Report - CS3205 Assignment 2

Aniswar Srivatsa Krishnan
Computer Science and Engineering
Indian Institute of Technology Madras
CS18B050

***Abstract*—The objective of this assignment is to emulate the TCP congestion control algorithm**

***Index Terms*—TCP, IP, networks, congestion, algorithm**

I. Introduction

The objective of this assignment is to emulate the TCP congestion control algorithm. Congestion control is one of the fundamentally important problems in networking. Specific TCP mechanisms are used to provide for a reliable data transfer service in the face of packet loss. In practice, such loss typically results from the overflowing of router buffers as the network becomes congested. Packet retransmission thus treats a symptom of network congestion (the loss of a specific transport-layer segment) but does not treat the cause of network congestion - too many sources attempting to send data at too high a rate. To treat the cause of network congestion, mechanisms are needed to throttle senders in the face of network congestion.

TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion. The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate

II. Experimental Details

*A. TCP Congestion Control Algorithm : **Additive Increase Multiplicative Decrease***

The Additive Increase Multiplicative Decrease (AIMD) Algorithm has three phases.

- **Slow Start Phase:** starts slowly - increment of congestion window is exponential till threshold is reached. Each time an ACK is received by the sender, the congestion window is increased by a constant (K_m) number of segments. So, the congestion window (CWND) becomes K_m times on every RTT (Round Trip Time).
- **Congestion Avoidance Phase**: After reaching the threshold, the congestion window is incremented by 1 for every RTT.
- **Congestion Detection Phase:** Multiplicative decrement. If congestion occurs, sender goes back to Slow Start phase or Congestion Avoidance phase. Congestion is detected through retransmission.
  - Case 1: Retransmission due to Timeout - congestion possibility is high. (Fast retransmission)
    1) ssthresh = cwnd / 2
    2) set cwnd = max(1, K_f* cwnd)
    3) start wth slow start phase again
  - Case 2: Retransmission due to 3 duplicate ACKs - congestion possibility is less. (Fast recovery)
    1) cwnd = ssthresh = cwnd/2
    2) start with congestion avoidance phase

  In this assignment, we consider only Case 1, i.e., retransmission due to timeout.

*B. **Assumptions and Variables***

- Receiver Window Size(RWS) is set to 1 MB, and does not change during the entire duration of the emulation.
- The Sender always has data to send to the receiver.
- Sender's Maximum Segment Size (MSS) is 1 KB. Each segment has a fixed length of one MSS.
- Go-back-N is used, but cumulative acknowledgments are not considered. For each segment, an individual timeout timer and ACK are used.
- The congestion window is always intrepreted as a multiple of MSS (1 KB).
- The congestion threshold is always set to 50% of the current CW value (when timeout occurs).
- $K_i, 1 \leq K_i \leq 4$ denotes the initial congestion window (CW). Default value is 1. The initial CW is given by:

$$CW_{new} = K_i \cdot MSS \tag{1}$$

- $K_m, 0.5 \leq K_m \leq 2$ denotes the multiplier of Congestion Window, during exponential growth phase. Default value is 1. When a segment's ACK is successfully received,

$$CW_{new} = min(CW_{old} + K_m \cdot MSS, RWS) \tag{2}$$

- $K_n, 0.5 \leq K_n \leq 2$ denotes the multiplier of Congestion Window, during linear growth phase. Default value is 1. When a segment's ACK is successfully received,

$$CW_{new} = min(CW_{old} + K_l \cdot MSS \cdot \frac{MSS}{CW_{old}}, RWS) \tag{3}$$

- $K_f, 0.1 \leq K_f \leq 0.5$ denotes the multiplier when a timeout occurs:

$$CW_{new} = max(1, K_f \cdot CW_{old}) \tag{4}$$

- $P_s, 0 < P_s < 1$, denotes the probability of not receiving the ACK packet for a given segment before its timeout occurs.

C. *Implementation Details*

*1) **Execution**:*

- The program is invoked with the following command-line parameters:

```
% ./cw -i <double> -m <double>
-n <double> -f <double>
-s <double> -T <int> -o outfile
```

- The values correspond to K_i , K_m , K_n , K_f , P_s and the total number of updates to be performed before the emulation stops. The output (specified below) is saved in an output file.
- The congestion window progression is done on a slot-by-slot basis. In each ”round”, a set of segments are sent, in proportion to the current value of CW, i.e. $N = \frac{CW}{MSS}$. For example, if CW is equal to 4.3 KB, five packets are sent. However, the CW growth is based on MSS values as explained earlier.
- For each segment transmitted, the ACK for this segment is received before timeout with random probability (1-P_s) and timeout occurs with probability P_s. Depending on this outcome, the CW increases and decreases as described earlier.
- The congestion window value is printed to the output file (one per line) at each CW update. A graph with x-axis being the update number and y-axis the corresponding CW is plotted.

The following are the functions and/or scripts used in the emulation. The algorithm is implemented in `C++` whereas the plotting of graphs is done with the help of `Python`

*2) **`argument_handler()`**:*

- This function takes care of handling the command line arguments as explained above and assigns it to the parameter variables which are declared globally
- The function also checks the constraints for the parameters mentioned and throws an error if the constraints are not satisfied.

*3) **`main()`**:*

- The function performs the simulation according to the algorithm described above. It does initialization of certain variables(especially a random number generator based on the Bernoulli distribution [1] with parameter P_s) first and then proceeds onto the main segment.
- The main segment of the program consists of a while loop which runs till the number of updates is less than T.
- In each iteration of the loop, the N is calculated based on the current value of the CW and a for loop is done for N iterations. The random number generator is used to check whether timeout has occured or not and depending on the outcome appropriate action is taken as per the algorithm.
- After every update to the CW, the value is written to the outfile.

*4) **`plot()`**:*

- This function calls the `Python` file `plot.py` (described below) using the `system()` fucntion, which then plots the graphs according to the output file generated. The path of the output file and the path of the file where the image is to be saved are given as command line arguments to the `Python` file.

*5) **`plot.py`**:*

- This `Python` script takes in the path of the output file and the path of the file where the image is to be saved as command line arguments and plots the graph using the `Matplotlib` library.

III. Results and Observation

The graphs are plotted for the following parameter combinations

$$K_i \in \{1, 3\} \tag{5}$$
$$K_m \in \{1, 1.5\} \tag{6}$$
$$K_n \in \{0.5, 1\} \tag{7}$$
$$K_f \in \{0.1, 0.3\} \tag{8}$$
$$P_s \in \{0.01, 0.0001\} \tag{9}$$

A. *Dependence on K_i*

K_i affects the starting value; for larger values of K_i, the value of CW starts from a proportionately larger value. E.g., Fig. 17. starts at a higher value than Fig. 1, Fig. 18. starts at a higher value than Fig. 2, etc.

B. *Dependence on K_m*

K_m affects the slope of the linear section of the graph (the slow start phase); for larger values of K_m , the value of CW grows more rapidly in the linear growth section of the graph. Graphs with larger K_m grow taller than ones with smaller K_m as a consequence. E.g., The slopes of the lines in Fig. 11. are higher value than the slopes of the lines in Fig. 3, The slopes of the lines in Fig. 12. are higher value than the slopes of the lines in Fig. 4, etc.

C. *Dependence on K_n*

Dependence on K_n affects the slope of the logarithmic section of the graph(the congestion avoidance phase); for larger values of K_n , the value of CW tries to converge towards a larger value, and reaches larger values. Graphs with larger K_n grow taller in this phase than ones with smaller K_n. E.g., compare Fig. 5 and Fig. 9, Fig. 1 and Fig. 3, etc.

D. *Dependence on K_f*

K_f affects value to which CW returns after a timeout; thus for larger K_f values, the linear sections are shorter and the logarithmic section is longer than for smaller K_f values. E.g., Fig. 3 has longer sections than Fig. 11, etc.

E. Dependence on P_s

For very small values of P_s like 0.0001 (which correspond to very rare timeouts), we observe that the graph stays in the slow start phase and saturates at 1024 which is the upper bound of CW due to the RWS being 1 MB and rarely comes down. This behaviour is expected , since there are almost no timeouts. But for relatively large P_s like 0.01 (which correspond to relatively more frequent timeouts), we observe that the timeouts are more frequent, resulting in shorter number of updates between timeouts, making the graphs smaller and leading to more number of segments or pieces in the graph. E.g., Fig. 2, Fig. 4, Fig. 6, Fig. 8 exhibit the behaviour of saturation, while Fig. 1, Fig. 3, Fig. 5, Fig. 7 exhibit more frequent timeouts.

IV. Graphs

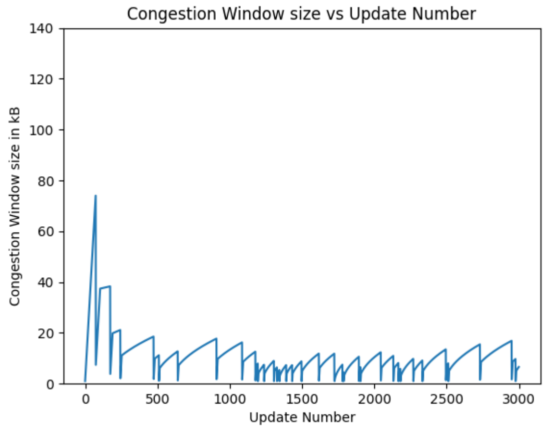

Fig. 1. $K_i = 1, K_m = 1.0, K_n = 0.5, K_f = 0.1, P_s = 0.01$

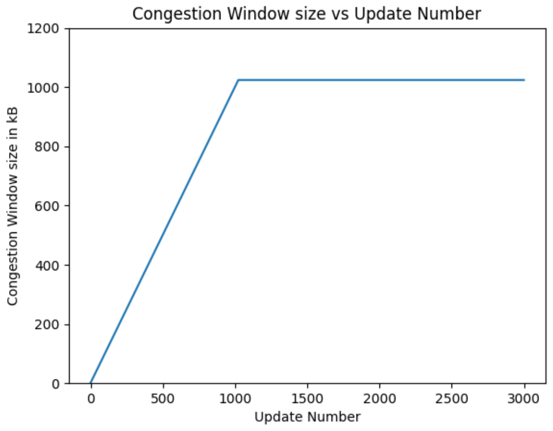

Fig. 2. $K_i = 1, K_m = 1.0, K_n = 0.5, K_f = 0.1, P_s = 0.0001$

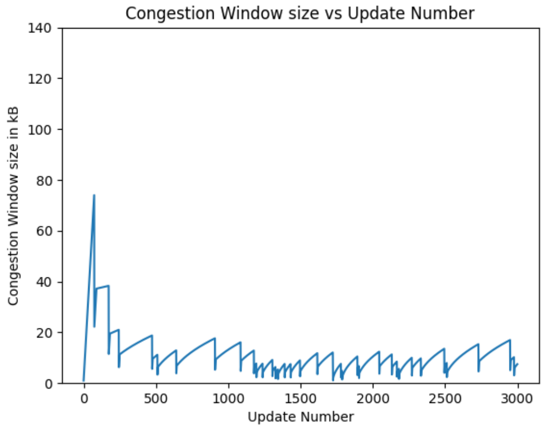

Fig. 3. $K_i = 1, K_m = 1.0, K_n = 0.5, K_f = 0.3, P_s = 0.01$

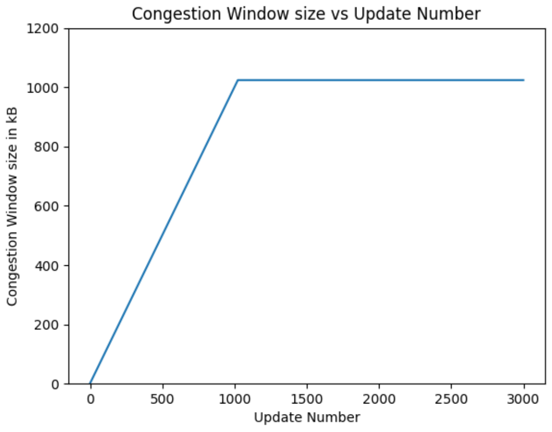

Fig. 4. $K_i = 1, K_m = 1.0, K_n = 0.5, K_f = 0.3, P_s = 0.0001$

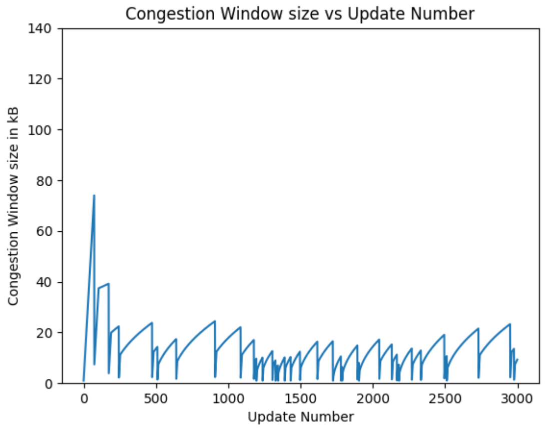

Fig. 5. $K_i = 1, K_m = 1.0, K_n = 1.0, K_f = 0.1, P_s = 0.01$

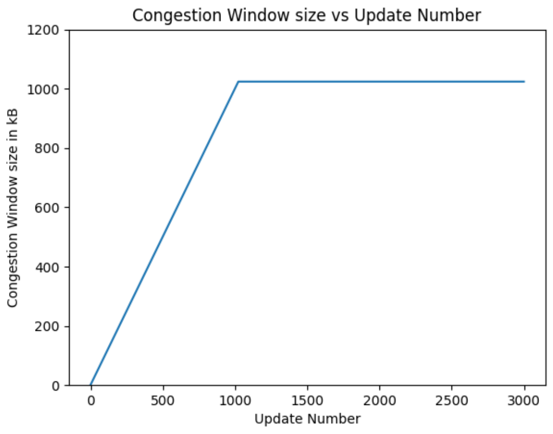

Fig. 6. $K_i = 1, K_m = 1.0, K_n = 1.0, K_f = 0.1, P_s = 0.0001$

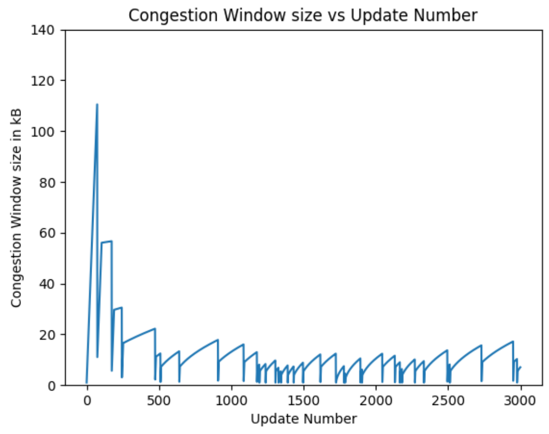

Fig. 9. $K_i = 1, K_m = 1.5, K_n = 0.5, K_f = 0.1, P_s = 0.01$

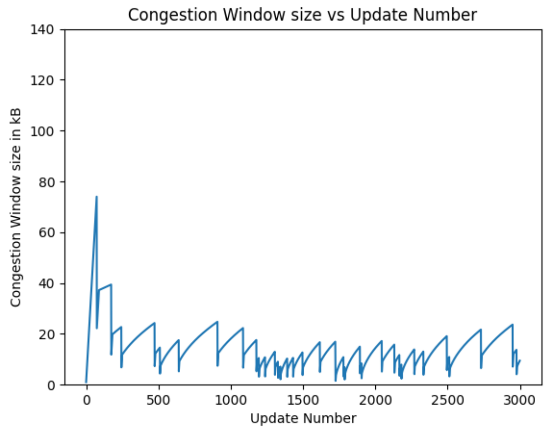

Fig. 7. $K_i = 1, K_m = 1.0, K_n = 1.0, K_f = 0.3, P_s = 0.01$

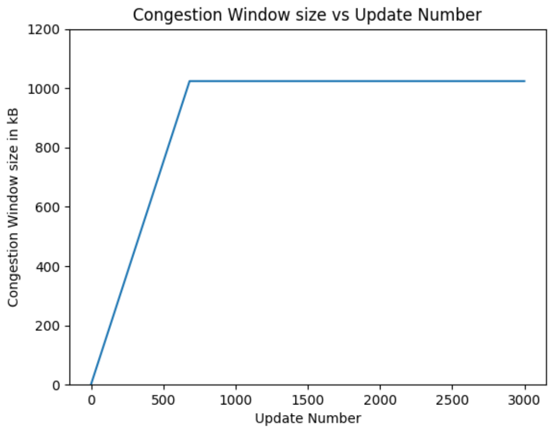

Fig. 10. $K_i = 1, K_m = 1.5, K_n = 0.5, K_f = 0.1, P_s = 0.0001$

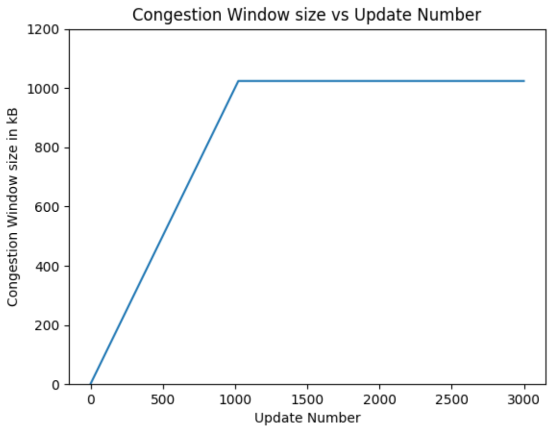

Fig. 8. $K_i = 1, K_m = 1.0, K_n = 1.0, K_f = 0.3, P_s = 0.0001$

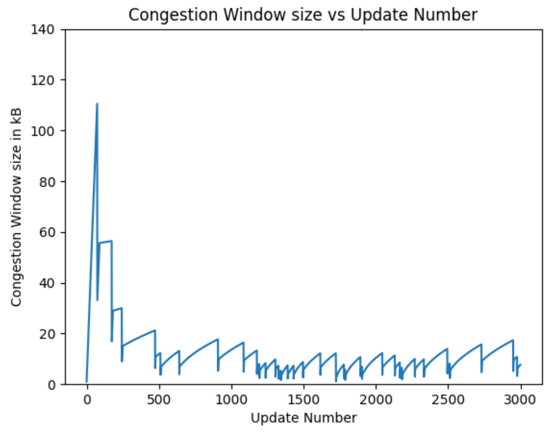

Fig. 11. $K_i = 1, K_m = 1.5, K_n = 0.5, K_f = 0.3, P_s = 0.01$

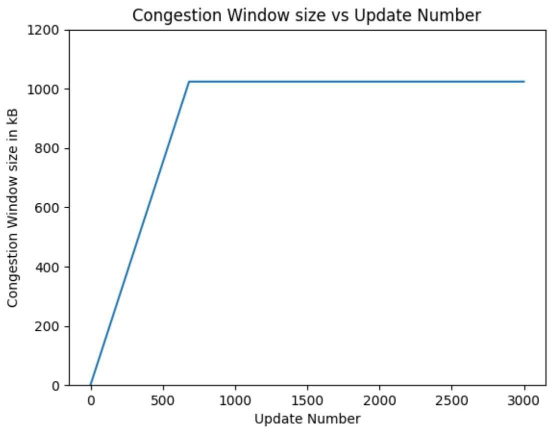

Fig. 12. $K_i = 1, K_m = 1.5, K_n = 0.5, K_f = 0.3, P_s = 0.0001$

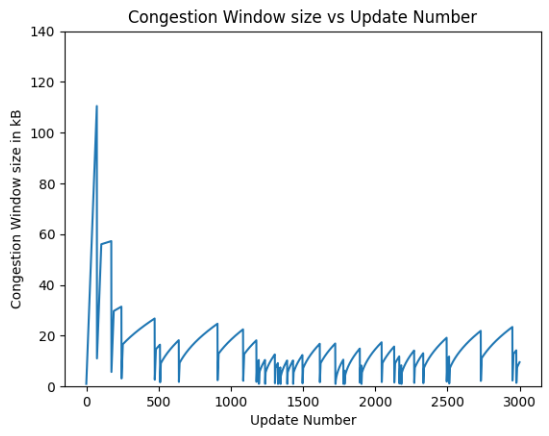

Fig. 13. $K_i = 1, K_m = 1.5, K_n = 1.0, K_f = 0.1, P_s = 0.01$

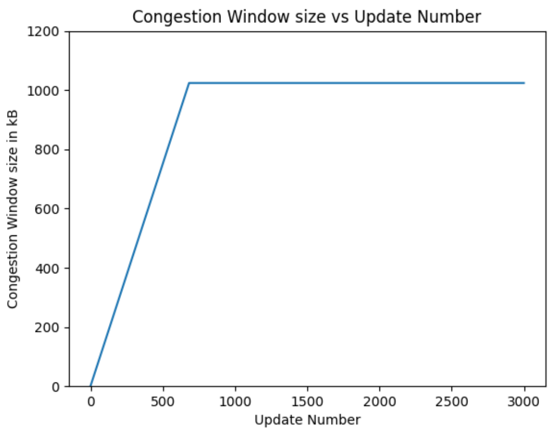

Fig. 14. $K_i = 1, K_m = 1.5, K_n = 1.0, K_f = 0.1, P_s = 0.0001$

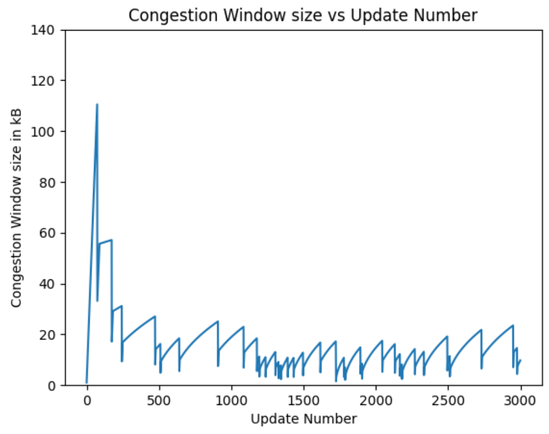

Fig. 15. $K_i = 1, K_m = 1.5, K_n = 1.0, K_f = 0.3, P_s = 0.01$

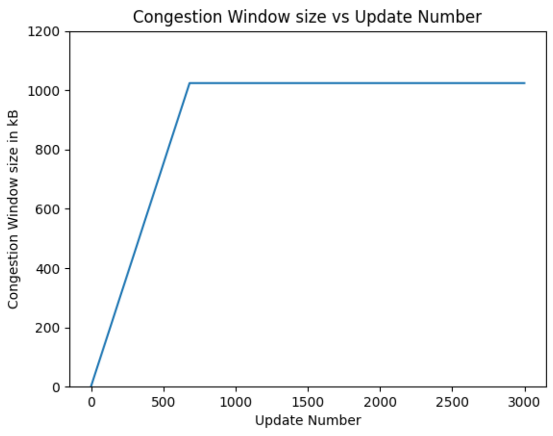

Fig. 16. $K_i = 1, K_m = 1.5, K_n = 1.0, K_f = 0.3, P_s = 0.0001$

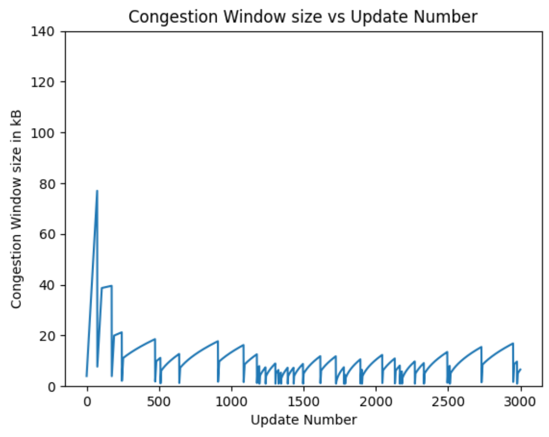

Fig. 17. $K_i = 4, K_m = 1.0, K_n = 0.5, K_f = 0.1, P_s = 0.01$

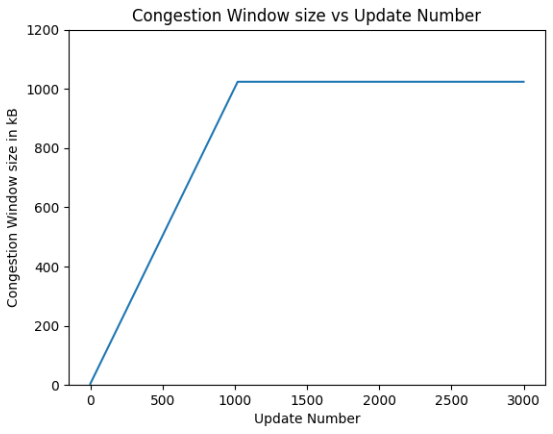

Fig. 18. $K_i = 4, K_m = 1.0, K_n = 0.5, K_f = 0.1, P_s = 0.0001$

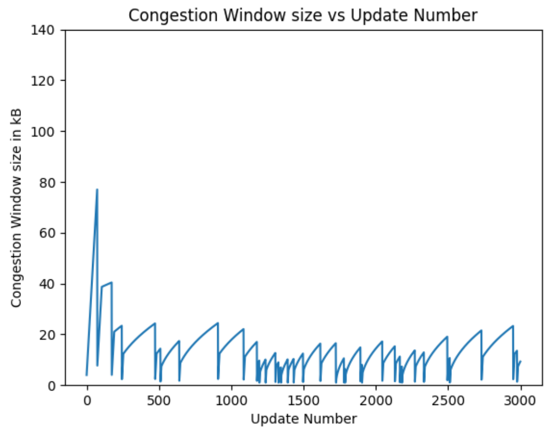

Fig. 21. $K_i = 4, K_m = 1.0, K_n = 1.0, K_f = 0.1, P_s = 0.01$

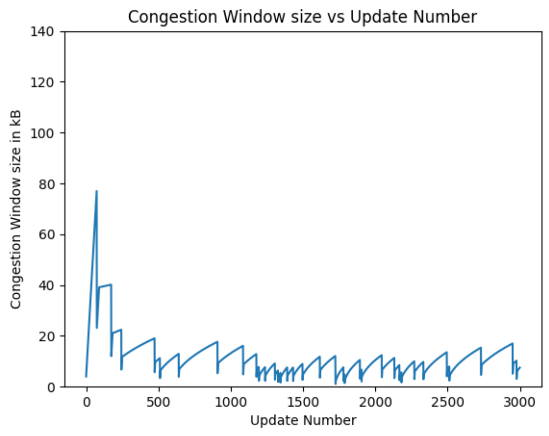

Fig. 19. $K_i = 4, K_m = 1.0, K_n = 0.5, K_f = 0.3, P_s = 0.01$

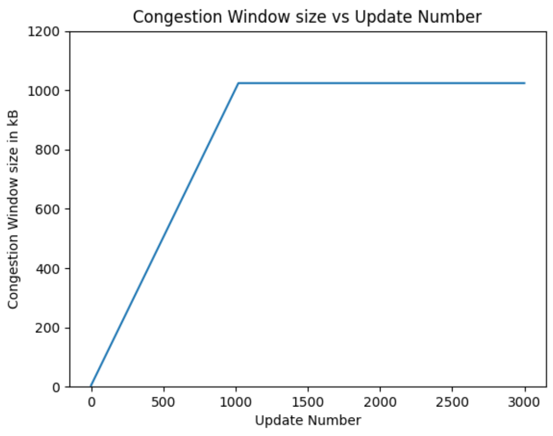

Fig. 22. $K_i = 4, K_m = 1.0, K_n = 1.0, K_f = 0.1, P_s = 0.0001$

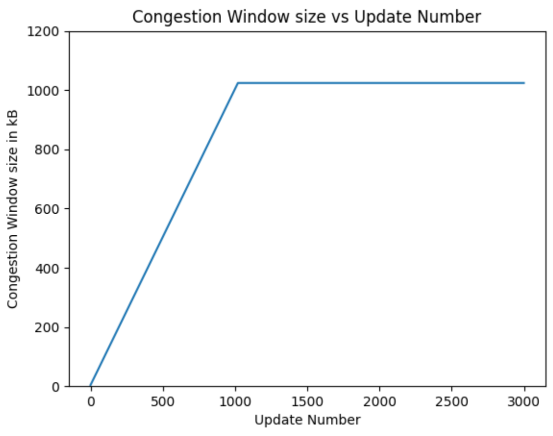

Fig. 20. $K_i = 4, K_m = 1.0, K_n = 0.5, K_f = 0.3, P_s = 0.0001$

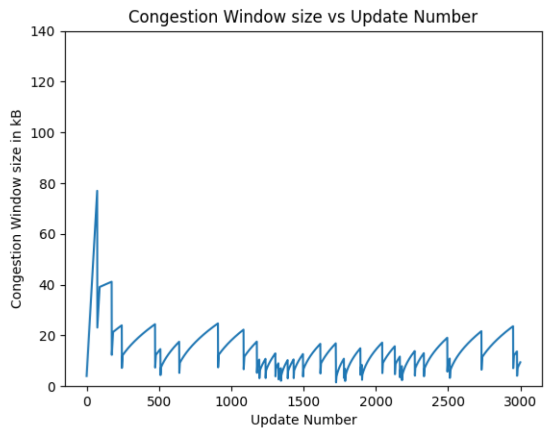

Fig. 23. $K_i = 4, K_m = 1.0, K_n = 1.0, K_f = 0.3, P_s = 0.01$

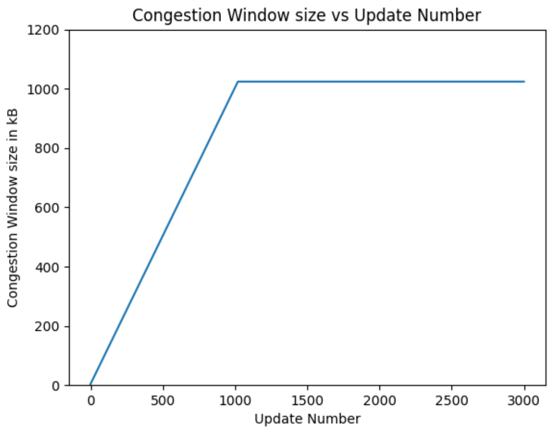

Fig. 24. $K_i = 4, K_m = 1.0, K_n = 1.0, K_f = 0.3, P_s = 0.0001$

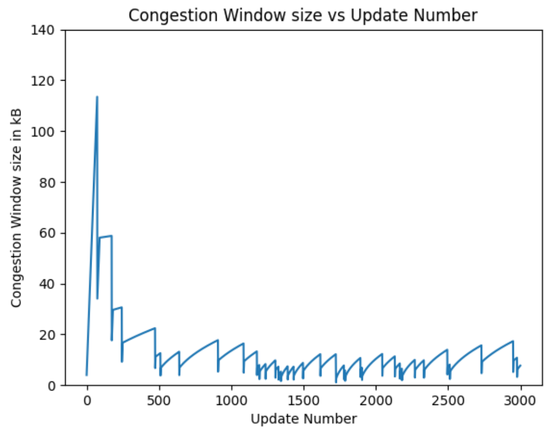

Fig. 27. $K_i = 4, K_m = 1.5, K_n = 0.5, K_f = 0.3, P_s = 0.01$

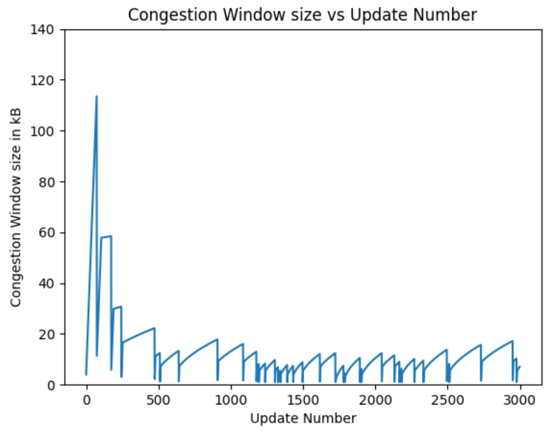

Fig. 25. $K_i = 4, K_m = 1.5, K_n = 0.5, K_f = 0.1, P_s = 0.01$

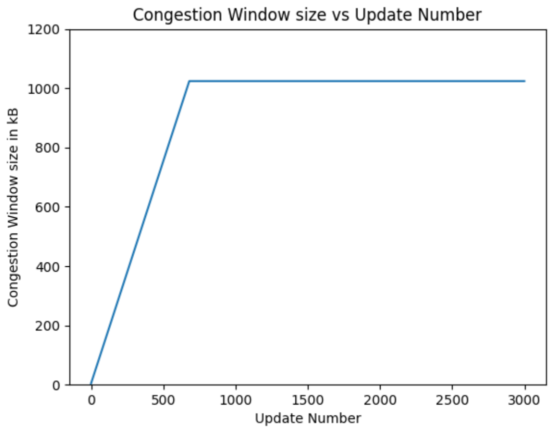

Fig. 28. $K_i = 4, K_m = 1.5, K_n = 0.5, K_f = 0.3, P_s = 0.0001$

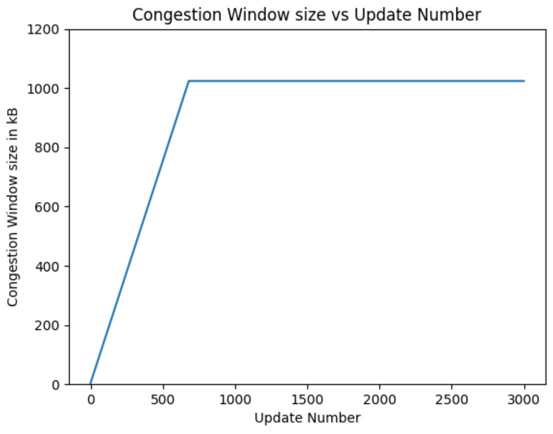

Fig. 26. $K_i = 4, K_m = 1.5, K_n = 0.5, K_f = 0.1, P_s = 0.0001$

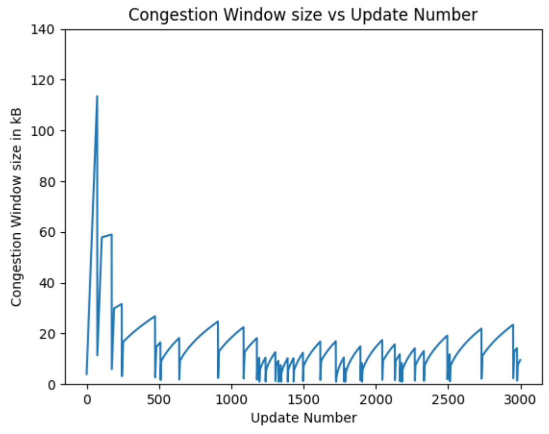

Fig. 29. $K_i = 4, K_m = 1.5, K_n = 1.0, K_f = 0.1, P_s = 0.01$

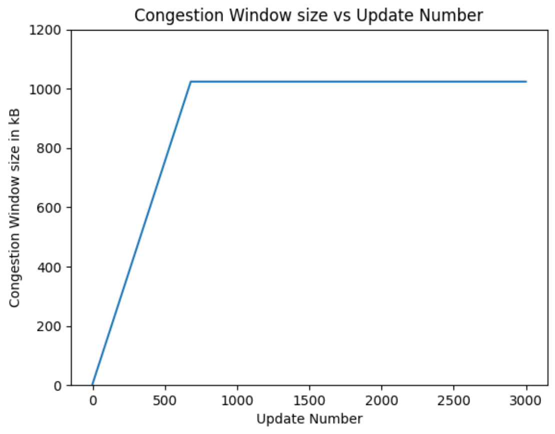

Fig. 30. $K_i = 4, K_m = 1.5, K_n = 1.0, K_f = 0.1, P_s = 0.0001$

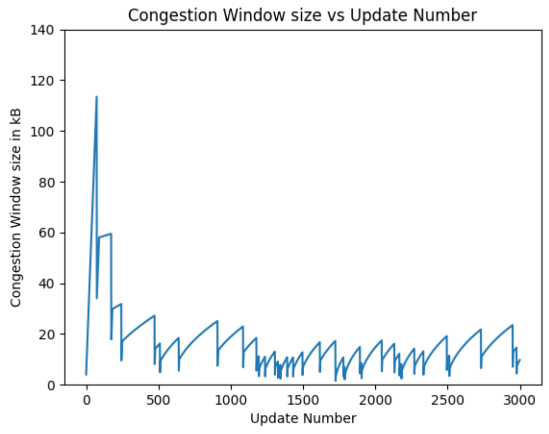

Fig. 31. $K_i = 4, K_m = 1.5, K_n = 1.0, K_f = 0.3, P_s = 0.01$

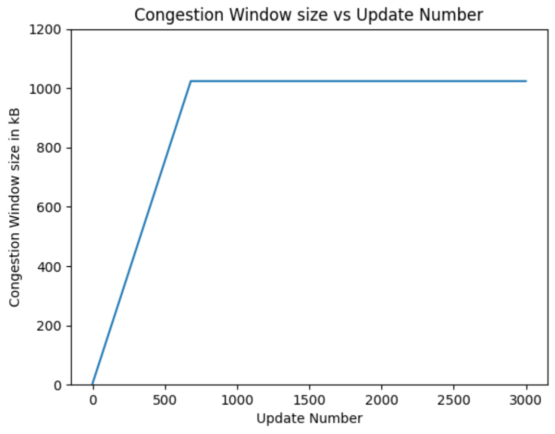

Fig. 32. $K_i = 4, K_m = 1.5, K_n = 1.0, K_f = 0.3, P_s = 0.0001$

V. LEARNINGS

This assignment helped us understand the celebrated TCP congestion control algorithm in great detail. We understood the effect of various parameters on the different phases of the algorithm with the help of many graphs. The assignment also familiarized me with the different random number generators available in C++. We also learned that even if we do not have control over the actual implementation of the TCP protocol, we could use simulation to understand the various properties of the protocol.

VI. ADDITIONAL THOUGHTS

- The algorithm can be extended to understanding the progress of the value of CW with respect to the RTT, along with the current analysis with respect to CW. This will help us visualize the exponential growth in the slow start phase.
- The implementation can include the Fast Recovery phase of the algorithm. This involves retransmission due to 3 duplicate ACK method in the congestion detection phase, and subsequent movement into the linear growth (congestion avoidance phase).
- The Vegas version of the TCP congestion control algorithm, which is the recent update to the protocol could also be included. This involves detecting congestion before timeout occurs, with the help of RTT analysis.

VII. CONCLUSION

Thus the simulation of the TCP Congestion Control algorithm has been completed successfully. It's performance under various parameters was studied. We observe that the results of the experiment correlate well with the mathematical formulation of the algorithm.

REFERENCES

[1] Bernoulli Distribution - C++ standard library reference documentation
[2] Matplotlib - Official Documentation