Lab 2 Report– Phase 1

Yuan Chang

chang658@purdue.edu

Stop_And_Go Implementation

- Note: all testing should be done with to_send_large.txt as the file being sent
- For all testing results do 5 runs and report mean and standard deviation. Use the format mean[std]
  - Ie. If your mean is 100 bytes and the standard deviation is 10 bytes, then report that as 100[10]

Structure Explanation

My stop_and_go implementation included the use of sending packages that have corresponding header sequence number (each package separated from the txt file has a unique corresponding sequence number). The intuition behind my stop_and_go sender implementation is send one package at a time which has a corresponding sequence number, and then wait for the ackownledgement from the receiver for that package. If acknowledge is received then proceed to send the next package, otherwise when socket.timeout happens just resend the current package again.
The intuition behind my stop_and_go receiver side is keep receiving the packages - I created two variables which is to store the current received package and the previous received package – if no packages received then socket.timeout and do nothing (this will trigger the sender to keep sending the current package), once a package is received it will store this package inside the previous package variable and continue to received the next package. If the next package is out of order which is not the previous_package + 1, then we will keep ackownledging the previous package to force the sender send the previous package + 1 again.

Statistic Report

The recorded goodput rate is 4669.19, 4457.88, 4603.95, 4218.37, 4307.61 / s.
The calculate Mean and Std are 4451.40[190.9]

Stop and go results

	Bandwidth = 200,000 bytes/s	Bandwidth = 20,000 bytes/s	Bandwidth = 2,000 bytes/s
Goodput (bytes/s)	4669.19	4648.28	917.97
Overhead (% of bytes sent)	2288	2288	1152869

Parameters should be:

- One way propagation delay : 100 ms
- Loss is 2% and reordering is 2%

Impact of window size

	Smallest window size = 1 * 1024 bytes	Second smallest window size = 2 * 1024 bytes	Medium window size = 3 * 1024 bytes	Second largest windows size = 5 * 1024 bytes	Largest window size = 10 * 1024 bytes
Goodput (bytes/s)	4478.69	8481.78	11332.44	19514.66	29083.57
Overhead (% of bytes sent)	323580	316874	353377	313380	420480

Parameters should be:

- Bandwidth 200,000 bytes/s
- One way propagation delay : 100 ms
- Loss is 2% and reordering is 2%
- This is testing your protocol, NOT stop and go

Note: Replace X in the above window sizes with the window size you used!!

Your protocol vs stop and go

	Your protocol	Stop and go
Goodput (bytes/s)	29083.57	4669.19
Overhead (% of bytes sent)	420480	2288

Parameters should be:

- Bandwidth 200,000 bytes/s
- One way propagation delay : 100 ms
- Loss is 2% and reordering is 2%
- The window size that works best for your application

Sensitivity Testing [optional for interim]

	5% loss and 5% reordering	2% loss and 2% reordering	No loss and no reordering
Your protocol goodput (bytes/s)	20396.38	29083.57	35028.47
Your protocol overhead (% of bytes sent)	613477	420480	297277
Stop and go goodput (bytes/s)	4126.51	4389.62	4650.69
Stop and go overhead (% of bytes sent)	377641	338077	301140

Parameters should be:

- Bandwidth = 200,000 bytes/s
- Use whichever window size works best for your algorithm

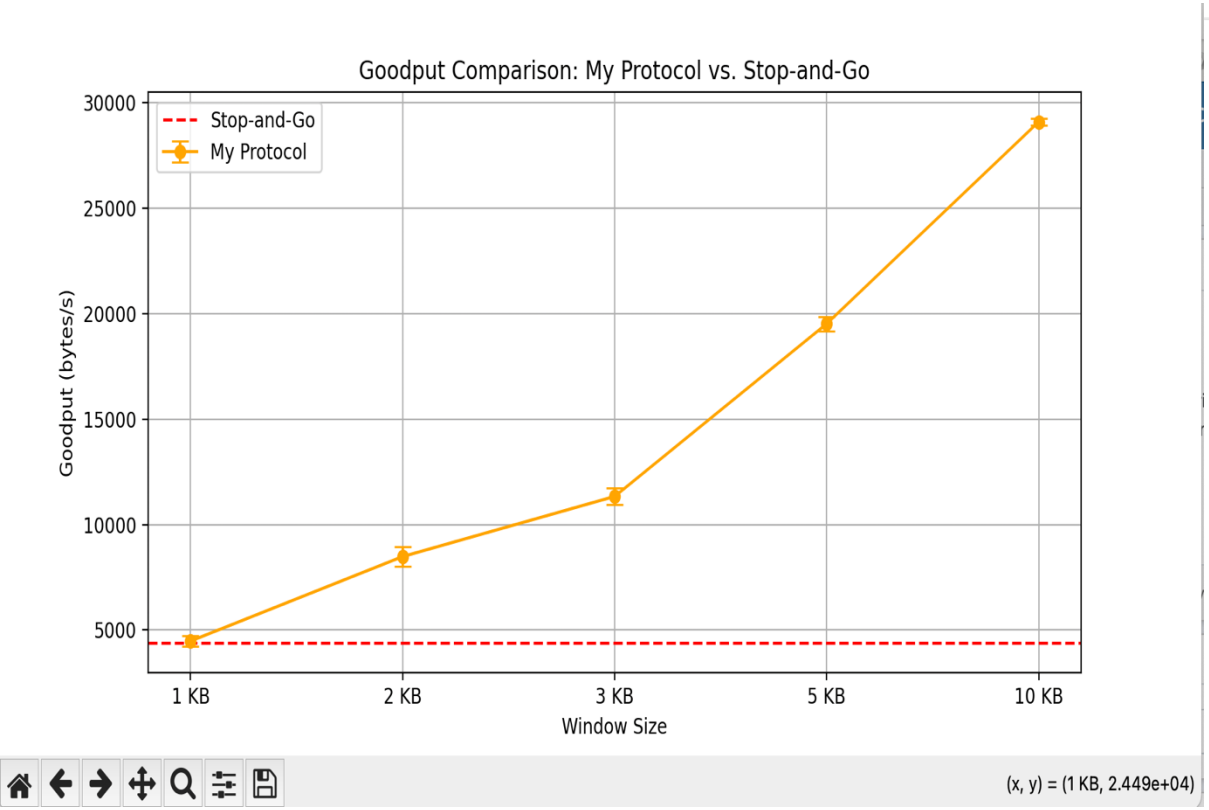
Variance graphs [optional for interim]

- You need graphs for the goodput over 5 runs with your protocol vs stop and go (including variance in the graphs)

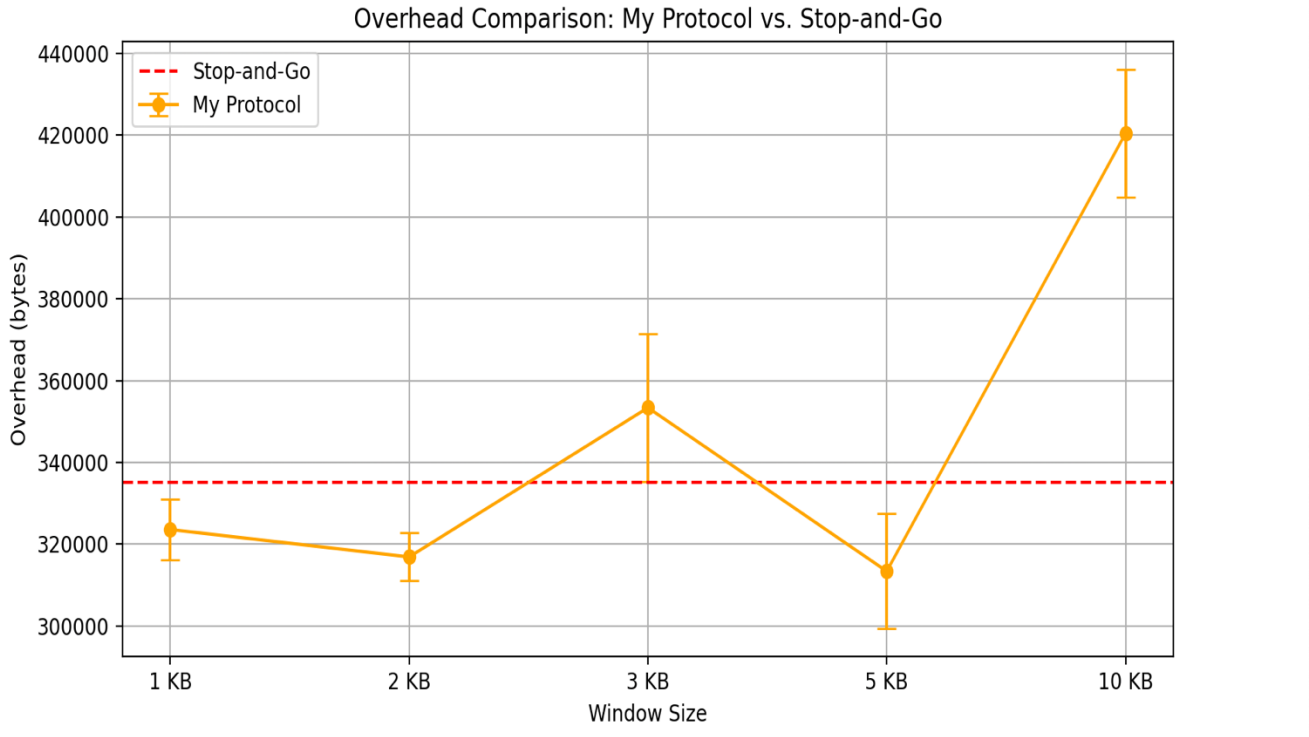
- You need graphs for the overhead over 5 runs with your protocol vs stop and go (including variance in the graphs)

- Parameters should be:
  - Bandwidth 200,000 bytes/s
  - One way propagation delay : 100 ms
  - Loss is 2% and reordering is 2%
  - The window size that works best for your application

Goodput (stop_and_go vs my protocal)



Overhead (stop_and_go vs my protocal)



Note: Stop_and_go is always 1 * 1024 bytes per window size, the yellow vertical line extending above and below each dot on the plot represents the variance

Protocol description (2 slides max)

- How did you set your timeout values?
- What window size did you use? Why?
- Does your code automatically adjust timeout and window size to network configuration? If so how?
- Provide a couple of lines of detail about your ACK scheme.
- Does the sender only retransmit on a timeout, or does any other condition trigger a retransmission in your implementation?
- Please list any other optimizations not listed above that you implemented.

Timeout Setting: Timeout setting for sender is 0.211 seconds. It is based on the RTT time plus the transmission rate. Intuitively, when sender sends a packet it should received the packet after the RTT+transmission time which is 0.1*2 + 1024/bandwidth. Thus, given packet size is 1024 bytes and 200000 bytes bandwidth is 0.200512, i round it up for a more flexible wait, thus i set the timeout for sender to be 0.211. For receiver, i did not implement a explicit timeout in it since the receiver keeps waiting for incoming message and process them, we don't need a timeout for receiver.

Window size: I implemented a flexible window size for my protocal and experimented with a window size of 1 packet each time for stop_and_go protocal. My protocal flexibly reads window size inside the config file. In the first stage of this project, I experimented with a window size of 1,2,3,5,and 10 and see different performances with each parameter.

Flexible adjustment: I did not adjust window size and timeout flexibly, everything is either fixed or determined by what is inside the config file.

Continue to the next page for more configuration details:

ACK Scheme detail Sender Side: the sender send out a package and its corresponding header

```
def create_package(chunk_dict, window_size, loop_counter):
    for i in range(window_size):
        chunk_dict[loop_counter + i] = chunk[i * max_packet_size : (i + 1) * max_packet_size]

def send_packets(chunk_dict, send_monitor, receiver_id, window_size):
    keys_list = sorted(list(chunk_dict.keys()))  # Convert to a sorted list

    for i in range(window_size):
        header = keys_list[i].to_bytes(4, byteorder='big')
        send_monitor.send(receiver_id, header + chunk_dict[keys_list[i]])
    print(f"first chunk sent is {list(chunk_dict.keys())}")
```

ACK Scheme detail on Receiver Side: The receiver receives the data and strips out the header, it decides if this data has been received or not according to its header. After the receiver received a maximum limit portion of the data, it sorts and rearranges the data to see if this protion of the data is intact or not.

```
received_header = int.from_bytes(data[:4], byteorder='big')
print(f"received header {received_header} received data{list(received_data.keys())} previous_data{list(previous_data.keys())}")
if received_header not in received_data and received_header not in previous_data:
    received_data[int.from_bytes(data[:4], byteorder='big')] = data[4:]
elif received_header in previous_data and len(previous_data) == window_size:
    key = sorted(list(previous_data.keys()))
    receiver_monitor.send(sender_id, b'ACK' + max(key).to_bytes(4, byteorder='big'))
    #print(f"send max key again {key}")
    continue
```

```
if first_packet == True and len(received_data) == window_size:
    key = sorted(list(received_data.keys()))
    to_match_list = [ACK_count + i for i in range(window_size)]
    if key == to_match_list:
        first_packet = False
        f.write(b''.join(received_data[k] for k in key))
        receiver_monitor.send(sender_id, b'ACK' + max(key).to_bytes(4, byteorder='big'))
        ACK_count += 1
        previous_data = received_data.copy()
        received_data = {}
        # f.close()
        # break
        #print(f"received first packet key is {key}")
        continue
    else:
        #print(f"firsttime receiving data and key is {key}")
        pass
```

Retransmission Conditions: The sender retransmit the current packages if it is timed out on receiving or if it receiving ACKs of the previous packet which has already been transmitted. In summary, the sender forces the receiver to roll over to its current transmission cycle.

Additional Optimizations: I used a flexible window size (can be user specified inside the config file) with partially selective ACKs on the receiver side. In summary, the sender and receiver can transmit and receive a user specified number of packets each time. The receiver stores each packet inside a dictionary and it classifies this incoming packets into one of the three catogories as – “not received and not inside previous_received data”, “received already”, or “outdated (previous_received)” data. The sender and the receiver both enforces a mechanism to force each other to match with each other’s state of transmission.

Summary of features

Protocol question	**Answer**
Window size (bytes)?	Yes, can be customized in config file
Automatic window size adjustment?	No
Automatic timeout adjustment?	Yes, on the sender side
Cumulative ack?	Yes
Selective ack?	Yes (only partly)
Non-timeout based retransmit?	Yes, on the receiver side

Interim Report only: Discussion

- If this is your interim report, what do you plan on implementing between now and the final report?
- [For final report, don't delete this slide, but instead move to the end as reference].

Intention and plans for further improvement: Given my current implementation, I achieved a baseline of 20000Bytes per second and higher for goodput. However, I am faced with another challenge which is the significance of my overhead value. My overhead value significantly increases especially as loss value factor increases. This is understandable as I am implementing a cumulative ACK; my sender sends N number of packets every time, my receiver receives up to N packets every time; if there are packets lost or if the sender times out before receiving acknowledgements, the sender simply retransmit an entire N packets again and again causing large overhead value.

In my next step, I am going to implement a fully selective ACK combined with cumulative sending technique. My plan is to send N number of packets at a time; and after the receiver receives N – x (0<=x<=N) packets, the receiver simply NACK the packets that are not received which will help the sender to determine to only transmit the lost packets instead of transmitting the entire N packets again which will tune down the overhead value.

Summary of changes: final report only

- Compactly summarize changes since your interim report.

The first version of the protocol sends one packet at a time, waiting for an acknowledgment before sending the next. This ensures reliability but yields very low transmission efficiency (goodput).

The second version improves performance with cumulative and selective acknowledgments via a sliding window (size set in the configuration file). The sender transmits up to 50 packets at once, and the receiver acknowledges only after all are received. While this significantly boosts goodput, it also increases overhead because losing any packet prompts retransmission of the entire batch.

The third version enhances the second by adding negative acknowledgments (NACKs) to selectively resend only lost packets. Instead of resending an entire window on a timeout (as in the second version), the sender now retransmits only the missing packets which can significantly help reducing overhead value. Also, the timeout values are carefully adjusted accounting for propagation delay, transmission rate, and processing time. Moreover, the timeout value is determined by summing twice the propagation delay, the transmission rate, and an additional margin for processing time. It is also worth noting that additional mechanisms have been implemented to clear buffers and handle edge cases which ensures successful executions over all scenarios.

Ablation table (final report only)

- Compare your final design with at least two other alternatives that may have performed less well.
- Example alternatives involve:
  - Turning off a feature (e.g., selective ACK) that was important for performance.
  - A design feature that you tried which (maybe surprisingly) hurt performance

- Feel free to add more rows to the table if necessary

- Parameters should be:
  - Bandwidth 200,000 bytes/s
  - One way propagation delay : 100 ms
  - Loss is 2% and reordering is 2%
  - The window size that works best for your application

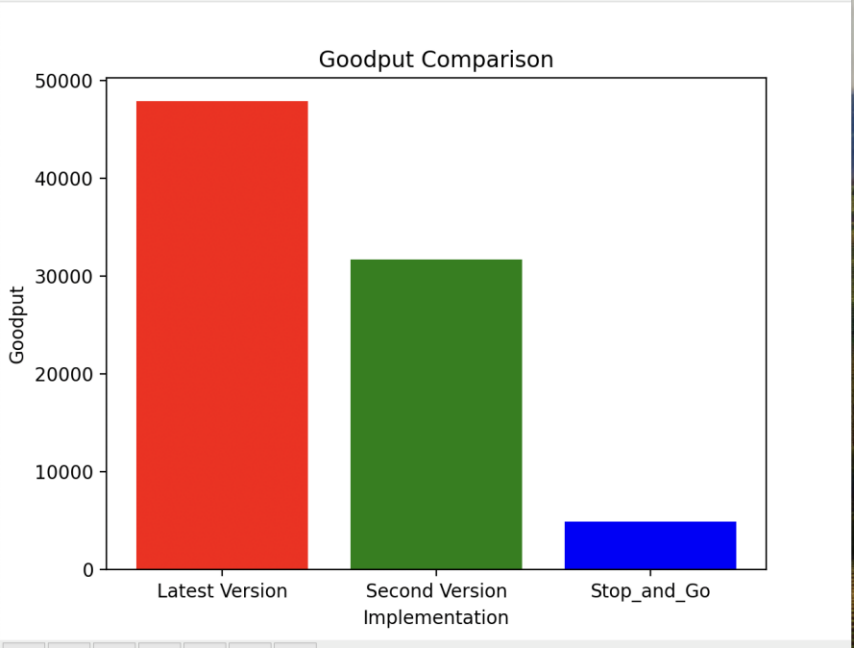
Ablation table (final report only)

Design Alternative Short description E.g., ”Disabling selective ACK.” “Adding feature X”.	Goodput (bytes/s)	Overhead (bytes)
Disabling NACK	Up to 25k to 30k	Up to 320k to 450k
Disabling cumulative ACK and selective NACK together	Up to 4k to 6k	Up to 2k to 3k

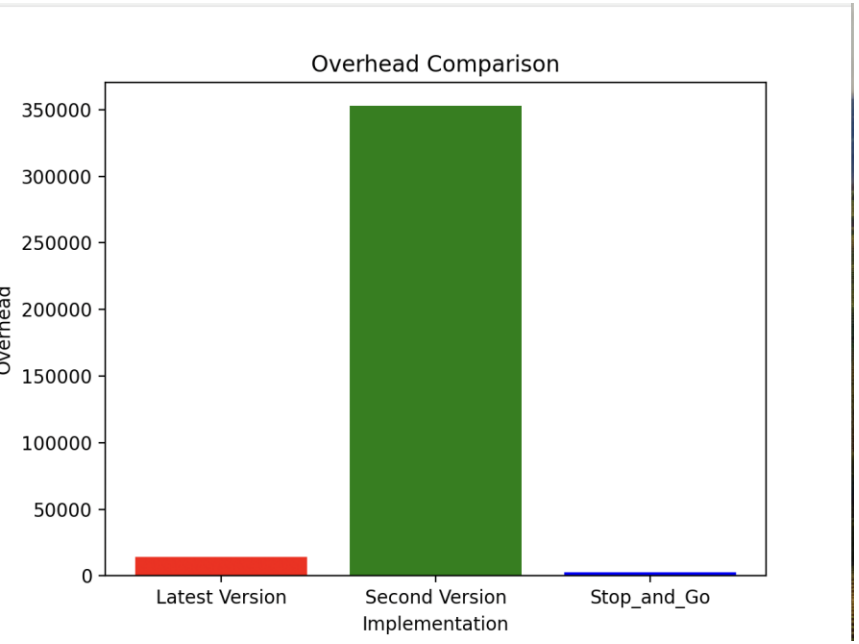
Final report only: Ablation results discussion

- What features were most crucial to achieving good performance? Why?
- Were there features that you thought should help but didn’t work out? Why do you think they didn’t improve performance?
- What ideas do you have for the future (that you did not implement)?

- The main features are NACKs and cumulative acknowledgments. Intuitively, NACKs reduce overhead, while cumulative ACKs improve goodput efficiency, so optimizing both can significantly boost overall performance.
- I also considered TCP congestion control using TCP Tahoe to further enhance performance by dynamically increasing the window size. However, for a small file transfer (such as a homework demonstration), the time required for congestion control to stabilize negatively impacts both goodput and overhead. Hence, I decided it was not appropriate for this assignment. Still, it could be beneficial in scenarios involving larger files and scaled sender–receiver settings. I briefly tested it but eventually abandoned the idea. Nonetheless, congestion control deserves more in-depth discussion.



Goodput Comparison:
I took 10 runs for each of my three protocals including “Stop_and_Go”, “Second_Version” and “Final_Version”. I calculated the average goodput of all of the ten runs for each of the protocal and plotted them against eachother. Out of all the “Final_Version” which is the red bar outperformed all of the other protocals with an average of 47k. The second version has an average of around 31k and the simplest protocol has an average of 4.9k.



Overhead Comparison:
I took 10 runs for each of my three protocals including “Stop_and_Go”, “Second_Version” and “Final_Version”. I calculated the average overhead of all of the ten runs for each of the protocal and plotted them against eachother. Out of all the “Stop_and_Go” which is the red bar outperformed all of the other protocals with an average of 2.3k. The second version has an average of around 350k which is the highest and the Third version protocol has an average of 13k.

Summary

In summary, considering both the goodput and overhead results, Final version (the third version) offers the best overall performance. While the second version significantly increases goodput, it also generates a large amount of overhead by retransmitting entire batches whenever a packet is lost. The first version keeps overhead low but provides extremely poor goodput efficiency, taking a very long time to complete transmissions.