Phyo Thiha March 10, 2013

CSC457 Reliable Data Transfer Protocol Project Write up

Initial values:
#define WINDOW_SIZE 15
#define TIME_OUT_TIME 0.4
The time out time actually depends on the average Round Trip Time but if it is about 0.1 seconds one way, then 0.4 seems work best for my implementation. WINDOW_SIZE is the size of the window on the sender side. The performance is heavily dependent on this. If each message size is big then bigger window size generally does better.

I'm assuming about a 100 characters per message here.

Header:

size	SEQ#	ACK#	checksum	data

The packet size is at most 64 bytes
(Header size - 7 bytes)
size - 1 byte
SEQ# - 2 bytes
ACK# - 2 bytes
checksum - 2 bytes
data - 57 bytes (at most)

Size:
a byte could represent 256 different values but for this program this will be at most 64. If it is above this, value then the packet must be corrupted and will be discarded.

SEQ# and ACK#
The SEQ and ACK could have been much smaller. Actually these two could be combined and only use 1 byte because the window size is only 15, but for flexibility it is nice to have at least a dedicated byte for each field.

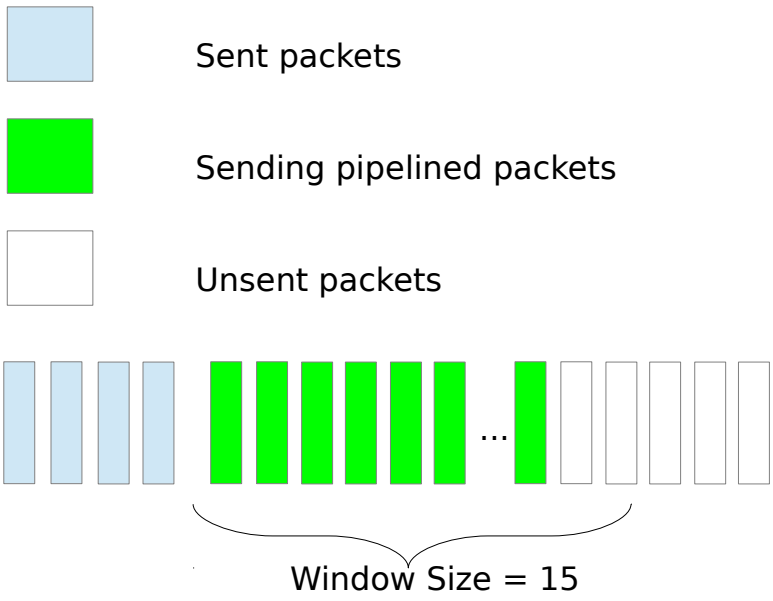
I should note that this program does not require separate SEQ and ACK field because it could be shared.

Checksum:
16 bit checksum is used. This means that the check could fail 1 in 65636 times or more when the packet is corrupted.

Reliable Transfer:
My implementation is a data transfer that TCP uses.

SENDER:
The sender side will send the packets that aren't ACKed yet, but the ACKs are cumulative so if I get an ACK that is bigger than expected (first one in the window), it simply slides the window to the right until the packet with the ACKed SEQ#. If it gets the ACK that is expected then it slides the window one to the right.

To do this I made a buffer on the sender side because the user can send the message as fast as they want but the sender might not have received enough ACKs to flush out the previous message. So, the buffer on the sender side is a linked list of packets. As soon as it gets a message from the upper layer, it breaks them into packets and put them into linked list.



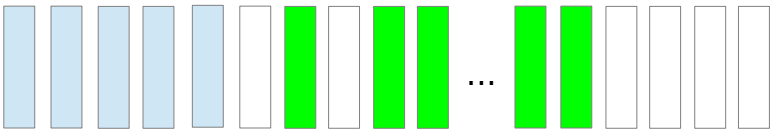
RECEIVER:
The receiver also has a buffer. When the receiver gets an out of order packet, it buffers that and waits for the next expected packet. When it receives an in order packet, it will coalesce the packets, make it into a message, and send that message to the upper layer.

The buffer on the sender side was a linked list but the receiver side only has to be an array of size WINDOW_SIZE. So the array is a circular buffer and only buffers at most WINDOW_SIZE many.

Received packets

Buffered packets

Packets not yet received



Sample OUTPUT:

$ rdt_sim 1000 0.1 100 0.1 0.1 0.1
usage: rdt_sim <sim_time> <mean_msg_arrivalint> <mean_msg_size>
<outoforder_rate>
<loss_rate> <corrupt_rate> <tracing_level>
$ rdt_sim 1000 0.1 100 0.1 0.1 0.1 0
Reliable data transfer simulation with:
simulation time is 1000.000 seconds
average message arrival interval is 0.100 seconds
average message size is 100 bytes
average out-of-order delivery rate is 10.00%
average loss rate is 10.00%
average corrupt rate is 10.00%
tracing level is 0
Please review these inputs and press <enter> to proceed.

At 0.00s: sender initializing ...
At 0.00s: receiver initializing ...
At 1000.52s: sender finalizing ...
At 1000.52s: receiver finalizing ...

Simulation completed at time 1000.52s with
1017520 characters sent
1017520 characters delivered
143589 packets passed between the sender and the receiver
Congratulations! This session is error-free, loss-free, and in order.

Stats:
I wrote a script that tests each parameters a hundred times.

rdt_sim 1000 0.1 100 0.1 0.1 0.1 0
Most of the time it ends in 1000-2000 seconds but there are times that could go very slow or the packet corruption not getting detected by the checksum.

rdt_sim 1000 0.1 100 0.5 0 0 0 (50% out of order only YES 50% not a typo)
About 1000seconds every time.
This implementation is very strong against out of order packets.

rdt_sim 1000 0.1 100 0 0.2 0 0 (20% lost packets)
The result ranges in [1000,2000] seconds most of the time. But there are cases where it takes about 8000 seconds.

rdt_sim 1000 0.1 100 0 0.2 0 0 (20% corrupt packets)
In [1000,2000] seconds but 1 time it did not stop or was very slow in 150 times.

Strength:
As the stats show, this implementation is very good at handling out of order packets.

Known Limitations:
It sometimes does not halt or takes a lot of time to recover when there are corruptions.