DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING, IIT MADRAS

CS3205: Assignment 2

Harshit Kedia
CS17B103

1 -GO BACK N PROTOCOL

Pkt Gen rate	Total Pkts	Pkt Length	Drop Prob	ReTransmission ratio	Round Trip Time
20	100	256	0	0.10	0.270
20	100	1500	0	0.08	0.490
20	100	256	0.001	0.16	0.326
20	100	1500	0.001	0.17	0.561
20	100	256	0.01	0.17	0.510
20	100	1500	0.01	0.19	0.532
20	100	256	0.1	0.55	1.114
20	100	1500	0.1	0.54	0.836
300	1500	256	0	0.00	0.178
300	1500	1500	0	0.00	0.276
300	1500	256	0.001	0.009	0.220
300	1500	1500	0.001	0.014	0.338
300	1500	256	0.01	0.043	0.232
300	1500	1500	0.01	0.045	0.331
300	1500	256	0.1	0.570	0.377
300	1500	1500	0.1	0.588	0.451

- Go back N protocol has been successfully implemented as a reliable transmission protocol, using a server and a client process, running on the same computer, sharing data using UDP sockets in Java language.
- The sender/client program runs 3 concurrent threads. One for periodically generating and placing data packets in the buffer (if space is available). Second, for receiving the acknowledgement packets from the server, and third one to send available buffer packets following the Go back-N policy.
- Having threads inevitably generates race conditions on the shared memory, like buffer and a few flag variables, hence locks were used to make program thread safe.
- The server/receiver program runs on only 1 thread, receiving the packet and reading data, hence having O(length) workload per packet, and sending acknowledgement packets meanwhile following the protocol policies.
- To limit the number of maximum re-sends to 5, as said in the problem statement, the server maintains a unique count per packet and doesn't allow it to drop any further.
- Re-transmission rate is not zero even for drop probability = 0, because of dynamic timeout period and the property of go-back-N, having to resend other packets at the window even in-case of 1 missing acknowledgement.
- Having a drop probability of 0.1 makes us resend 50% the data. This shows that this protocol is not very good for practical purposes.

2 -SELECTIVE REPEAT PROTOCOL

Pkt Gen rate	Total Packets	Drop Probability	Re-Transmission ratio	Round Trip Time
20	100	0	0.00	0.357
20	100	0.001	0.00	0.343
20	100	0.01	0.02	0.380
20	100	0.1	0.160	0.358
20	100	0.25	0.310	0.330
20	100	0.5	1.070	0.318
300	1500	0	0	0.262
300	1500	0.001	0.051	0.281
300	1500	0.01	0.392	0.360
300	1500	0.1	0.669	0.315
300	1500	0.25	1.049	0.357
300	1500	0.50	2.053	0.271

- Selective Repeat protocol has also been implemented as a reliable transmission protocol, using a server and a client process, running on the same computer, sharing data using UDP sockets in Java language.
- Similar to part 1, the sender has 3 threads, one each for generating load, sending packets, and receiving acknowledgements. Also, the server here uses 1 thread but has more complex work to do than Go back N.
- The packet length is taken as a random number between 40 and 1500, hence not mentioned exclusively in table.
- Since we need individual timers for each sent packet, the overhead for the same is expensive in Java. Hence having a big WINDOW_SIZE leads to run-time errors and exceptions.
- Here also the maximum re-sends of a packet is limited to 10, as mentioned in the problem statement. It can be checked by putting a drop probability of 1.
- Having a small buffer-size leads to buffer being built up in receiver side to store following packets, and since we do send the acknowledgements instantly, the buffer on the sender's side gets cleared up, making it to send further more packets, which eventually get dropped at receiver side because of buffer limitation.
- Setting the timer value <10 ms, leads to exception and memory overflow issue, hence this longer timeout-window also adds up with the previous mentioned problem.
- Compared to Go Back N, here the amount of data re-transmitted is lesser for given drop probability for smaller sending rates. Surprisingly, for high packet sending rate, the ratio is very high.

3 -GENERAL INFERENCES

- The maximum possible size of sequence number is taken large, 4 Bytes to avoid confusion and ambiguity.
- Since we use the nanoTime() system call a lot of times, even from different threads, the function being thread unsafe, requires a lock to gain mutual exclusion, else it returns a value of 0, making results erroneous.
- In the sender process for both protocols, we have multiple threads altering the contents of the buffer, hence we also need a lock for the buffer to make the program thread safe.
- Having these locks guarantees mutual exclusion but on the cost of program parallelism, we can clearly see the selective repeat suffering due to so many parts of the code being the critical section.
- The 1st packet always take significantly more round trip time than subsequent ones, this maybe because the OS takes time to establish a strong connection between processes.
- If we always spin in the packed sending thread, to check for available packets in buffer, the other thread doesn't get enough CPU time, hence we must yield out thread using sleep, though for a few nanoseconds (50 here).

- In the receiver side, to simulate reading of entire packet, a for loop with small computation is done for the packet.length cycles.
- Round Trip Time is calculated as the time taken by packet to go from sender to receiver and of acknowledgement to arrive back in the successful iteration for the packet. If the packet is dropped, that overhead is not count, otherwise the average RTT will depend on the timeout period.
- Having the DROP PROBABILITY as a small number, leads to almost no drops, even after multiple runs and higher total number of packets, hence I have not used real-time simulating values like 10e-5.

4 -SUMMARY AND LEARNING

- The average Re-Transmission ratio is lower for Selective Repeat protocol in case of very small Drop probability and smaller packet generation rate.
- Port number chosen must be a larger number(>1024), these are freely available for use to processes.
- Since the output is dependent on a probabilistic parameter, the experiments are run multiple times and average of them is reported in the table(s) given.
- It is observed that the total time taken to transmit all packets at very small drop probability is very close to (Total Packets / Rate). This indicates functioning and well-optimized code is written having low junk overheads.
- For all the variables, I have set some default values, so that in-case if some flag is missed during running the code, it doesn't get stuck or throw some random error. IP is set to local host, unless mentioned otherwise in command-line parameter.
- This is our first time writing code at this scale, using Sockets, Threads and Locks, and then debugging corner cases and deadlocks, all in Java, a relatively undiscovered language from my side, dealing with so many objects. This assignment was very fruitful and presented a steep learning curve and a great overall experience.