# Exercise 2 - Final Design Document

Erfan Sharafzadeh Farnaz Yousefi
e.sharafzadeh@jhu.edu f.yousefi@jhu.edu

10/23/2019

## 1 INTRODUCTION

This document provides our final design for implementing a UDP/IP-based reliable, ordered multicast application. This protocol should handle message omissions in different loss rates, in the ugrad lab network. Also, the protocol will handle total ordering in a FIFO manner. The expected testing scenario is composed of eight machines running the mcast application, six of which are sending a number of packets and all of them will store them in the same order. The mcast applications will start their operating by a triggering signal from the mcast_start application.

## 2 PROTOCOL DESIGN

This section presents the final design of our multicast protocol. We base our design on one of the **Free Access protocols** called **Lamport Timestamps Protocol** and will elaborate on the Ordering, Reliability, and performance concepts in the following sections.

Each process has a sending window and a special data structure (refer to 2.1) for the received packets. The processes stamp a local incremental sequence number (index) and their Lamport timestamp containing their unique *Process ID* and their local counter when sending a data message. Upon receiving one data message, each process stores the data in the data structure along with its sequence ID. The processes then deliver packets (based on their Lamport counter and process id)when the delivery conditions (addressed in Section 2.1) are met.

**In our initial design**, each process issued an order message at specific time spots with the last ordered sequence id received from each processor. These order messages were stamped with an integer value called *order ID* and in each round, each processor would wait for the order message from all others and would store the minimum set of the received ordered messages in its file.

**The problem with our initial design was** that we were actually implementing a safe delivery instead of total ordering. We figured that we could deliver messages only by looking at *Lamport counters*. We didn't need to use the order messages which decreased performance by synchronizing all the machines.

### 2.1 ORDERING

To ensure total ordering, we deliver a received packet to file only when:

- We have next-to-be-delivered index from all processes . or,

| TYPE | PID | LAST_DELIVERED_CTR | INDEX | Lamport_CTR | RANDOM_NUM | PAYLOAD |
|---|---|---|---|---|---|---|
| 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes | 1400 bytes |

Table 2.1: data packet structure

- For the processes that we don't, we know that they are finished sending their messages and we have delivered them all.

When these conditions are satisfied, we deliver the data with the lowest *Lamport counter*. When two or more processes have data with equal *Lamport counter*, we deliver them from the lowest *process id* to highest. We allow the operating system's lazy writes (without explicit flushing) in order to avoid extra I/O blocking which might hurt the latency.

## 2.2 Reliability

To ensure reliability, whenever we receive an out-of-order packet from a process, we send a message called NACK (with the structure shown in 2.3) which contains all the indexes that we have missed from that process. Upon receiving this NACK message, the process would retransmit the missed packets. There is also another type of message called ACK (2.4) which is only sent by the processes that are only receivers. These messages contain the sender's last delivered Lamport counter. Note that for the sender processes, this counter is piggybacked on all their messages.

## 2.3 Performance

Since we are not using a ring-based protocol, our processes will send their messages simultaneously as long as others are receiving the messages and acknowledging them by reporting their last delivered counter. To control the transmission speed we can alter the following parameters during the runs:

- The size of the sending/receiving window (a.k.a. buffer size)

  We tuned our window size altering it from 50 to 150. For the current combination of variables, we decided it to be 80.

- The delay in sending NACK messages for the out-of-order data.

  Blindly sending nacks for every out-of-order data will cause a huge storm in the network.Therefore, we implement a timestamping solution by adding a timeval for each data slot in our receive window. Whenever we send a NACK for that data, we store the *timeofday* in the timeval, and when we receive an out-of-order message, and the slot needs to be NACKed, we ensure that we have waited for a specific amount of time since our previous NACK (for our case it is 1ms).

- The delay in responding to repeated NACKS.

  When a process receives a NACK message, it will wait for a specified time since previous data retransmission. The current value for our experiments is 4ms and it ensures that if we receive multiple NACKS for a single data from multiple processes, we don't congest the network with repeated retransmissions.

The decided values for each of these variables are based on multiple runs, and benchmarks on the ugrad lab. Also, using Unicast for poll and feedback messages would considerably improve the transmission performance. However, in our current implementation, we only use Multicast for all message types.

## 2.4 Message Format

We use multiple message types in our reliable ordered protocol.

- **START** indicates the start of sending and is generated by the start_mcast process.
- **DATA** is the general messages type for sending data.

| TYPE | PID | LAST_DELIVERED_CTR | POLLED_PID |
|---|---|---|---|
| 4 bytes | 4 bytes | 4 bytes | 4 bytes |

Table 2.2: poll message structure

| FEEDBACK_TYPE | PID | LAST_DELIVERED_CTR | NUM_OF_NACKS | NACK_INDICES |
|---|---|---|---|---|
| 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4*NUM_OF_NACKS |

Table 2.3: NACK message structure

- **FINALIZE** indicates the last data packet.
- **POLL** is sent when we timeout; either when we don't receive anything from an unfinished process, or from anyone.
- **FEEDBACK** is either ACK or NACK for sending feedbacks.

Tables 2.1, 2.2, 2.3 and 2.4 present the packet structure of data, poll, NACK and ACK messages respectively.

## 2.5 Data Structures

As each process will act as a sender and a receiver concurrently, we need two main structures for storing the send window and the receive window.

All of these windows are stored in a dataMatrix which is an array of pointers to array of windowSlots of the size *windowSize*. However, the way we handle shifting in the sending window is different from how we handle receiving windows of other processes. For the send window, only when we shift the windowStartPointer that we know all processes have delivered until that point.

Finally, Tables 2.9, 2.10 and 2.8 depict the status of the mcast programs, the window slot and session respectively.

## 2.6 Experiments

We run the scenario with 8 machines, six of which sending 16000 data packets and all of machines receiving and storing the data. We present two experiment results in this report.

First, with the uncongested ugrad lab, we have been able to run our protocol with different loss rates at least once (in the presence of The course TA!). However, when we tried to run all benchmarks required for the report, due to high congestion of the ugrad lab network and simultaneous benchmarks, we weren't able to reproduce the expected results. We present both results in the report for your reference. Tables 2.6 and 2.5 present our benchmark results for uncongested, and congested networks respectively. Figures 2.6 and 2.6 present our benchmark results for uncongested, and congested networks respectively.

Based on our results, and based on the number of different message types sent by each process, we understand that introducing packet loss will cause high amounts of re-transmissions. In a no-loss scenario, we will have re-transmission for one fourth of the total number of data packets. This will grow linearly as we add the loss rate. This re-transmissions, and repeated NACK requests contribute to the most delays in our protocols.

On the other hand, reaching the optimal results will require tuning all parameters and is also dependant on the number of machines we are experimenting on.

| FEEDBACK_TYPE | PID | LAST_DELIVERED_CTR |
|---|---|---|
| 4 bytes | 4 bytes | 4 bytes |

Table 2.4: ACK message structure

| Run | 0% | 1% | 2% | 5% | 10% | 20% KILLED |
|---|---|---|---|---|---|---|
| 1 | 11.557 | 30.738 | 45.424 | 57.528 | 61.205 | KILLED |
| 2 | 13.6 | 31.544 | 38.8 | 54.297 | 66.887 | KILLED |
| 3 | 41.627 | 35.715 | 36.55 | 49.231 | 61.302 | KILLED |
| 4 | 34.403 | 55.540 | 40.807 | 49.374 | 67.230 | KILLED |
| 5 | 42.713 | 41.581 | 58.787 | 43.72 | 65.946 | KILLED |

Table 2.5: Last Experiments Results in very congested ugrad lab (in seconds)

| Run | 0% | 1% | 2% | 5% | 10% | 20% |
|---|---|---|---|---|---|---|
| 1 | 11 | 16 | 21 | 23 | 35 | 48 |

Table 2.6: Best Experiment Results in ugrad lab (in seconds)

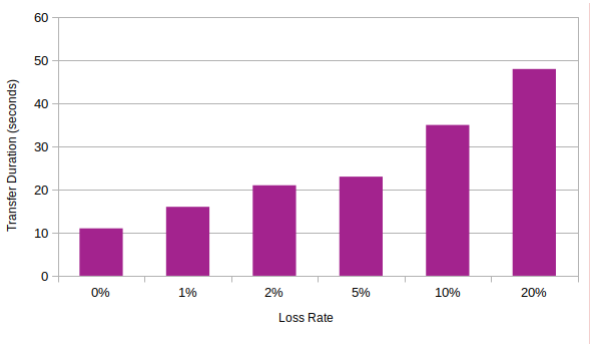


Figure 2.1: The best results we cot in uncongested ugrad network

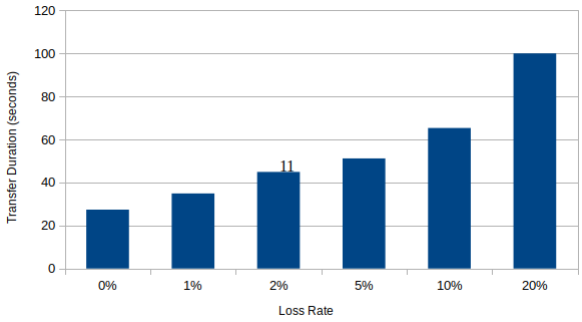


Figure 2.2: The results we got in very congested ugrad network

| | | | | |
|---|---|---|---|---|
| $p_0$ | slot struct | slot struct | slot struct | ... |
| $p_1$ | slot struct | slot struct | slot struct | ... |
| $p_2$ | slot struct | slot struct | slot struct | ... |
| $p_3$ | slot struct | slot struct | slot struct | ... |
| ... | | | | |

Table 2.7: Window structure

| Session struct. | | |
|---|---|---|
| Field Name | Data Type | Description |
| state | enum STATE | |
| numberOfMachines | int | |
| delay | int | |
| dataMatrix | windowSlot** | contains sending window and receiving windows |
| lastInOrderReceivedIndexes | int* | highest indexes up to which all messages are received in order |
| highestReceivedIndexes | int* | |
| windowStartPointers | int* | |
| lastDeliveredCounters | int* | is updated by receiving any message from a process |
| lastDeliveredIndexes | int* | last delivered index of each process |
| isInLastWindow | int* | is set 1 for each process upon receiving finalize message |
| lastExpectedIndexes | int* | is set to index of the finalize message (last packet) |
| timoutTimestamps | struct timeval* | |
| localClock | int | |
| machineIndex | int | |
| numberOfPackets | int | |
| lossRate | int | |
| windowSize | int | |
| lastSentIndex | int | |
| lastDeliveredPointer | int | |
| sendingSocket | int | |
| receivingSocket | int | |
| file | file* | |

Table 2.8: The struct format of Session

| Status ENUM |
|---|
| WAITING |
| SENDING |
| RECEIVING |
| FINALIZING |

Table 2.9: The Status ENUM

| Window Slot Struct. | |
|---|---|
| Field Name | Data Type |
| index | int |
| randomNumber | int |
| LamportCounter | int |
| valid | byte (0 or 1) |
| $fb_timer$ | timeval |

Table 2.10: The struct of Window Slot