CSCI 351 Project 3 Report - Simulating Synchrony

Alex Bigley (amb068) Warren Wang (wmw015)
Quan Nguyen Tu (qnt001) Andrew Passero (ajp031)

May 2, 2023

1 Problem 1/Problem 2 Analysis

English Explanation of Solution: We will have all processes increment their round numbers at arbitrary time intervals. They will always be waiting to get a low-level receive. When they get a receive from another process, they will look at the round number tagged to the sequence number (each message will be of form $< msg, round >$).
Assume that the current process P_i is in round r and is receiving a message from P_j in round k.
Two cases must be covered when a process receives a message:

1. The message is tagged with a round number from the future (i.e. k ¿ r). The current node will have to hold onto this message until it reaches r = k + 1 round to announce that it has received it. The message was thus sent from P_j at round k and P_i was supposed to receive it at round k+1.

2. The message is tagged with a round number from the past (i.e. k ¡ r). The current node can just announce that it has received the message in the next round r+1.

Complexity:

`SynchP` has a constant time complexity $O(1)$ because all is does it fire off a low level send. `SynchP` has a constant message complexity $O(1)$ because the graph is fully connected and only requires a single, low-level send.

`Asynch_Receive` can have up to $O(n)$ time complexity if its backlog is sufficiently large enough from every other process, so it may have $n - 1$ messages backlogged from the future if this particular process is really slow. Then, it will loop through all those backlogged messages at every call of `Asynch_Receive` to see if it can announce any receives from the particular process j.

Correctness: Whenever a process receives a message it will always check that message's tagged round number from the sender with its round number. If the tagged round number is smaller than its round number, the process will know that it is valid to announce that it has received that message and will

Algorithm 1 SynchP and AsyncP pseudocode

```
1: SynchP(x, j):
2:     Send x to j using low level send
3: Asynch_Receive(x,j):
4:     contents, sent_round = x[0], x[1]
5:         if sent_round < ith_round:
6:         Announce received x from j
7:     else:
8:         backlog.append(x)
9:     for x in backlog:
10:        contents, sent_round, sender = x[0], x[1], x[2]
11:        if sent_round < ith_round:
12:            Announce receive x from sender
```

do so.However, if the tagged round number is larger than its round number, the process will know that it must wait until its round number has sufficiently advanced enough till its round number is greater than that tagged round number to announce. Therefore, the process will hold onto any messages it receives that have round numbers larger than its own until at some later round when its round number is one greater then it will announce to the user that it has received that message.

2 Problem 3 Analysis

English Explanation of Solution We have built upon our solution for Problem 1/2, this time adding functionality for both a high-level and low-level receive. The low-level recieve works as normal: it iterates over all processes, sends all messages contained within a message list, receives a validation response from P_i, and then sends a message to P_j to begin the next round of message passing. The high-level receive works by first iterating through the ranking order of each process; if the current process' rank is valid, it will then execute a low-level receive, and if not, it sends a look message to receive and validate the next proper process. (The last process in the rank indicates that it is time to move to the next round.) The high-level receive then waits to hear which process is next in line, then returns the value of that process.
The algorithm in the main program file works by iterating over all processes each taking their turn to send their respective messages. Assume a 2d array of messages exists where $msg[i][j]$ is the current message to be sent. In each simulated round, a high-level send/receive will allow P_i to send its message to P_j across every round.

Pseudocode: Complexity: The complexity of a single round where each process is able to send a message to every other process is O(n^2)

```
1: Round()
2:       for j = 0; j < n; j++
3:            if j == i:
4:                 LowLevelSend(messageList)
5:            else receive()
6:       wait for end of round message
7:
8: LowLevelSend(messageList)
9:       Send messageList[j] to all j
10:      wait for validation from all j
11:      Send Sender complete message to all
12:      if i == n send last sender message for all
13:
14: receive()
15:      Wait for message from Pi
16:      send Validation to pi
17:      wait until last sender message
```

Correctness: The above psuedocode is slightly simplified but overall the idea ensure a synchronous system on a round by round basis. In a single round each process takes a turn sending a message to each process while every other process waits to receive the message. Once they receive the message, the send validation to the sender. The sender cannot exit the sendMessage function until it has received validation from each other process. One it has, it exists and every process has received it's respective message. Every process takes a turn at sender in a given round. Therefore there is no way for a single sender to move forward until it has received validation that every other process got the sent message and all of the processes cannot move onto the next round until every process has sent and received validation. Therefore there is no possible way for any process to be a round ahead or behind of any other process. This proves a synchronous system as no process can proceed until every process has completed its work.

3 Problem 4 Analysis: 2 Phases Commit Protocol (2PC)

English Explanation of Solution 2PC is a special consensus algorithm that is famous in transaction problems, particularly in resources manager. The main goal is to ensure all transactions are either committed by the resources managers or aborted. The processes are divided into 1 coordinator and participants, and the protocol, unsurprisingly, consists of 2 phases:

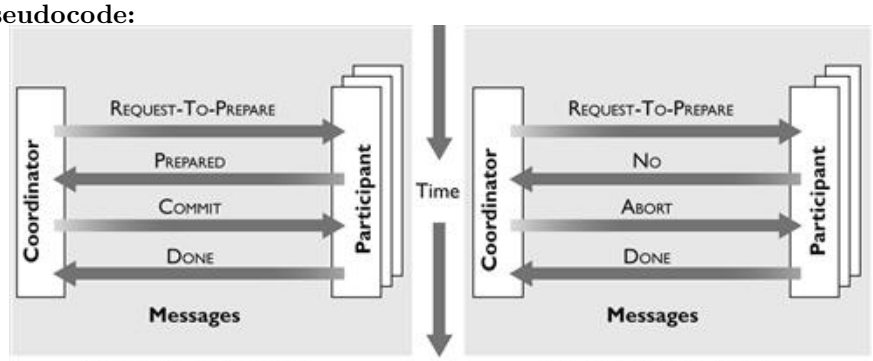
Phase 1: The coordinator sends out Request-to-prepare, then waits for responses. The participants who receive the request can either respond COMMIT,

ABORT, or unideally fail and never respond (This is where the synchronous system is required)

Phase 2: The coordinator receives responses and makes the decision based on the vote, then broadcast it. Participants acknowledge receipt of the commit.

Our algorithm is based on the solution from Problem 3, so the code structure is different from normal implementation. Nevertheless, the logic behind is the same: The 2 phases are considered rounds, and P_0 is always considered as the coordinator.

Pseudocode:



a. The transaction commits b. The transaction aborts

Horizontal arrows indicate messages between the coordinator and participant.
Time is moving down the page, so the first message in both cases is Request-To-Prepare.

From: Bernstein, P. A., & Newcomer, E. (2009). Principles of transaction processing (2nd ed., Ser. The morgan kaufmann series in data management systems). Morgan Kaufmann.

Complexity: The complexity of a single round/phase is the same as the solution of Problem 3, $O(n^2)$

Correctness: The proof of correctness can also be given from Problem 3. In addition, since the coordinator makes the decision to commit if and only if all participants respond COMMIT, this ensures that every process at least agrees on committing the transaction in Phase 2.