Documentation

This software has the Stop-and-wait algorithm, as well as the sliding window algorithm implemented. The software creates a UdpSocketObject by instantiating sock. Then allocates a message that is 160-bytes in size.

Performance of the UDP point-to-point communication is evaluated using the following three different cases to test with.

1. Unreliable transfer
2. Stop-and-wait
3. Sliding Window

Case 1 (Base UDP)

Frames are sent as fast as possible from the client to the server. Frame detection and sequencing are ignored by the server. The client initiates a transfer of a sequence of messages with the sequence number prepended in the header. The server will then be expecting the same number of messages that the client is claiming to have sent. The server will actually hang if the number of frames don't add up to account for all that are expected. This works this way by design.

Case 2 (Stop and Wait)

If the sender does not receive the acknowledgment after a certain amount of time, the sender will expire the frame that was sent and attempt retransmission of that frame.

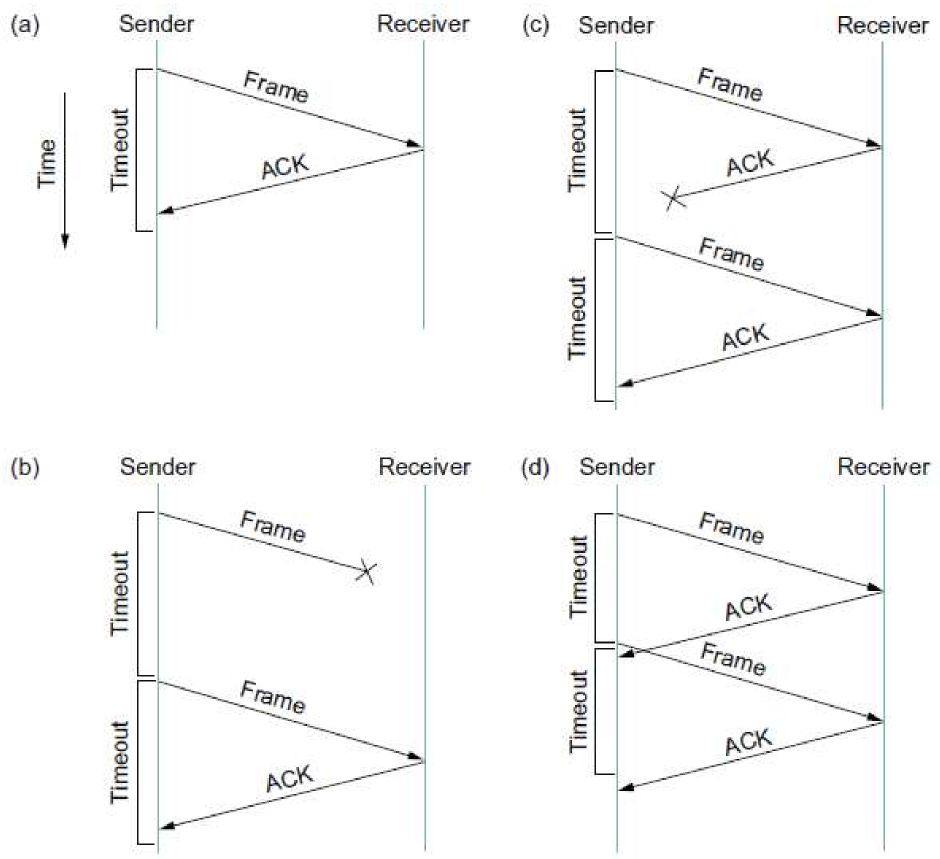


Figure 1: Timeline showing four different scenarios of the Stop-and-wait algorithm. (from our Textbook)

Figure 1 illustrates the situation when the acknowledgment (ACK) is received before the timer expires. Both (b) and (c) illustrates the situation where the original frame and the ACK never makes it to the destination. (d) illustrates the situation where the timeout expires before the ACK gets back to the sender, assuming for sure it will get back and the original frame is resent by the sender.

Both events (c) and (d) illustrate the situation when the sender transmits a frame and the receiver receives it and sends the ACK back to the sender, but the ACK is either takes too long to arrive to the sender or it simply gets lost in transit. In both of these situations, the sender times out and retransmits the frame again, but the receiver will get confused and assume that it is actually the next frame in sequence, since it actually did receive the frame and did send an ACK back to the previous original frame. This issue can result in duplicate copies of the frame being delivered. To work around this potential issue, the header in the Stop-and-wait protocol will usually include single bit sequence number.

The implementation of the Stop-and-wait protocol algorithm utilizes a sequence number that keeps track of the count from the first frame to the last frame to be sent. Each sequence number is prepended to the header of the frame. For every single frame that is sent by the client, the server receives the message and copies the sequence number contained in the message to an ACK, and then returns it to the client. If the client does not receive the ACK right away, it starts a timer that will wait up to 1500 usec. If the timer reaches 1500 usec, the client expires and retransmits the original message again.

The main limitation of the Stop-and-wait algorithm is that it makes it possible for the sender to have only a single outstanding frame out on the link at any given time, and the link may have a much higher capacity. The workaround to this is to have multiple frames in transit while waiting for an ACK.

Case 3 (Sliding Window)

The client assigns a sequence number to each frame that is sent and will continue sending more messages as long as the number of message in transit doesn't exceed the set window size. This type of transmission allows for higher bandwidth throughput than the Stop-and-wait algorithm.

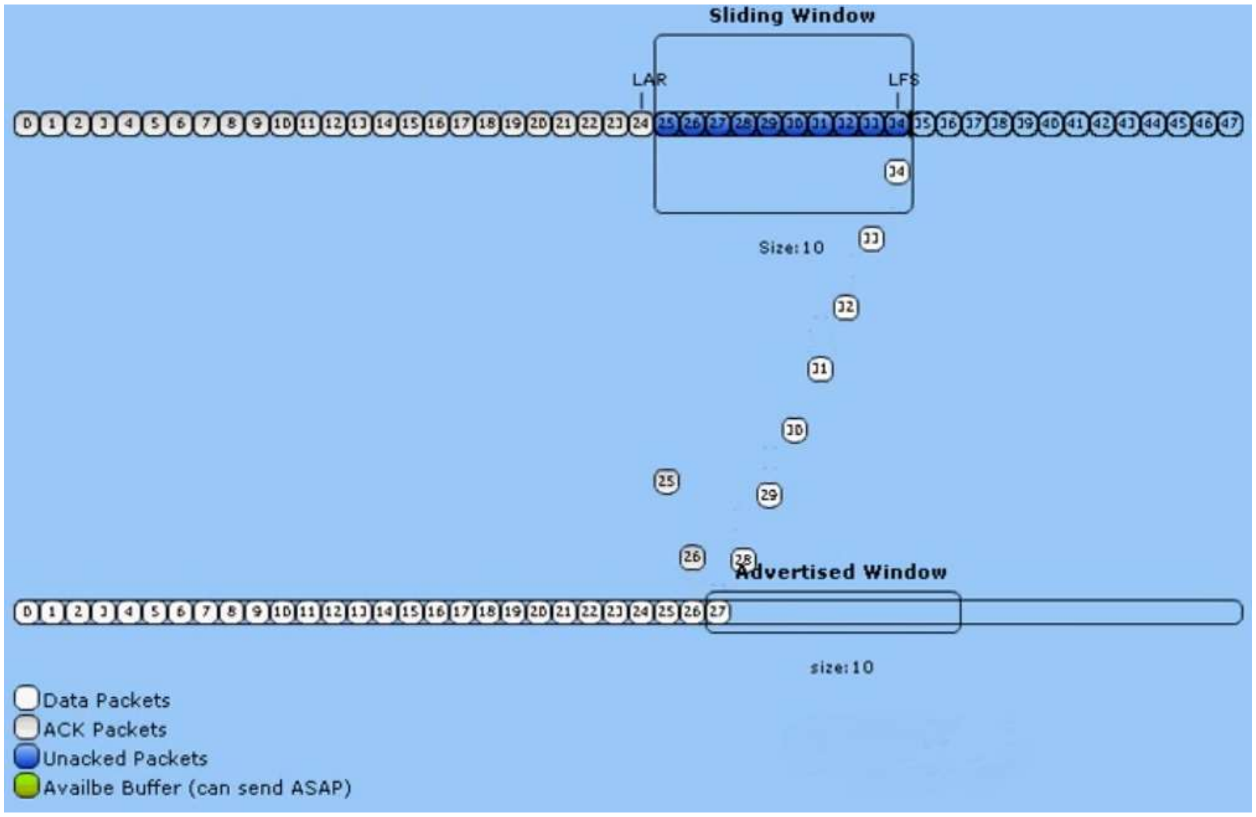


Figure 2 (Sliding Window Protocol) – Found on Youtube: https://www.youtube.com/watch?v=TRTglg7D8oI

Execution output

```
Window Windows Size ( 1 Elasped time = 5093907
retransmits = 0
Window Windows Size ( 2 Elasped time = 2384050
retransmits = 0
Window Windows Size ( 3 Elasped time = 1423449
retransmits = 0
Window Windows Size ( 4 Elasped time = 1386933
retransmits = 0
Window Windows Size ( 5 Elasped time = 1412530
retransmits = 0
Window Windows Size ( 6 Elasped time = 1419085
retransmits = 0
Window Windows Size ( 7 Elasped time = 1407191
retransmits = 0
Window Windows Size ( 8 Elasped time = 1409724
retransmits = 0
Window Windows Size ( 9 Elasped time = 1397218
retransmits = 0
Window Windows Size ( 10 Elasped time = 1395060
retransmits = 0
Window Windows Size ( 11 Elasped time = 1386126
retransmits = 0
Window Windows Size ( 12 Elasped time = 1397365
retransmits = 0
Window Windows Size ( 13 Elasped time = 1418849
retransmits = 0
Window Windows Size ( 14 Elasped time = 1411166
retransmits = 0
Window Windows Size ( 15 Elasped time = 1411388
retransmits = 0
Window Windows Size ( 16 Elasped time = 1391996
retransmits = 0
Window Windows Size ( 17 Elasped time = 1413714
retransmits = 0
Window Windows Size ( 18 Elasped time = 1400436
retransmits = 0
Window Windows Size ( 19 Elasped time = 1398245
retransmits = 0
Window Windows Size ( 20 Elasped time = 1428184
retransmits = 0
Window Windows Size ( 21 Elasped time = 1409560
retransmits = 0
Window Windows Size ( 22 Elasped time = 1420331
retransmits = 0
Window Windows Size ( 23 Elasped time = 1415531
retransmits = 0
Window Windows Size ( 24 Elasped time = 1429188
retransmits = 0
Window Windows Size ( 25 Elasped time = 1398386
retransmits = 0
Window Windows Size ( 26 Elasped time = 1402405
retransmits = 0
Window Windows Size ( 27 Elasped time = 1421411
retransmits = 0
```

```
Window Windows Size ( 28 Elasped time = 1416263
retransmits = 0
Window Windows Size ( 29 Elasped time = 1407691
retransmits = 0
Window Windows Size ( 30 Elasped time = 1392417
retransmits = 0
finished
```

Performance evaluation

Client output averages are given through five complete tests using the three cases. For each case, a 1460 byte message is constructed with a sequence number prepended in the header. Each test will send 20,000 messages from a client running on a UWB Linux computer to a server running on another UWB Linux computer over a 1000BaseT switched LAN. For each test, the average transfer time is given, with the exception of case 3 where averages of each window size are given. Full program output for all test cases are included in the test1.txt, test2.txt, and test3.txt files.

Case 1 – Base UDP

Send average: 244,764 usec

Of all the cases, the UDP flood was the fastest with very consistent times across all tests. While all five tests completed with all messages received by the server, there was no guarantee that the server received the messages in the same order that the client sent them. In the event that even a single message was lost, no retransmission is possible and the server will hang (by design) because of the missing frame.

Case 2 – Stop-and-Wait

Send average: 4,274,732 usec

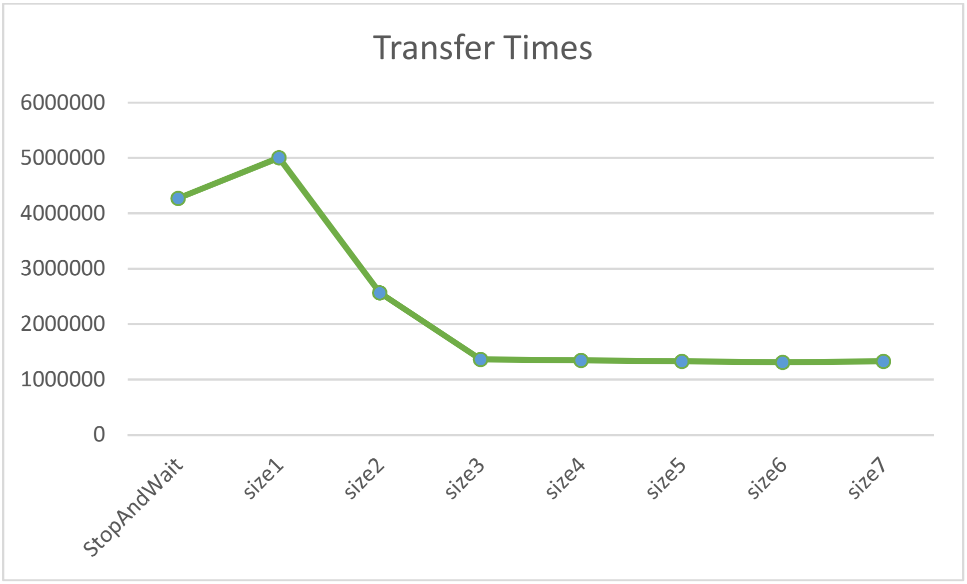
This case also proved to yield very consistent times across all tests, however stop-and-wait was 17x slower than a UDP flood, illustrating that sending data as a burst stream makes a large difference. While these particular tests did not require any message retransmissions, this implementation has shown to be very reliable at detecting improper sequencing and ensures all messages are eventually received in the correct order. In the event that a frame is lost or arrives out of sequence, the server will drop improperly sequenced message and force the client to resend. Testing on a reliable network showed that dropped frames occasionally happened, but were relatively few.

Case 3 – Sliding Window

Window	Window Size (1)	Windows Size (2)	Windows Size (3)	Windows Size (4)	Windows Size (5)	Windows Size (6)	Windows Size (7)
Average (usec)	5,005,722	2,566,522	1,365,388	1,346,831	1,330,059	1,312,952	1,328,369

Table 1: Sliding Window Results (More in the additional files provided.)

The sliding window algorithm had results that ranged 1.2x slower than stop-and-wait to 3.2x faster than stop-and-wait, where times were directly affected by window size. Table 1 only shows up to window size 7, and while up to size 30 was tested, times were consistent beyond window size 5 and are omitted from the table. Complete results for all five tests with all window sizes can be found in the file test3.txt.



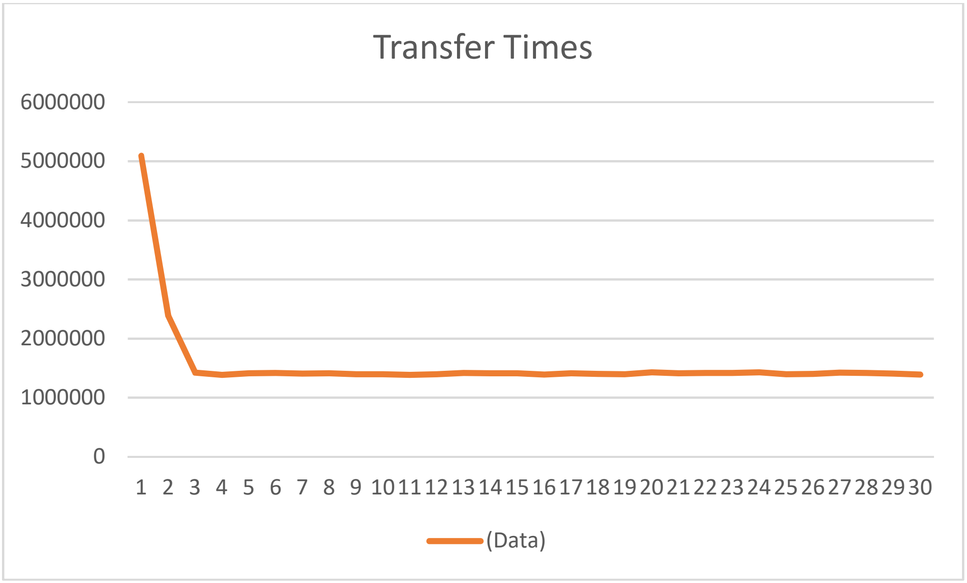


Figure 3: Stop-and-Wait vs. Sliding Window Algorithm Window Size time comparison

For both stop-and-wait and sliding window, when the window size is 1, the effective transmission rate should be similar, but it is not. One possible reason that sliding window is slower in this particular test is the additional overhead associated with the sliding window algorithm. The implementation is more complex for both the client and server and the additional operations have a cumulative slowing effect.

This penalty starts to disappear as network efficiency comes into play with having multiple frames in transit at once. When window size is 2, the sliding window case is almost 2x faster than a window size of 1 and is 1.7x faster than stop-and-wait. Increasing the window size to 3 appears to find a new threshold where the network is no longer a factor. Unfortunately, it's difficult to tell what becomes the new bottleneck because the increasing window size should make times continue to decrease toward UDP packet flood, but do not.

Retransmissions were not encountered during final testing, but the sliding window size 1 and 2 were subject to the most message timeouts. Additionally, the total number of retransmissions was substantially higher than counts encountered in stop-and-wait or any other sliding window size. Subtle implementation changes to the sliding window client or server could still result in the completion of sending all messages, but encounter thousands of retransmits—or none, depending on what was done. This volatility leads me to believe that efficiency in algorithm implementation has as much effect as network stability for this particular protocol.