Design Document for CS425/ECE428 MP1

Zhuo Li[1], Ruiqi Zhong[2]
1. Dept. of Computer Science, UIUC
2. Dept. of Electrical and Computer Engineering, UIUC
Email: {zhuol2, rzhong5@illinois.edu}

1 Overview

1.1 Functionality

In this Machine Problem, we implemented a simple chat room with **tolerance to multiple node failures** and **total ordering guaranteed**. For each node running on different virtual machines, it can take user input from keyboard and send it to all the nodes which are connected. Each node can leave or fail at any time, and all the other nodes will detect its leave or failure with notification to the user. All the functionalities will remain well behaved among non-failure nodes.

1.2 Network Protocols

For message transmission and failure detection, we both use **TCP** as the transport layer protocol which enhance a FIFO and packet loss/delay tolerant communication in each channel.

As we also implemented total ordering (i.e. All the messages will be delivered in the same order on each node), the multicasts within this chat room distributed system are hybrid FIFO-total ordering.

Failure detection, in practice, is usually implemented with UDP which causes less bandwidth. Some randomized algorithms are used to select only part of nodes to send ping/ack or heartbeat in each round. In our implementation, we make it simplified via TCP channel shared with normal messages. This configuration is only for experiments, practical case should be more complex.

2 Algorithm Designs

For both failure detection and total ordering, there are more than one way/algorithm to implement. In this section, we will take a deep look at the algorithms we choose and the reason why they stand out among others in our work. Pseudocodes will be provided for implementation details at the end of each sub-section.

2.1 Failure Detection

There are two basic ways for failure detection: Ping/Ack and Heartbeat. In our implementation, we choose **Heartbeat** because it uses only 1 message every T time period comparing with 2 messages per T time period for Ping/Ack.

As for topological design, we decided to choose **all-to-all** algorithm to enhance detection for failures in the system. Even though the all-to-all algorithm consumes much more bandwidth than centralized or ring-based algorithms, it prevents the case in which single point failure or multiple nodes failure happens can not be detected. All-to-all algorithm makes our system robust to single/multiple failures.

2.1.1 Algorithm

Our all-to-all heartbeat algorithm works as the following:

For each node (e.g. Node A), once set up TCP connection with another node (e.g. Node B), start sending heartbeat every $T = 100ms$ time period. It also set up a timer task for detecting heartbeat from B every $T + \Delta_2 = 200ms$ time period. Here we assume the maximum one-way delay is $100ms$. After setup connections for all other nodes, we will have a list of failure detection timer task for each other node.

Once we receives a heartbeat from another node (e.g. Node B), we clear its timer task in our list, and restart it for another $T + \Delta_2 = 200ms$.

If we didn't hear the heartbeat from another node for a while (i.e. its timer task is not reset by our detector), we predicate this node has died. We then clear our heartbeat for that node, close connections with the node. We also delete the node from our multicast list, and send a multicast to all other nodes informing them the failure.

Once a node (e.g. Node C) receives an informing failure (i.e. B has failed), no matter it is detected by itself or not, it should also consider the node has failed. It clear heartbeat, close connection and notify the user.

2.1.2 Implementation Details

Pseudocodes are provided as following (HB stands for Heartbeat):

```
1 Initialization at N_i when setup connection with N_j:
2   init Heartbeat timer task, send(N_j, [HB], T)
3   init Detector timer task H_j for [HB] from N_j
4
```

```
 5 When at N_i, H_j time's up (Consider N_j fails):
 6   Heartbeat[N_j].clear()
 7   Close connection with N_j
 8   Multicast(g, ([FAIL], N_j))
 9
10 When receives [HB] from N_j:
11   reset and restart H_j
12
13 When receives ([FAIL], N_j):
14   Heartbeat[N_j].clear()
15   Close connection with N_j
```

2.2 Total Ordering

There are mainly two kinds of approaches for the implementation of total ordering: sequencer-based approach and ISIS algorithm. We choose the **ISIS algorithm**. For that although the ISIS algorithm may have a higher latency for receiving all proposed priority semaphores than sequencer method, the sequencer method will encounter bottleneck when the number of group members becomes increasingly large and the multicast is frequent. Also, if the sequencer is failed, the total ordering could not be guaranteed. However, in ISIS algorithm, when one Node fails, we will reduce the number of proposed priorities the message sender is waiting for to acquire agreed priority for each message to guarantee total ordering.

2.2.1 Algorithm

Our ISIS algorithm for total ordering algorithm works as the following:

We give each Node a unique nodeId, which is marked by its port number and affiliated to all semaphores sent by this Node. A Message class is defined to store the content of a message, a label to mark if the message is deliverable, the original proposed priority of a message which is affiliated to all semaphores to uniquely identify the message, and the current priority of this message. The priority array consists of two Integer elements, one is the net priority and the other is the port number, which marks the presenter of the net priority. For our semaphores, we use "[M]" to mark message, use "[AP]" to mark agreed priority, use "[PP]" to mark proposed priority and use "[OP]" to mark original priority for the convenience of identification.

We use a sendList, which is a PriorityQueue structure, to store all messages sent and received by this Node in order to decide the sequence of delivery at this Node. Also, a msgList is used to store all messages sent by this Node, whose entries are also PriorityQueue structures corresponding to messages to store proposed priorities received by sender and determine the agreed priority for every message that is multicast. Every time we add a Message into the

PriorityQueue, it will be inserted to the right place according to its proposed priority value. The variable total_priority is used to calculate the up-to-date proposed priorities for messages which should be calculated as the following formula:

$$total\_priority = max(all\ observed\ agreed\ priorities, any\ previously\ proposed\ priority)$$

$$proposed\_priority = total\_priority + 1$$

As the message sender, before the Node multicasts message from keyboard input to all other Nodes, the original proposed priority for this message is calculated based on total_priority of this Node. The semaphore which is multicast is wrapped by nodeId, sign "[M]" and its original proposed priority for this message. A Message class is created for this message, put into the sendList and the PriorityQueue entry created for the message itself in msgList.

When a Node receives a semaphore containing a message along with its original priority, it will propose a priority according to total_priority and send a semaphore back to the message sender containing its proposed priority and original proposed priority. Also, the Node needs to store the message along with its proposed priority into its sendList.

If message sender receives a semaphore containing proposed priority and original priority, it will create a new Message class with the proposed priority and add it to the PriorityQueue entry corresponding to this message in msgList. If the size of the entry is equal to the number of nodes, then agreed priority will be calculated. The current priority of the corresponding Message in the sendList should also be updated with the agreed priority for this message. Then, we update the total_priority before we deliver all the messages deliverable at the front of the sendList.

If the Node receives a semaphore containing agreed priority and original priority, it will find the corresponding Message in the sendList and update its current priority. The total_priority for this Node is updated before we deliver all the messages deliverable at the front of the sendList.

2.2.2 Implementation Details

Pseudocodes are provided as following:

```
 1 When keyboard input:
 2   Proposed priority = max(all observed agreed priorities, any
previously proposed priority)
 3   multicast(str + proposed priority)
 4   sendList.add(Message)
 5   msgPriorityQueue.add(Message)
 6   msgList.add(msgPriorityQueue)

 7  When receives [M]:
```

```
 8    Reply with proposed priority
 9  sendList.add(Message)

10 When receives [PP]:
11   msgPriorityQueue.add(Message with [PP])
12   If(msgPriorityQueue.size() == number of nodes)
13     Agreed priority = max(all proposed priorities)
14     Update current priority of Message in sendList
15     multicast(agreed prioity + original priority)
16     Update total_priority
17     Deliver all deliverable messages at front of sendList

18 When receives [AP]:
19   Update current priority of Message in sendList
20   Update total_priority
21   Deliver all deliverable messages at front of sendList
```

3 System Evaluation

3.1 Metrics

We measure the system performance in terms of failure detector bandwidth and the failure detection time, which are two important criteria for communication network system evaluation.

3.2 Evaluation Methods

3.2.1 Preparation

There are always time skews between different machines, which has a terrible effect on our measurement accuracy of almost all metrics that relies on time (e.g. failure detection time, throughput, bandwidth, etc). To ensure the precision of our evaluation, we decide to use **ntp** tools (an implementation of NTP protocol) on each virtual machine.

The synchronization method is by following the configurations below:
https://docs.oracle.com/cd/E26996_01/E18548/html/manager_ntpconfig.html

After synchronization on the 10 virtual machines (adjustment within $\pm 5ms$), we can trust the accuracy shown on each node during the experiment.

3.2.2 Bandwidth

We measure the bandwidth with the number of messages sent out by the failure detector in unit time. For our system, the total bandwidth should be of $O(N^2)$ complexity, for that we use **all-to-all algorithm** and there are N nodes, each of which multicasts heartbeat continuously to $N-1$ nodes. The total bandwidth for our system will increase in $O(N^2)$ trend as the number of nodes N increases.

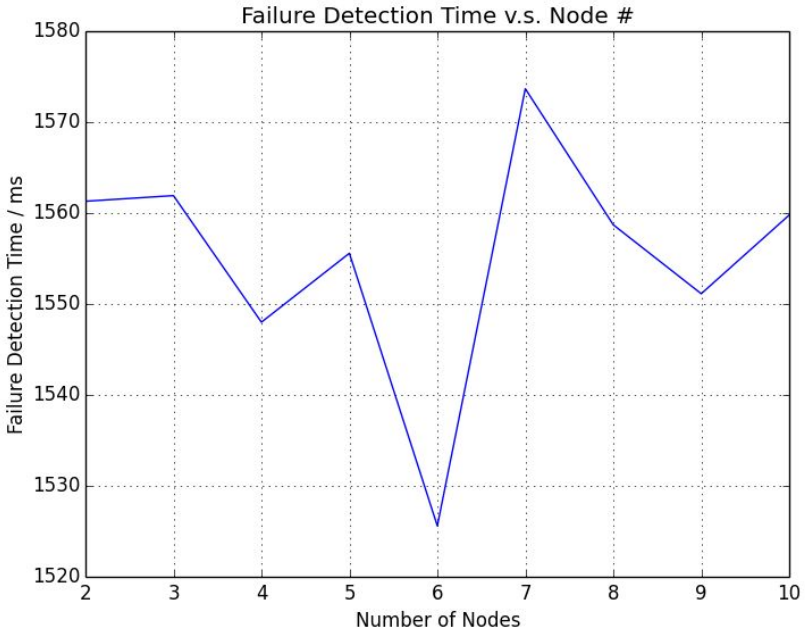
3.2.3 Failure Detection Time

We measure the failure detection time by the interval between one Node exits and another Node detects its failure. The unit time is measured in ms.

3.3 Evaluation Results

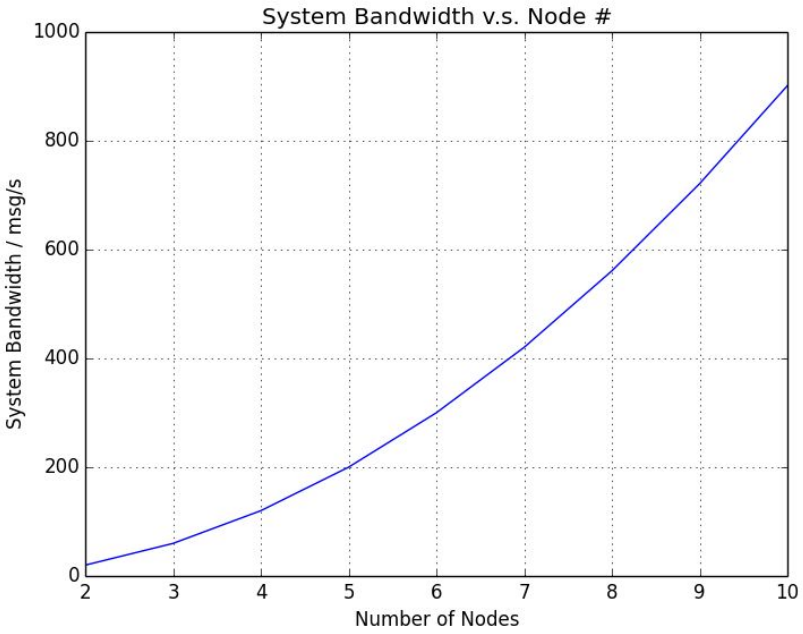
3.3.1 Bandwidth

The horizontal axis represents the number of nodes in our chat room, the vertical axis represents the number of messages sent to all other nodes in $10000ms$, which is expected to be $N*100$ for a node. The total bandwidth for the communication system should be $N*(N-1)$ messages per unit time.



3.3.2 Failure Detection Time

The horizontal axis represents the number of nodes in our chat room, the vertical axis represents the average failure detection time in ms in our chat room with certain number of nodes.



Confidence Interval & Confidence Probability for failure detection time to node numbers

	2	3	4	5	6	7	8	9	10
Confidence Interval	[1525, 1600]	[1500, 1590]	[1510, 1575]	[1505, 1600]	[1505, 1550]	[1560, 1600]	[1540, 1585]	[1510, 1600]	[1525, 1600]
Confidence Probability	80%	91.61 %	83.3%	83.3%	80%	91.67 %	85.7%	93.75 %	88.89 %

3.3.3 Analysis

We find that failure detection time will keep stable as the number of nodes increases, which proves the scalability of our system. However, the result shows the bottleneck of all-to-all algorithm in failure detection, that is, with the increase of node numbers, the bandwidth will increase in $O(N^2)$ trend.

4 Summary

We implemented a simple chat room with tolerance to multiple node failures and total ordering guaranteed. We successfully use Heartbeat and all-to-all algorithm to implement the failure detection function and implement ISIS algorithm to guarantee total ordering in chat room. From our evaluation results, we see that our implementation perform well and demonstrates reasonable failure detection time and bandwidth as we expected.