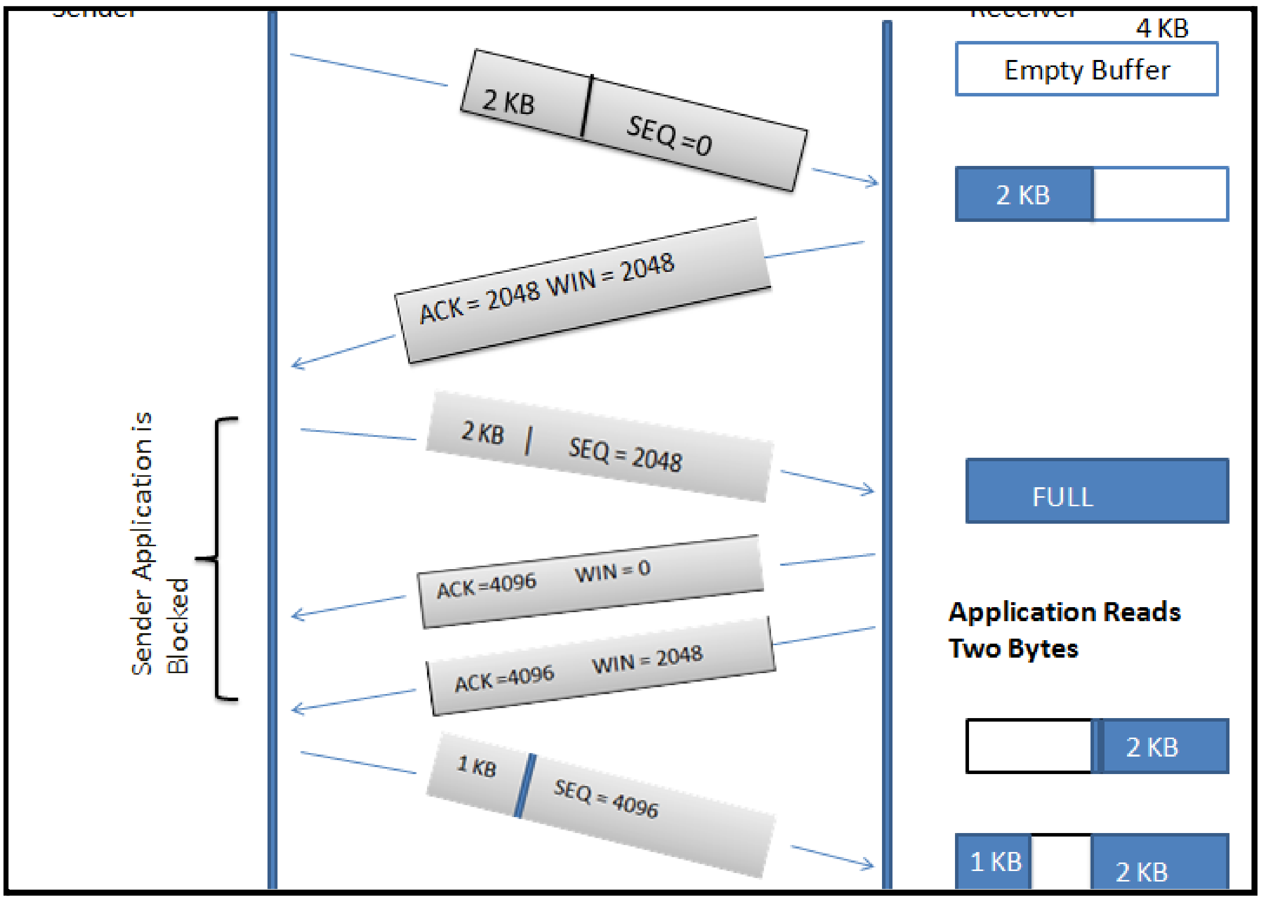
CS3205 A3 REPORT

TCP congestion control



Rudra Desai (CS18B012)

26-03-2021
3rd Year, BTech CSE

AIM

This project aims to simulate the simplified TCP congestion control algorithm and generate plots for the same.

INTRODUCTION

The TCP protocol is a transport layer protocol, mainly used when reliability and multiplexing are needed. It is a full-duplex protocol, providing communication in both directions. As the number of devices using this protocol increases with time, congestion occurs. The following report shows how the Go-Back-N sliding window algorithm and individual acknowledgments modify CWS (congestion window size) to adjust according to the congestion while maintaining 100% link usage.

EXPERIMENTAL SETUP

- We assume that the Receiver window size stays 1MB for the whole simulation.
- The sender always has data to send.
- Sender's MSS is 1 KB. Each segment has a fixed length of one MSS.
- The Go-Back-N algorithm with individual acknowledgment is used.
- The congestion window is always interpreted as multiples of MSS

ENTITIES INVOLVED AND FUNCTIONS USED

1. **Sender Class**
    - Sends data segments and receives acknowledgments
    - Controls the CWS according to the acknowledgments received
    - Uses individual timers for each data segment sent.
    - Go-Back-N with individual acknowledgments is used.
    - The implementation is similar to TCP Tahoe, but three duplicate acknowledgments are not considered because of this experiment's simplicity.
    - **External Methods** (Used by Switch class object during simulation)
        i. **print()** → Prints all the object variables for debugging purposes
        ii. **send_next_segment()** → Creates and returns the next data segment

to be sent. The Switch class object calls this function during simulation.

iii. **recv_ack(segment_no)** → Receives the acknowledgment for the data segment with id segment_no and modifies cws accordingly.

iv. **check_timeout()** → Checks for timeout. If a timeout occurs, change threshold, cws, next_windo_to_send, curr_window_start, etc., to handle timeouts.

v. **is_ready_to_send()** → Checks sender's readiness for sending new segments.

vi. **is_complete()** → Checks if the sender has received the acknowledgment for the last packet.

- **Internal Methods**

  i. **clear_segment_sent_times()** → Clears all the running timeout timers. Used in case of timeouts.

  ii. **change_cws()** → Checks for various conditions and changes cws accordingly.

2. **Receiver Class**
   - Receives data segments and send ACKs
   - For the sake of this experiment, out-of-order segments are discarded.
   - It sends individual ACKs instead of cumulative ACKs.
   - **External Methods** (Used by Switch class object during simulation)

     i. **print()** → Prints all the object variables for debugging purposes

     ii. **recv_segment(p)** → Receives data segment specified with p. Discards out-of-order segments. In case of successful retrieval, update the next ACK to be sent.

     iii. **send_ack()** → Send ACK based on the last received segment.

     iv. **ready_to_send_ack()** →Checks receiver object's readiness to send ACK

     v. **is_complete()** → Checks if the receiver has sent the ACK for the last segment.

3. **Switch Class**
   - Implementation of a simple packet switch.
   - Helps in sending/receiving segments between sender and receiver objects.
   - It pulls segments from the sender object and pushes them to the queue. Similarly, it removes ACKs from the queue and moves them to the sender.
   - The main difference in implementation from the original switch controller switch receives segments using sockets. But, in our implementation, we

especially pull/push it from the objects.

- **External Methods**
    i. **print()** → Print debug information according to the log level.
    ii. **simulate()** → Simulates sending and receiving of segments and ACKs. It uses the external functions from receiver and sender classes. Also, it prints debug information according to the log level.

4. **Argparse Class**
    - Helper class for command-line argument parsing
    - It contains several functions used for parsing arguments from the command line (i.e., using argc and argv)

FORMULAS USED

1. **Initial** CWS update : $CWS_{new} = K_i * MSS$
2. CWS update in **exponential growth** : $CWS_{new} = min(CWS_{old} + K_m * MSS , RWS)$
3. CWS update in **linear growth phase** : $CWS_{new} = min(CWS_{old} + K_n * MSS * MSS / CWS_{old} , RWS)$
4. CWS update when a **timeout occurs**: $CWS_{new} = max(1, K_f * CWS_{old})$

RESULTS AND OBSERVATION

1. Higher the fs' value, lesser the chances of a timeout, and lesser variation in plot curves. CWS increases exponentially until a timeout occurs.
2. The lesser the value of K_f, the higher the dip in CWS after a timeout, i.e., it recovers slower.
3. Higher the Kn's value, slower the growth of CWS in linear phase, i.e., it takes more time to hit timeout again.
4. Lesser the value of K_i, the lesser the value of the initial CWS.
5. The higher the value of K_m, the faster it recovers in exponential growth.

LEARNINGS

We learned how TCP recovers from congestion by modifying it not to degrade the switch utilization with this experiment. Also, we tweaked various params like K_i, K_m, K_n, K_f, P_s and saw their effects on CWS. Moreover, we saw how the Go-Back-N algorithm works along with individual ACKs.

CONCLUSION

Even though the whole experiment was highly simplified for better understanding, we learned a lot. This experiment helped us understand one of the most critical TCP layer factors, TCP congestion control.

ADDITIONAL REMARKS

The simulation could have been made better by actually using multiple devices and then plotting the rates.

REFERENCES

1. Stanford CS144 course
2. https://www.geeksforgeeks.org/sliding-window-protocol-set-2-receiver-side/
3. https://www.geeksforgeeks.org/sliding-window-protocol-set-3-selective-repeat/