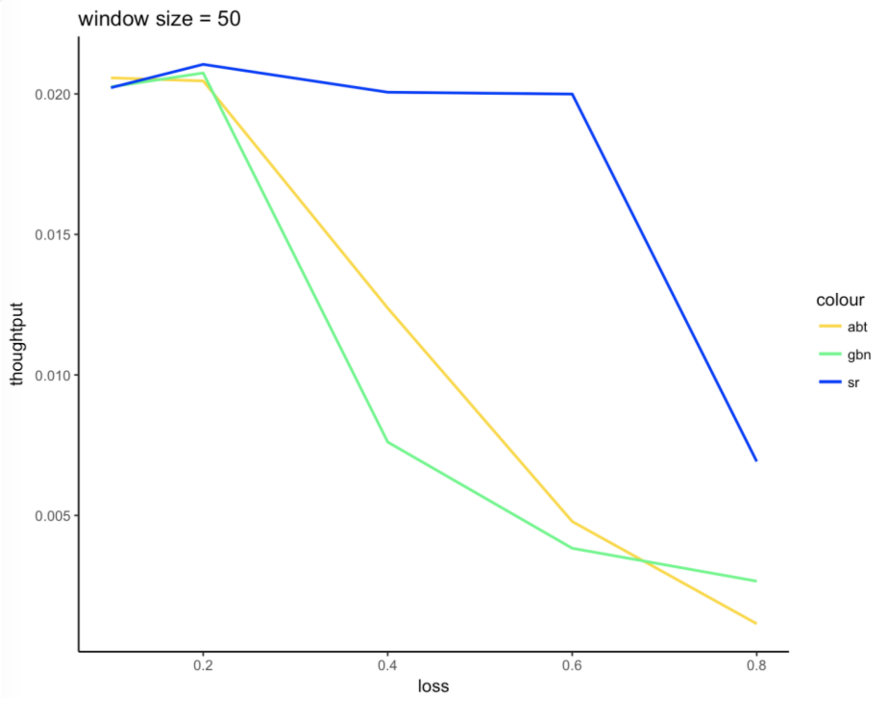
Table 1

The results show as we expected, the increase of corruption probability correspond to the trend of decrease of throughput.
From the attribute of RDT we know there is no influence on RDT if we change window size, these attribute was approved by our experiment. From follow two graphs, we could found that when window size = 10 and window size = 50, RDT keeps the same curve.

In general, GBD performs better than ABT, because ABT use too much time for wait, so GBN introduced sliding widow, however, for GBN, the receiver will discard any frame that does not have the exact sequence number it expects and will resend an ACK for the last correct in-order frame. So the side effect is that if the first packet always send failed, even all other packet send successfully, sender still need to resend all packet again. This phenomenon could observed in follow two graphs, when window size is larger and loss increased, the performance of GBN is worse than abt.

SR performs as we expected, for first experiment, SR performs best.


window size = 10
thoughtput
0.020
0.015
0.010
0.005
0.2
0.4
0.6
0.8
loss
colour
abt
gbn
sr


window size = 50
thoughtput
0.020
0.015
0.010
0.005
0.2
0.4
0.6
0.8
loss
colour
abt
gbn
sr

Experiments 2

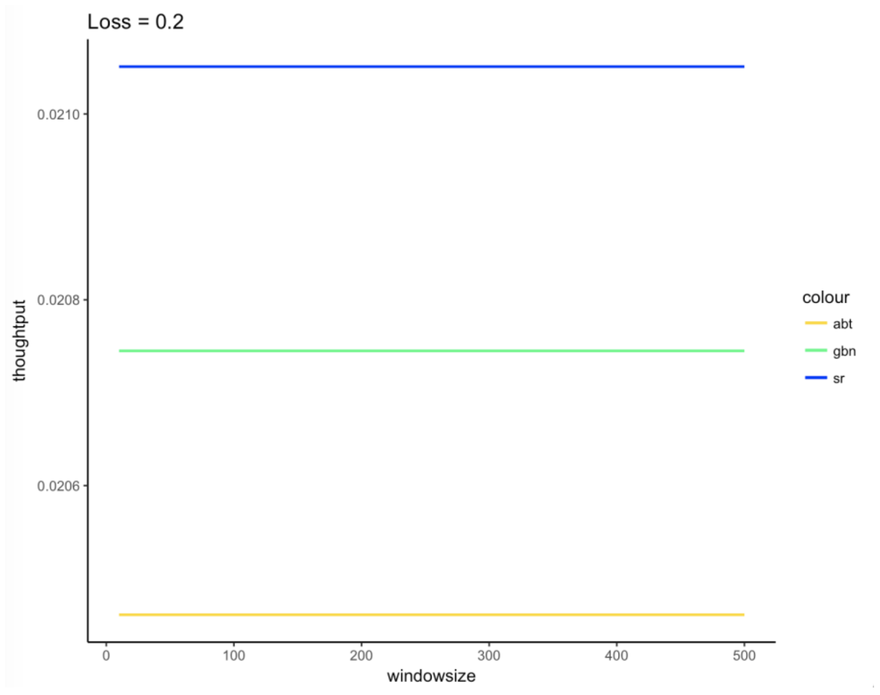
Incremental window size based on three different loss probabilities

Protocol	Message	Corruption	Loss	Window
Alternative Bit Protocol	1000	0.2	0.2, 0.5, 0.8	10, 50, 100, 200, 500
Go Back End	1000	0.2	0.2, 0.5, 0.8	10, 50, 100, 200, 500
Selective	1000	0.2	0.2, 0.5, 0.8	10, 50, 100, 200, 500

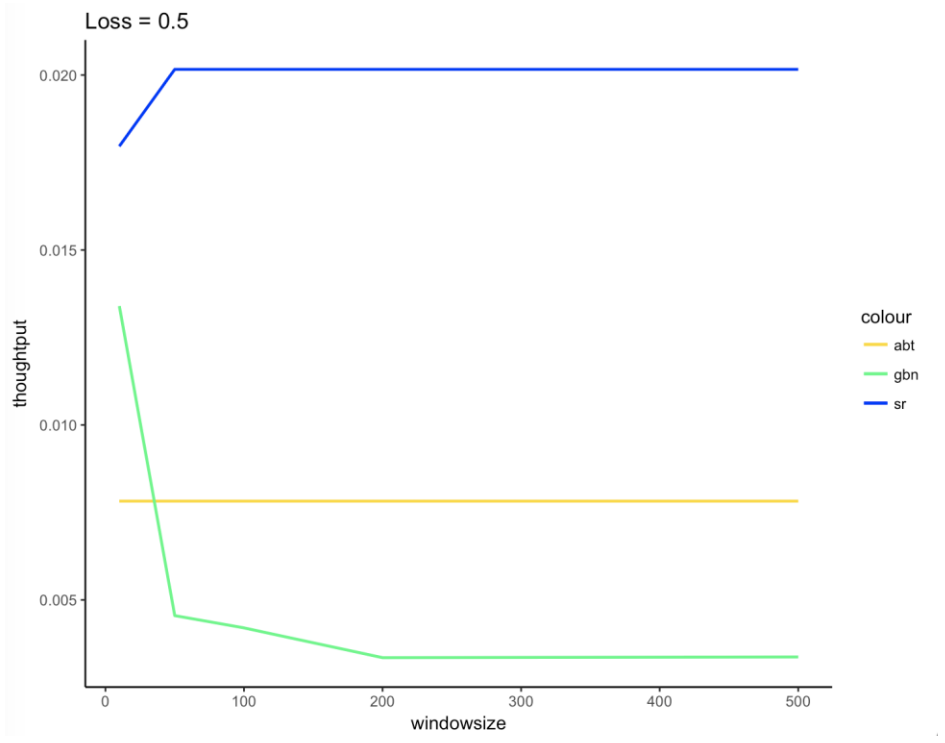
Table 2

Same with the first experiment, there is no influence on ABT when we only change window size.

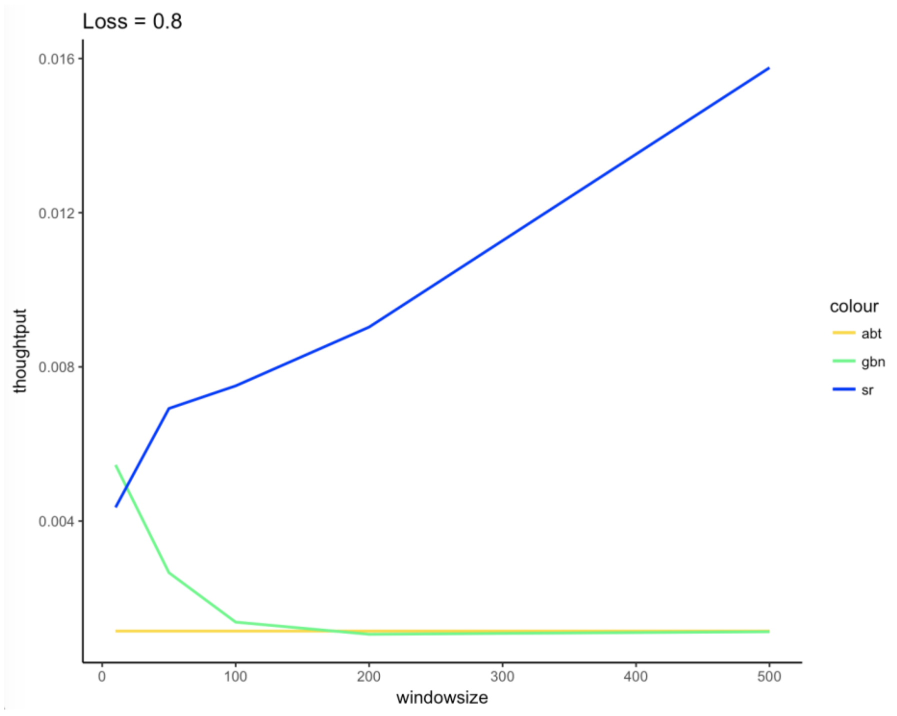
For loss = 0.2, ABT, GBN and SR have a steady performance, and also as we expected, SR performs best, next is GBN and last is ABT.



For loss = 0.5, at fist GBN performs better, however, with the increase of window size, ABT performs better than GBN, but still worse than SR, we think it may because the drawback of GBN, although window size larger than previous, it means that one end could receive more packets, at the same time, it also means that if first of packets fail to send, sender need to retransmit more packet, in this situation, even window size larger, efficiency not good enough. It means that we cannot just to augment windows size to pursue a good performance when we use GBN protocol.



For loss = 0.8, it’s very interesting, when window size become larger, SR performs better, for GBN and ABT, at first, GBN performs better than ABT, then GBN has almost same performance with ABT. The phenomenon happens on SR maybe because we used dynamic update timeout value, from this graph, we could found that SR left GBN and ABT far behind.



From above experiment, we could found that if we use one protocol, if we want to pursue a better performance, we couldn't only enlarge the window size, we need to consider the side effect brought by a larger window, and also for different circumstance we need to use different protocol, there is no panacean.