CSE 489: PA #2

Due on April 1, 2016 at 2:00pm

Dimitrios Koutsonikolas T/TH 2:00-3:20PM

Robert Shannon

Statement of Academic Integrity

I have read and understood the course academic integrity policy located under this link: `http://www.cse.buffalo.edu/faculty/dimitrio/courses/cse4589_s16/index.html#integrity`

Multiple Software Timers for Selective Repeat

The Selective Repeat protocol requires an individual timer for each unacknowledged packet, but the simulator only provides a single hardware timer. To work around this constraint, multiple software timers were implemented using the single hardware timer provided by the simulator. This was achieved by the following steps:

1. Use a struct `<pkt_timer>` to represent each software timer
2. Set the hardware timer to fire every 1.0 second
3. Fire all expired software timers each time a hardware timer interrupt occurs
4. For each software timer interrupt, call the software timer interrupt handler routine
5. After handling the software timer interrupt, restart the timer
6. Repeat steps 3-5 until program exits

```
/**
 * A packet timer.
 */
struct pkt_timer {
  int seq_num;     // Sequence number of the packet associated with this timer
  float next_fire; // Next scheduled timer interrupt
  bool active;     // Whether this timer is active or not
};
/**
 * Container for all packet timers.
 */
std::vector<pkt_timer> pkt_timers;
/**
 * Helper methods to manage multiple packet timers
 * and their corresponding interrupt handlers.
 */
/* Create new packet timer */
void new_pkt_timer(int seq_num);
/* Destroy existing packet timer */
void destroy_pkt_timer(int seq_num);
/* Check if packet timer exists */
bool pkt_timer_exists(int seq_num);
/* Trigger a packet timer interrupt */
void fire_pkt_timer(int seq_num);
/* Default packet timer interrupt handler routine */
void pkt_timer_interrupt_handler(int seq_num);
/* Fire interrupt for all expired packet timers */
void fire_expired_pkt_timers();
```

Timeout Schemes Used

The timeout interval (how long a protocol waits before resending unacknowledged packet(s)) was chosen to maximize throughput performance with an application generating messages every 50 time units under the widest range of conditions possible. The "optimal" values below were determined through experimentation, with foreknowledge of the RTT time to be on average 10.0 time units.

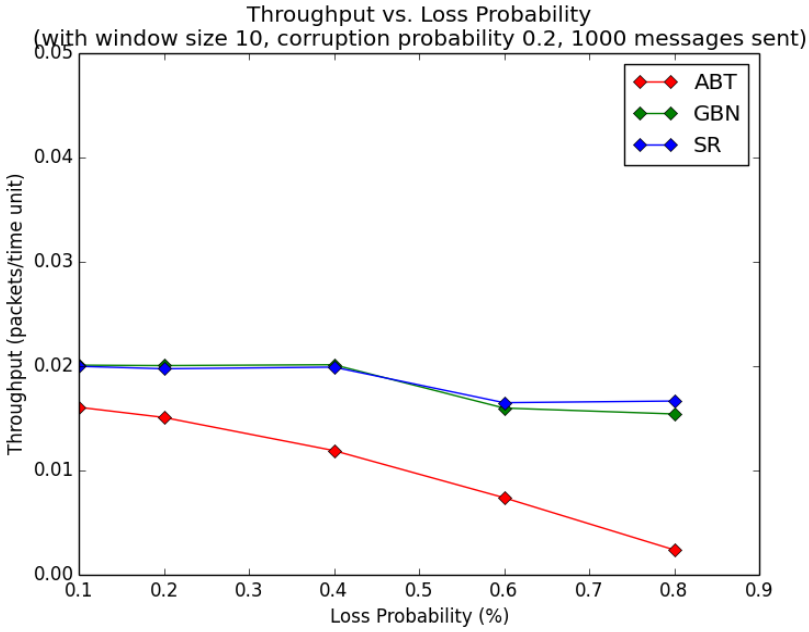
1. **Alternating Bit (ABT)** — $t = 10.0$, re-send unacknowledged packet after 10.0 time units.
2. **Go-Back-N (GBN)** — $t = 11.0$, re-send all unacknowledged packets after 11.0 time units.
3. **Selective-Repeat (SR)** — $t = 15.0$, re-send unacknowledged packet after 15.0 time units. Note that SR uses an individual timer for each unacknowledged packet.

All of the timeout values are very close (or equal to in the case of ABT) to the link's average RTT. This is because the average RTT is the expected amount of time to receive an ACK after sending a packet. If an ACK is not received within 1 RTT, it is assumed that the corresponding packet or the ACK itself were either lost or corrupted, and as a result needs to be re-sent.

Experiments

In each of the experiments below, the protocols were tested by sending 1,000 messages from sender to receiver, with a mean time of 50 time units between message arrivals, a corruption probability of 0.2. The window sizes and loss probabilities were varied as specified, and the resulting throughput plotted.

1.a. Window size: 10; X-axis: Loss probability; Y-axis: Throughput (ABT, GBN and SR)

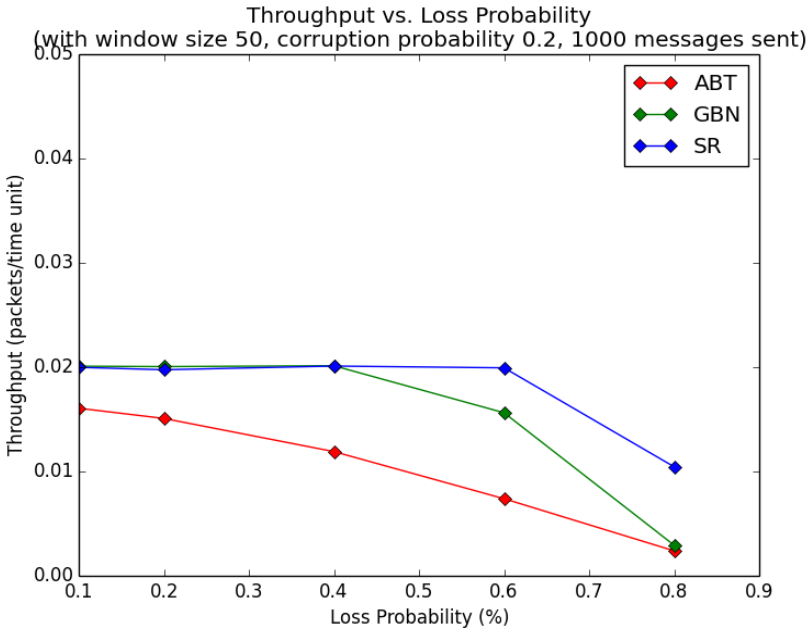


Expected: Throughput to decrease as loss probability increases. Similar throughput performance of both SR and GBN (with SR utilizing the link more efficiently). Inferior throughput performance of ABT when compared to SR and GBN across a wide range of loss probabilities.

Observed: Increasing loss probability lowers throughput. The maximum throughput of both SR and GBN are about equal with this window size. SR's throughput drops lower than that of GBN's at a loss probability at roughly $l \geq 0.6$. Both SR and GBN maintain their maximum throughput through $0.1 \leq l \leq 0.4$, while ABT's throughput drops consistently as loss probability is increased.

Conclusion: Increased packet loss eventually decreases throughput. SR and GBN appear to be more resilient to packet loss than ABT.

1.b. Window size: 50; X-axis: Loss probability; Y-axis: Throughput (ABT, GBN and SR)

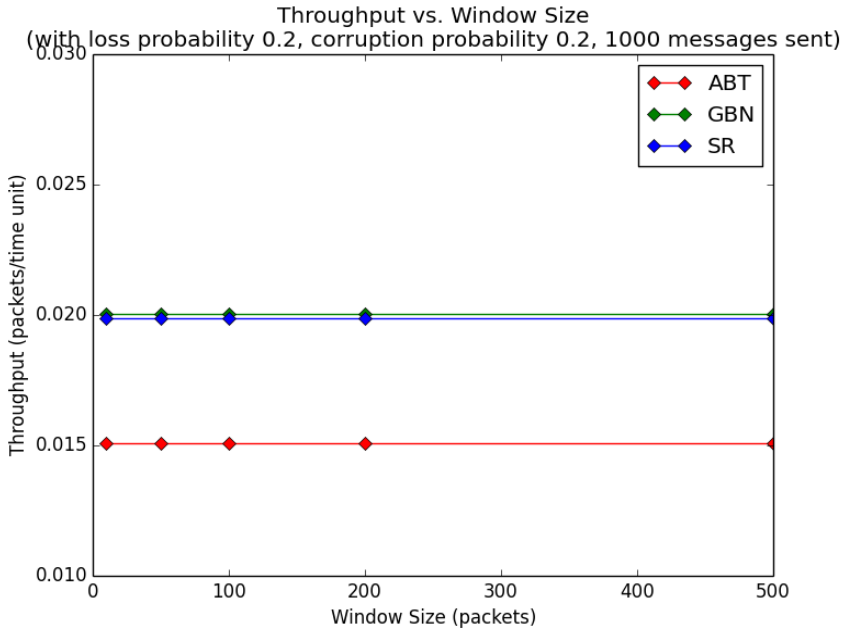


Expected: Same results as experiment (1a), but more throughput for SR and GBN with a larger window size.

Observed: Throughput drops with $t \geq 0.6$ for SR, and $t \geq 0.4$ for GBN. SR can maintain maximum throughput with up to 60% packet loss, while GBN can only do so with up to 40% packet loss. SR has a throughput that is consistently greater than or equal to that of GBN. ABT is again outperformed by both GBN and SR under all packet loss scenarios.

Conclusion: The larger window size did not result in more throughput, likely because the sender was not sending messages fast enough to fill the pipeline. Interestingly, increasing the window size made SR more resilient to packet loss while maintaining maximum throughput. SR at least matches or outperforms GBN in terms of throughput at most levels of packet loss.

2.a. Loss probability: 0.2; X-axis: Window size; Y-axis: Throughput (ABT, GBN and SR)

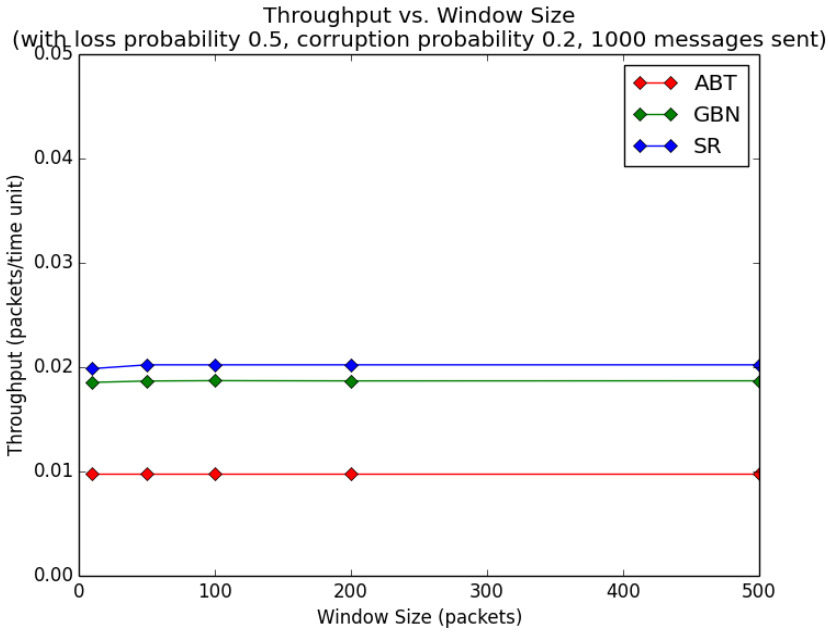


Expected: Constant throughput amongst all three protocols, with GBN and SR offering the highest throughput and ABT the lowest. ABT to most certainly be constant because it does not make use of windows.

Observed: All three protocols have a constant throughput across all window sizes.

Conclusion: A larger window will increase throughput only if an application is sending messages fast enough to fill it. Since SR and GBN both use windows, their maximum throughput is the same when packet loss is minimal.

2.b. Loss probability: 0.5; X-axis: Window size; Y-axis: Throughput (ABT, GBN and SR)

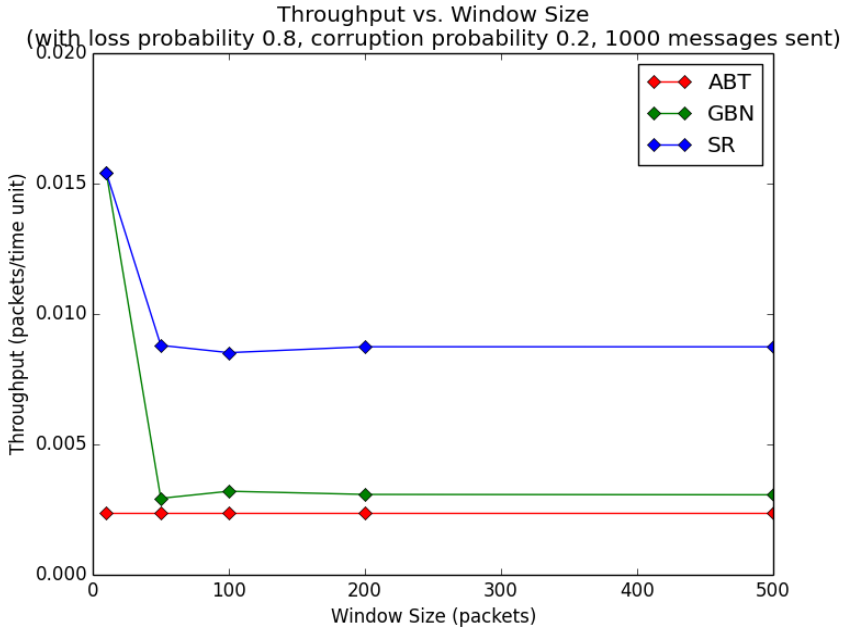


Expected: Constant throughput for each protocol, but lower than than that of experiment (2a) due to increased packet loss.

Observed: All three protocols again have a constant throughput, but this time SR clearly outperforms GBN. ABT is again outperformed by both SR and GBN. Both the throughput of GBN and ABT dropped, while SR remained almost unchanged.

Conclusion: SR's throughput is more resilient to packet loss than both GBN and ABT.

2.c. Loss probability: 0.8; X-axis: Window size; Y-axis: Throughput (ABT, GBN and SR) in one graph/plot.



Expected: Constant throughput for each protocol, but lower than than that of experiment (2b) due to increased packet loss.

Observed: Constant throughput for ABT. Large drop of throughput for both GBN and SR with window size ≥ 10, which stablizes with window size ≥ 50. SR decisvely outperforms GBN in terms of throughput with window size ≥ 50. Window size 10 offers the most throughput under this high packet loss scenario.

Conclusion: The throughput of SR is more resilient than GBN and ABT under conditions of high packet loss. There also exists an optimal window size which maximizes throughput under high packet loss conditions, in this case it is 10.

Conclusions Summarized

1. Increasing packet loss eventually decreases throughput in all three protocols.
2. The throughput of both SR and GBN are more resilient to packet loss than ABT.
3. The throughput of SR is more resilient to packet loss than both GBN and ABT.
4. Under ideal conditions ($l = 0.0, c = 0.0$) a larger window size will increase throughput only if an application sends messages fast enough to fill the window.
5. There exists a window size which maximizes throughput under high packet loss conditions ($l \geq 0.5, c \geq 0.2$).
6. From (1-5) SR is the protocol which offers the most throughput under the most diverse range of conditions.