Reliable Data Transfer

Team: joy_maa

Anish Banerjee* Ankit Mondal†

October-November 2023

§1. Checkpoint 1

1.1. Our Method

- To receive the data reliably, we maintain an `receivedlist` initialized with a dummy character. In each iteration, we update the `receivedlist` with the data received from the server. This helps us maintain the data's order, as it is not received sequentially from the server.
- However, the server may refuse to respond to some queries hence we do not expect to receive the entire file at the end of the first iteration. So, we repeat this process multiple times till we receive the entire file.
- We have also calibrated the timeout to a suitable value so that we don't wait too long for skipped requests.
- We have also added a wait period after every message to ensure that messages are sent at more or less uniform rate without resulting in squishing the data.

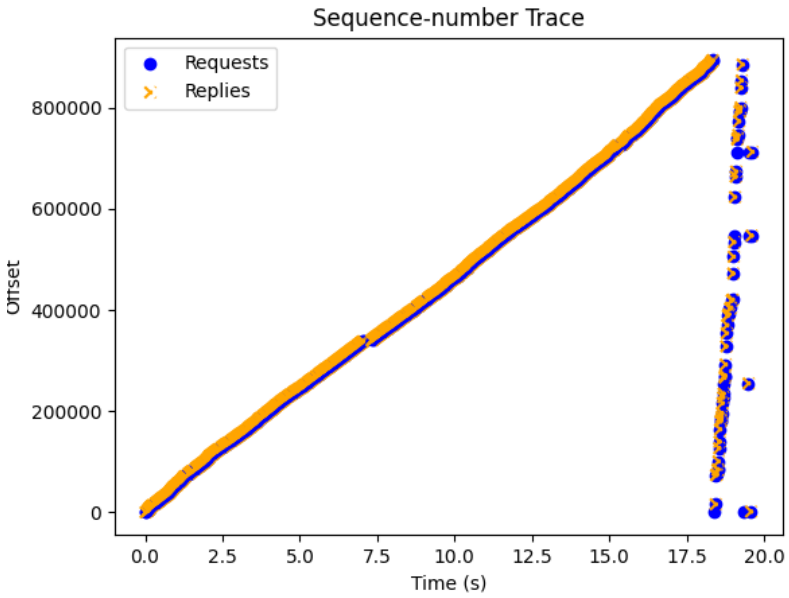

Figure 1: Sequence-number trace

*2021CS10134
†2021CS10229

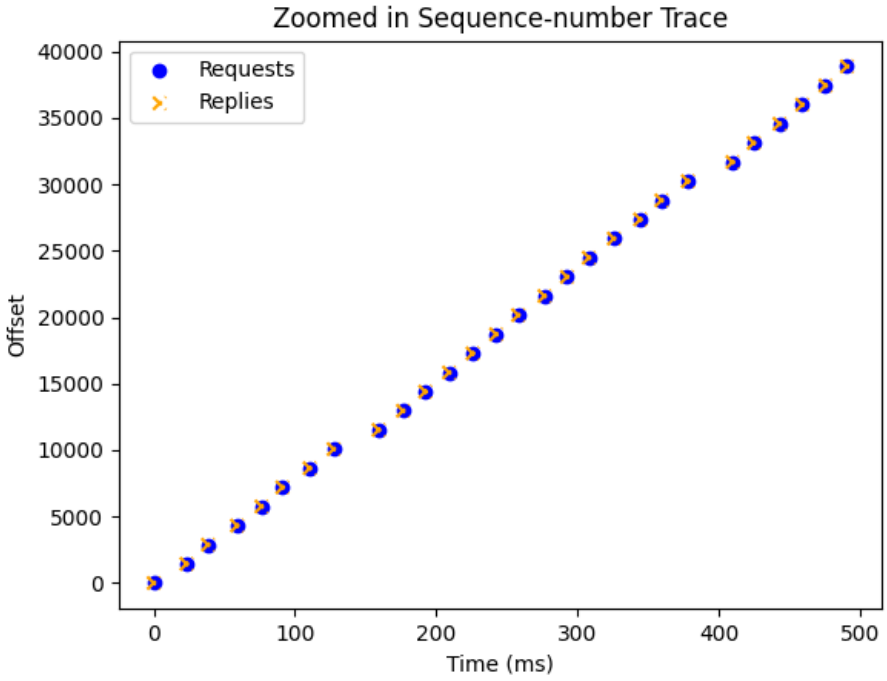

Figure 2: Zoomed in Sequence-number trace

1.2. Graphs

- Fig. 1 shows the sequence number trace for an approx 0.01 MB file. Requests are shown in blue and replies in orange. The replies appear to almost overlap with the requests. As mentioned in our method above, we make multiple passes to obtain the entire file. This is evident in the graph too - approximately,
  - The first pass starts at 0s and continues till 17.5s,
  - The second pass starts at 17.5s and ends at 18s and,
  - The last pass starts at 18s and ends almost immediately as it sends just 4 requests.
- Fig. 2 shows a zoomed-in version of the above trace for requests and replies within 500ms from the initial time. From this too, we observe that the requests and responses are almost overlapping. There is a very small round trip time and in a few messages **there is a visible gap between the query and reply**. As the RTT increases the gap between 2 queries also rises (as time between successive queries is round trip time + sleep time per message). Some **queries are dropped midway**. These queries are raised again in the subsequent passes.

1.3. Appendix: Other Attempts

We also tried implementing a stop-and-wait protocol in which the sender keeps sending requests till it receives a particular line. When a particular message is dropped, it keeps sending the same request till the response is received. We also tried sending the requests in burst sizes, decreasing the burst size if we had to wait to long for a particular message.

Result: The time to receive file decreased but penalty increased as some messages are squished while trying to find the correct rate to send the requests. Thus for the time being we move ahead with the safer code mentioned at the beginning of the report. The overall graph is also showing much less linearity as compared to previous uniform rate version.

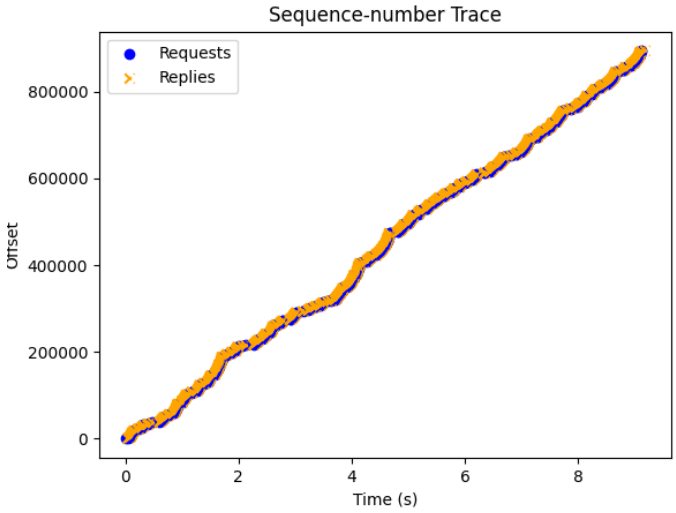

Figure 3: Sequence-number trace 2

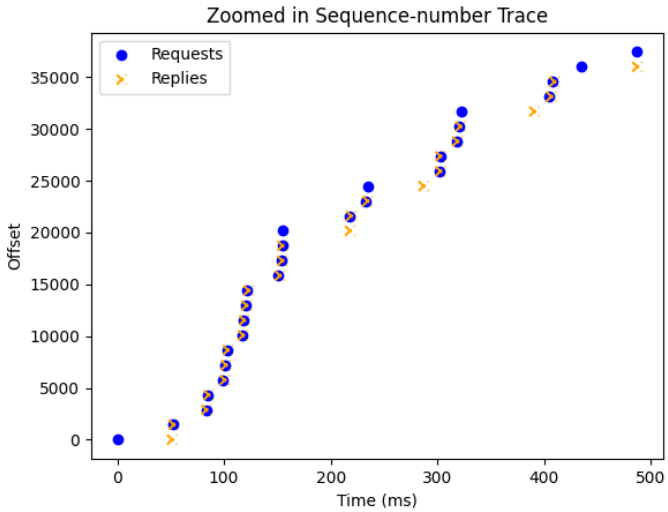

Figure 4: Zoomed Sequence Number Trace 2

§2. Checkpoint 2

2.1. Our Method

We made the following modifications to the earlier approach:

- **AIMD:** We send requests in bursts whose size is controlled by AIMD:
  - If we receive responses to all the requests, we increase the burst size $b = b + 1$
  - However, if the difference between the number of requests sent and the number of responses received is large, it means that we might be exceeding the number of tokens in the bucket. When this difference crosses a certain threshold (determined by the RTT), we increase the burst size by $b = b + \frac{1}{b}$ instead of 1. This helps to decrease the penalty.
  - If some requests are lost, or we get squished, we multiplicatively decrease the burst size $b = \frac{b}{2}$
- **Time between two bursts:** If the burst size is stuck at 1, it means that our rate of sending bursts is too high for the server, even if we send one burst at a time. So, we increase the sleep time between two bursts.
- **EWMA:** To measure the RTT, we use EWMA:

  $$\mathsf{RTT} = \alpha\mathsf{RTT} + (1 - \alpha)\mathsf{sampleRTT}$$

  where $\alpha = 0.125$
- **Optimal Sleep Time:** We performed a binary search for optimal sleep time on Vayu (≈0.008s). However, the same value did not work for Localhost, where the optimal value was ≈ 0.02s. The fractional increase approach helped us adapt our code to both situations since the burst size never increased too much.

2.2. Graphs

Fig. 5 shows the graphs obtained for the constant rate server running at Localhost and Fig. 6 shows the graphs obtained for Vayu 9801.

Observations

- **Total time taken** to obtain the file for Vayu is around **18s** while that for Localhost is around **25s**. The time for Localhost is more than that of Vayu because the file sizes differ (Localhost ≈ 1,00,000 lines; Vayu ≈ 50,000 lines).
- **Penalties** for Vayu is **9** while for Localhost it is **52**.
- In both cases, we require roughly 4 passes to obtain the entire file.
- The requests and responses almost overlap with each other in the graph for Localhost. On the other hand, the difference between sending and receiving is more visible in the Vayu graph. This is because the RTT for Vayu will be more than that for Localhost.
- The burst sizes are seen to increase when the responses for the requests are received. However, the burst size gets halved when the responses are not received.
- If more requests are not received, the time difference between bursts increases.
- In the graphs for burstsizes v/s time, we observe that the burst size decreases suddenly after a certain point (near the completion of the second pass). This is because we are traversing through the received list sequentially in blocks, and as most of the packets have been received, we don't send requests for them again.
- We don't get squished in either of the two cases
- Compared to the (hacky) fixed-rate version implemented in the appendix of the previous section where wait time was hand-fixed by us, the time taken now and penalties are slightly larger. This is because the code incurs some penalty while finding the right rate to send the messages.

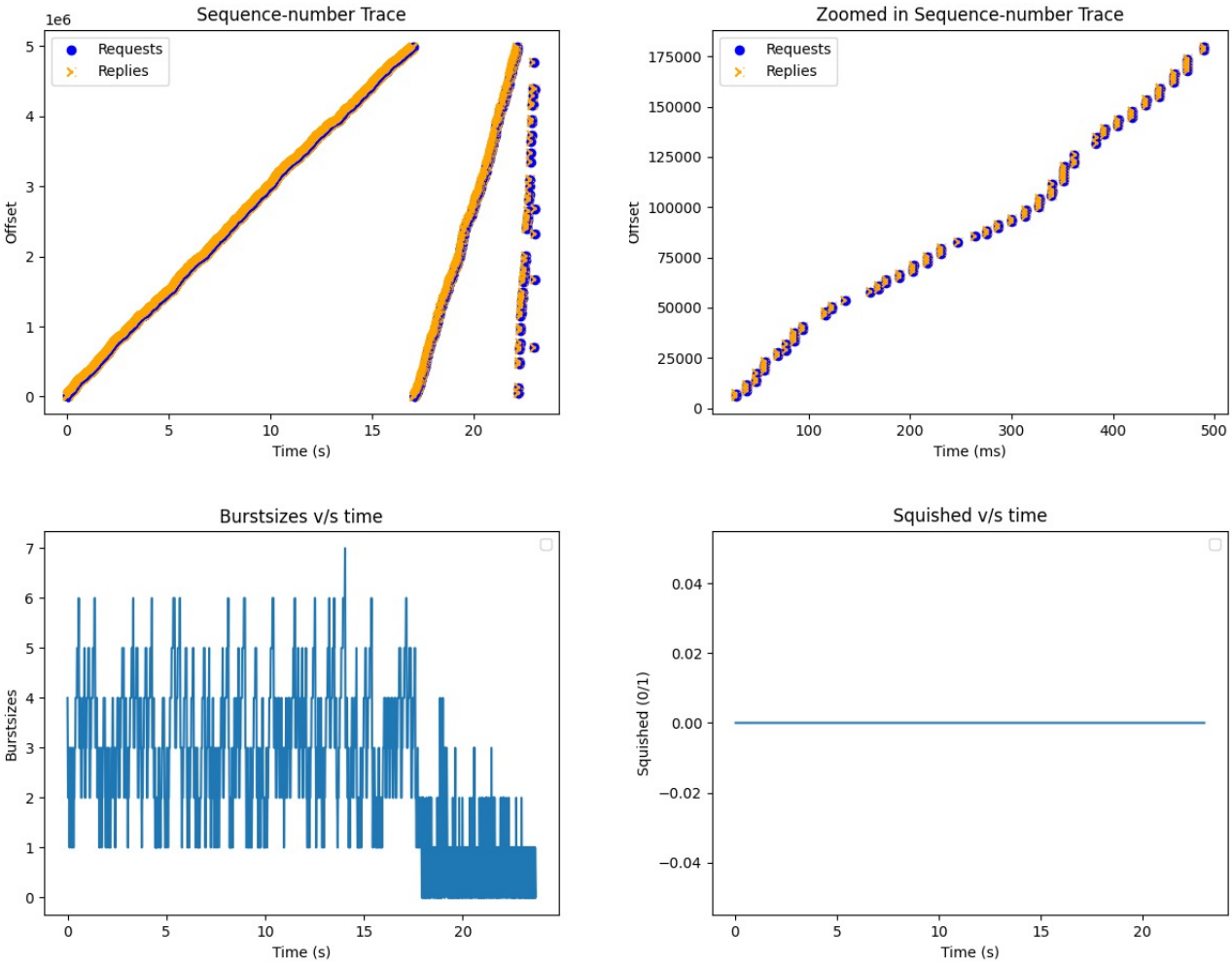

Figure 5: Graphs for Localhost

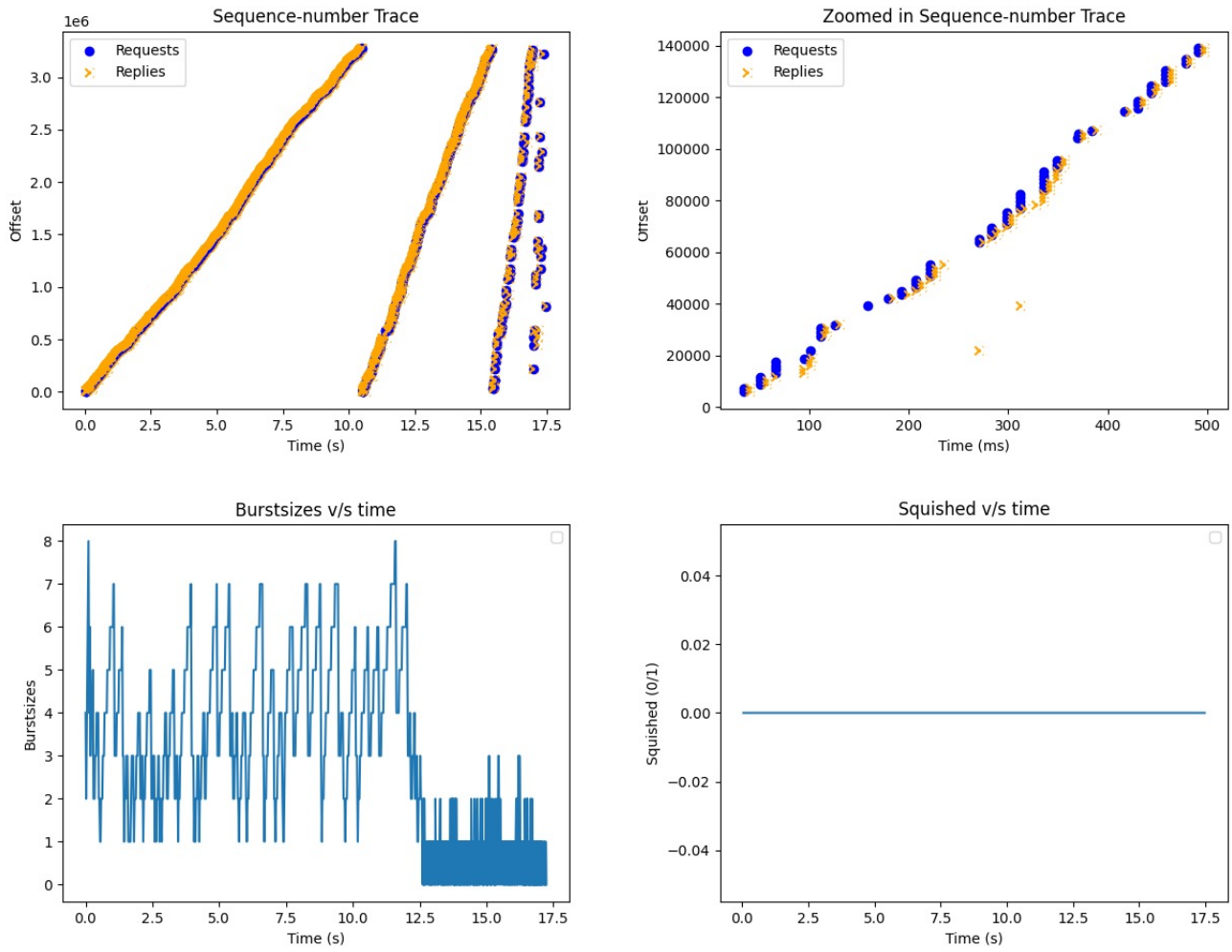

Figure 6: Graphs for Vayu