Reliable Transport

Brandt Elison
Joe Eklund

February 20, 2016

1 Implementation

Our implementation of TCP reliability was a modification of the bene network simulator. However, before we began writing any code to implement reliability we drew a diagram outlining the steps required to implement TCP. Figure 1 is a digitized version of this outline.

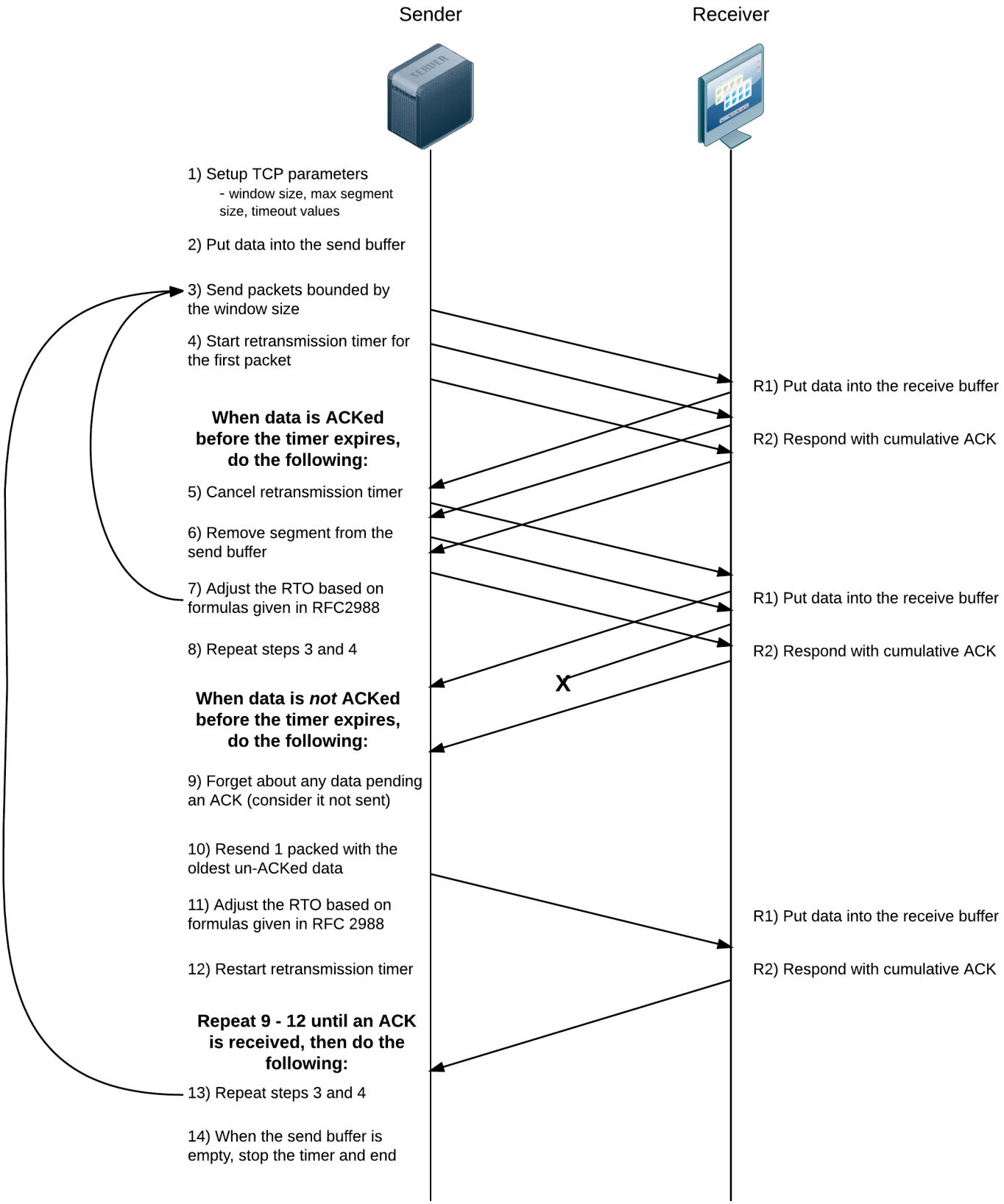
The following numbers correspond to the step numbers shown in Figure 1 and in which method(s) they were implemented. These methods can be found in tcp.py in the source code.

1. TCP Constructor
2. `send(data)`
3. `send_max()`
4. `send_packet(data, sequence)`
5. `handle_ack(packet)`, `cancel_timer()`
6. `handle_ack(packet)`
7. `adjust_timeout(sample_rtt)`
8. `send_packet(data, sequence)`, `send_max()`
9. `retransmit(event)`
10. `retransmit(event)`
11. `retransmit(event)`
12. `send_packet(data, sequence)`
13. `send_packet(data, sequence)`, `send_max()`
14. `send_max()`

R1. `handle_data(packet)`

R2. `send_ack(time_stamp)`

Figure 1: The diagram of our reliability outline.



We added a dynamic retransmission timer that uses an exponetial weighted moving average (EWMA) for the timer value and the variance. We implemented our dynamic timer in tcp.py by creating a function called `adjust_timeout(sample_rtt)` which is invoked every time the server receives an ACK for un-ACKed data. We also modified `retransmit()` to double the timeout value when the function is invoked. This is in keeping with RFC 2988.

2 Test Results

After implementing reliable transport, we ran tests on two files called text.txt and internet-architecture.pdf. We wrote a script that copies the test files from one directory to another and then diffs them to verify that our implementation of TCP reliability functions correctly.

For each file we tested loss rates ranging from 0.0 to 0.5 on both a fixed one second timer and dynamic retransmission timer. The dynamic timer uses an EWMA for the timer value and variance. For test.txt we recorded the average time it took to transfer the file over 1000 tests. For internet-architecture.pdf we averaged tranfer times over 100 tests. The results of these tests are shown in tables 1 and 2.

Table 1: Test results for transfering test.txt

Loss Rate	Fixed Timer (sec)	Dynamic Timer (sec)
0.0	0.0832	0.0832
0.1	1.175	1.677
0.2	2.312	7.735
0.5	16.078	2,067,467.769

Table 2: Test results for transfering internet-architecture.pdf

Loss Rate	Fixed Timer (sec)	Dynamic Timer (sec)
0.0	1.084	1.084
0.1	37.0	48.093
0.2	71.484	162.994
0.5	389.159	391,497,109.508

Snippet 1: Dynamic retransmission output

```
1 Starting timer with timeout of: 3.0
2 Received ACK with RTT of: 0.0208
3 Timeout adjusted from 3.0 to 1.0208
4 Starting timer with timeout of: 1.0208
5 Received ACK with RTT of: 0.0216
6 Timeout adjusted from 1.0208 to 1.0209
7 Starting timer with timeout of: 1.0209
8 **Timer expired. Doubled timeout to 2.0418
9 Starting timer with timeout of: 2.0418
10 Received ACK with RTT of: 0.0208
11 Timeout adjusted from 2.0418 to 1.0208875
12 Starting timer with timeout of: 1.0208875
13 Received ACK with RTT of: 0.0216
14 Timeout adjusted from 1.0208875 to 1.0209765625
```

```
15 Starting timer with timeout of: 1.0209765625
16 Received ACK with RTT of: 0.0224
17 Timeout adjusted from 1.0209765625 to 1.02115449219
18 Received ACK with RTT of: 0.0216
19 Timeout adjusted from 1.02115449219 to 1.02121018066
```

As evidenced from the output above, the timeout converges from 3.0 seconds to about 1.02 seconds. This shows that the dynamic retransmission timer is in fact converging to the right value over time. Lines 7 and 8 of Snippet 1 show that exponential backoff works as expected.

Tables 1 and 2 show that with a loss rate of 0.0, both the fixed timer and dynamic timer transmit files in the same amount of time. This makes sense because we never have have to retransmit packets and the timer never expires. The transmission times are relatively short from the 0.0 to 0.2 loss rate range for both types of timers. However, the time it takes to transmit with a dynamic timer of a loss rate of 0.5 is untenable; the transmission of internet-architecture.pdf took 391,497,109.508 seconds (12.41 years) which is clearly unacceptable.

3 Experiments

Our experiments consisted of sending the file internet-architecture.pdf on a network with the following parameters:

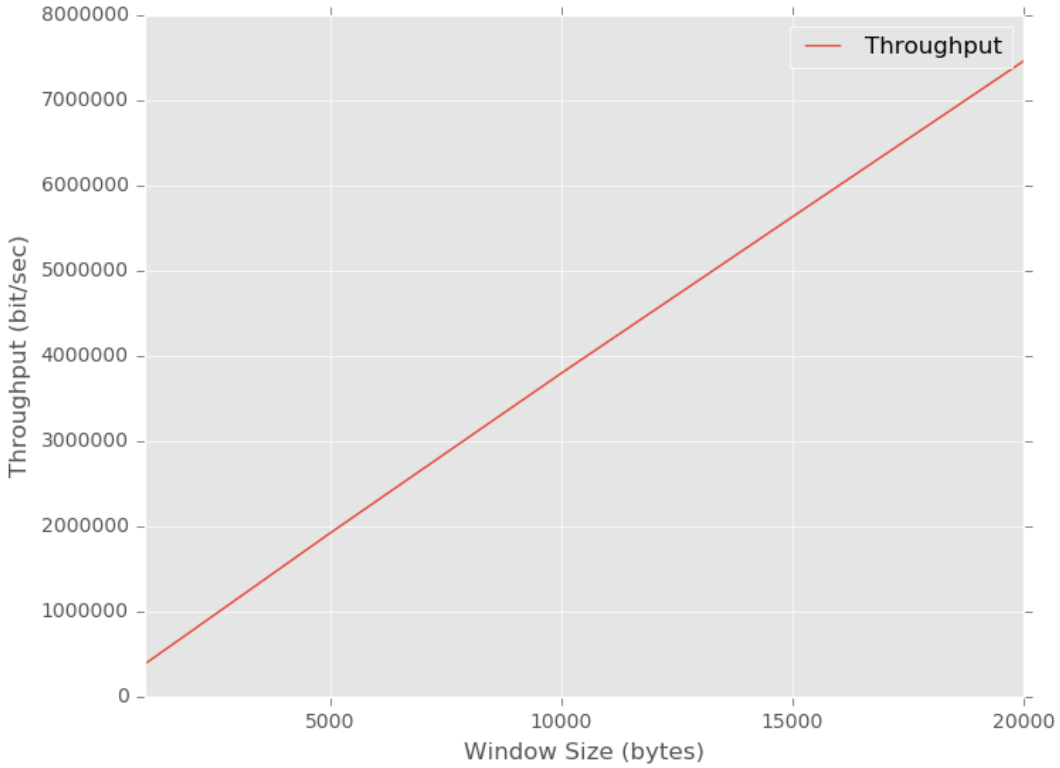
- Link speed n_1 to n_2: 10 Mbps
- Link speed n_2 to n_3: 10 Mbps
- Propagation delay n_1 to n_2: 10 ms
- Propagation delay n_2 to n_3: 10 ms
- Queue Size: 100 Packets
- Loss Rate: 0.0

We ran 6 tests, changing the window size each time in the range of 1,000 to 20,000 bytes (see Table 3 Column 1 for the exact window sizes we used). Figures 2 and 3 respectively plot the Throughtput and Queueing Delay columns of Table 3 vs each of the window sizes in Table 3.

Table 3: Test results for transfering internet-architecture.pdf

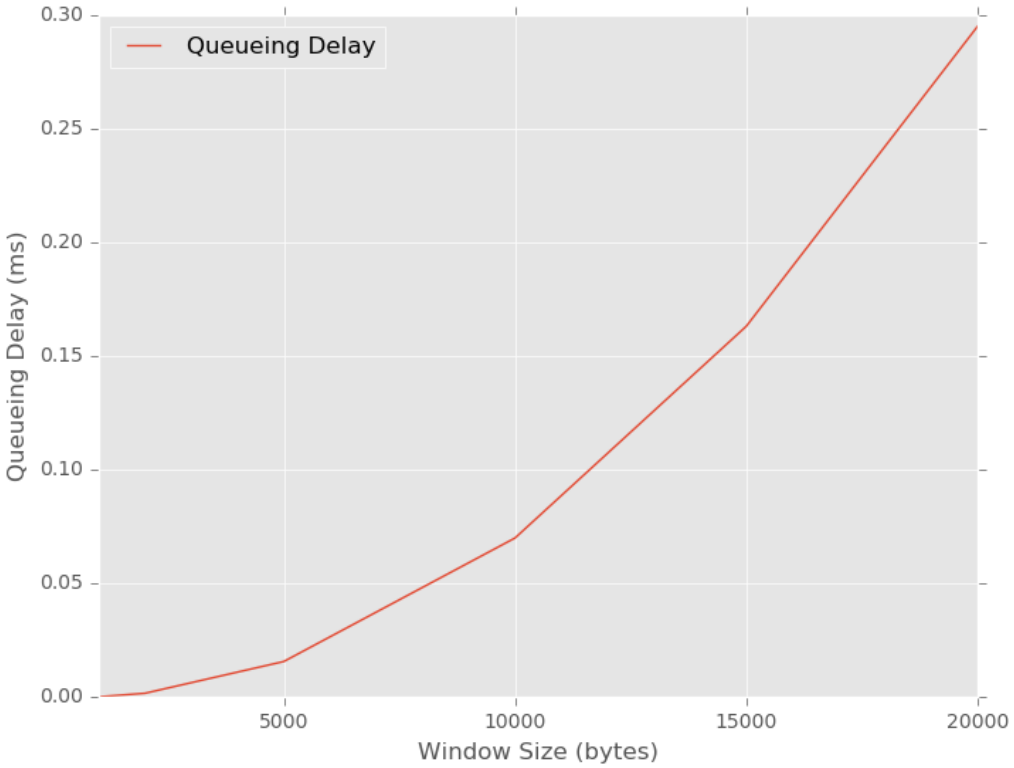
Window Size (bytes)	Transmission Time (sec)	Throughput (bits/sec)	Average Queueing Delay (ms)
1,000	10.711	384,270.68	0.0
2,000	05.366	767,079.34	0.001553
5,000	02.145	1,918,762.49	0.01553
10,000	01.084	3,795,738.90	0.06990
15,000	00.731	5,632,279.53	0.1631
20,000	00.552	7,462,002.55	0.2951

Figure 2: Window Size vs Throughput for transmitting internet-architecture.pdf



By looking at Figure 2 we can see our throughput varies linearly with our window size. This makes sense because elements that would usually prevent this linear relationship, such as packet loss or low bandwidth on the link, are not factors in this situation. Our loss rate was set to 0.0, so there was no simulated packet loss to reduce throughput; our link speed was 10 Mbps, which gives plenty of bandwidth to handle the window size of 20 kB. Therefore, doubling the window size does simply double the rate at which the receiver gets data a.k.a. doubles the throughput.

Figure 3: Window Size vs Queueing Delay for transmitting internet-architecture.pdf



By looking at Figure 3 we can see our average queueing delay looks like it varies exponentially with window size. Again this result makes sense because the queueing delay builds slowly at first while the link is able to handle the volume of packets, but as the window size gets larger, the link will have to queue more of the packets until all of the aditional packets are queued.

While the exponential increase makes the queueing delay seem substantial, it should be remembered that the unit of measurement in Figure 3 for queueing delay is milliseconds. Even though the queueing delay looks large for a window size of 20,000 bytes, the delay is actually only 0.2951 ms per packet. This is because our link is so fast and our loss rate is 0.0.