Program2

Aika Usui

# Diagram

A close up of a map

Description automatically generated

A close up of a logo

Description automatically generated

A close up of a map

Description automatically generated

A screenshot of a cell phone

Description automatically generated

## Stop-and-wait algorithms

Follows the stop-and-wait algorithm in that a client writes a message sequence number in message[0], sends message[] and waits until it receives an integer acknowledgment from a server, while the server receives message[], copies the sequence number from message[0] to an acknowledgment, and returns it to the client.

#### Client

Client sends a message to server for max times and timer starts. While waiting, it checks if timeout occurs or not. (if the client cannot receive an ack immediately, timeout occurs.) If timeout occurs, it will send a message again and increment retransmit value. Otherwise it receives ack from server and increments sequence if the sequence is equals to the ack. This method returns the retransmit time.

#### Server

Server receives a message from client for max times. If the ack matches to the expected sequence, send back the sequence as an ack, and increment sequence by one. If ack does not equals to the expected sequence, break the do while loop.

## Sliding-window algorithms

The client keeps writing a message sequence number in message[0] and sending the message[] as long as the number of in-transit messages is less than a given window size. The server receives message[], records the sequence number in its array and returns as its acknowledgment the minimum sequence number of messages it has not yet received (called a cumulative acknowledgment).

#### Client

Client sends a message to server for max time as long as the ack from server is within the range. If so, client sends a message and keeps receiving ack from server. If the ack is equivalent to the ackSeq, increments ackSeq. AckSeq is the cumulative ack. If the ack from server is not within the range, check if timeout occurs or not. If timeout occurs, client resends message to server and increments retransmits by one. If not, keep polling and receiving ack from server. If cumulative ack is smaller than or equals to ack from server, updates cumulative ack. If not, resend the message and increments retransmits. This method returns the retransmit time.

#### Server

Server creates an array of boolean that holds if the message is received or not. (Initialize as false.) After that, server keeps receiving a message from client for max times. If ack from client and expected sequence are the same, mark array[sequence] as true and scan the array to find last true element to find cumulative ack value. Afterwards, server advances sequence to the last consecutive true element’s index + 1 and update the ack value. Then, send ack back to client. If ack from client and expected sequence are not the same, mark the message as received. Then, send ack back to client.

# Performance Result

Packet size: 1460 bytes, Bandwidth: 1Gbps, timeout = 1500 us, Repetitions = 20000. Throughput = size/total time

Conducted the program under these css lab. Client: csslab11.uwb.edu, Server: csslab10.uwb.edu

## Stop-and-wait

Here are the average values for following variables.

|  |  |  |
| --- | --- | --- |
| Elapsed time (us) | # of Retransmits | Throughput (Bpus) |
| 1739304.75 | 1 | 16.788 |

I experienced retransmits one in 5 times. It seems to be stable way to transfer packets.

## Sliding-window

Here are the average values for following variables.

|  |  |  |  |
| --- | --- | --- | --- |
| Window size | Elapsed time (us) | # of Retransmits | Throughput (Bpus) |
| 1 | 2428725.5 | 0 | 12.023 |
| 2 | 1277761 | 0 | 22.852 |
| 3 | 1018699 | 0 | 28.664 |
| 4 | 901770.5 | 0 | 32.381 |
| 5 | 887909 | 0 | 32.886 |
| 6 | 862389 | 0 | 33.589 |
| 7 | 835937 | 0 | 34.931 |
| 8 | 828020 | 0 | 35.265 |
| 9 | 827528 | 0 | 35.286 |
| 10 | 812540 | 0 | 35.937 |
| 11 | 818844 | 0 | 35.660 |
| 12 | 823729 | 0 | 35.449 |
| 13 | 802953 | 0 | 36.366 |
| 14 | 803884 | 0 | 36.324 |
| 15 | 807373 | 0 | 36.167 |
| 16 | 802656.5 | 0 | 36.379 |
| 17 | 822056 | 0 | 35.521 |
| 18 | 800232.5 | 0 | 36.489 |
| 19 | 799021.5 | 0 | 36.545 |
| 20 | 801670.5 | 0 | 36.424 |
| 21 | 796902 | 0 | 36.642 |
| 22 | 817542 | 0 | 35.717 |
| 23 | 796882.5 | 0 | 36.643 |
| 24 | 803868 | 0 | 36.324 |
| 25 | 844632.5 | 0 | 34.571 |
| 26 | 800022 | 0 | 36.499 |
| 27 | 801585 | 0 | 36.428 |
| 28 | 815067.5 | 0 | 35.825 |
| 29 | 826882.5 | 0 | 35.313 |
| 30 | 797412 | 0 | 36.618 |

The table and the graph show the changes in elapsed time, retransmits, and throughput over the change in window size. As the window size increases, the speed and throughput get increase. Compare window size 1 and size 30, its through put is almost three times bigger than its of size 1. Up to window size 7, I observed the significant change in the throughput value. After that, although I observed one significant drop on window size 25, throughput seems to be about the same value and became stable. (between 34-36)

## Comparison on Stop-and-wait and Sliding-window

Since Sliding-window with window size of 1 is the same as stop-and-wait, compare stop-and-wait and sliding-window in terms of speed, stop-and-wait seems to be faster. However, as the window size increase, sliding window is faster than stop-and-wait and its throughput is much greater than stop-and-wait. Also, I never experienced retransmits in sliding-window. Thus, I think I can conclude that sliding-window is much safer way to transfer packets and more faster if you use large number of window size.

## Discussions on the retransmission rate

Since I didn’t experience a lot of retransmissions over the execution, I cannot calculate the reasonable retransmission rate. However, as far as I tested, stop-and-wait has a higher rate, about 20% for retransmission and sliding-window has 0%. I realized this rate changes on each host that I use to execute.