Comp 512 P2 Treasure Island Board Game Report

1 Paxos Sequence Architecture

We achieve total order of moves by using a paxos to create concensus on sequences of delivery orders. Since basic paxos algorithm only garentee the final values of each node to be consistent, but does't ensure confirmations are executed in the same order in each node. The only thing to handle in this approach is to ensure confirms (of sequence) are executed in the same order everywhere, this is handled by an additional step of algorithm we designed to give paxos such feature. Such aditional step also play a big role in group maintainance. Some implementation highlights are as below:

- A MessageListener thread to handle all kinds of incoming message(propose/promise/ acceptRequest/acceptACK/confirm etc.) and send corresponding response (promise/ acceptAck etc.)
- A proposeTracker thread to act as a proposer trying to suggest an sequence/order of moves to be delivered in every node.
- A thread in MessageListener class to turn gcl.readGCMessage to be non-blocking, which massively increase overall performance.
- We maintain a queue that tracks each of the confirms, we have to collect all confirmAcks from all processes to determine whether we deliver that confirmed sequence or discard it.
- Only confirms with BallotID larger then any confirm.BallotID you already received are added to the queue and broadcast confirmAck(true,BallotID), otherwise we broadcast confirmAck(False,BallotID).

1.1 Message Listener

In the messageListener thread, the only thing that behave different from the standard paxos algorithm is as follows:

```
Upon Receive a Confirm(id, value):

if no confirm received yet:
    add this confirm in the tempConfirmQueue
    maxballotID = id
    broadcast confirmack(accept)
if received confirm:
    if id > maxBallotID:
    maxBallotID = id
    add this confirm in the tempConfirmQueue
    broadcast confirmack(accept)
    if id < maxBallotID:
    broadcast confirmack(reject)
Upon Receive confirmack(id, confirmack):
    confirmackbuffer.get(id).add(confirmack)
if paxos have undeliveredMsg:
do processTempConfirm()
Upon processTempConfirm:
    peek the first confirm c in the tempConfirmqueue
    if no confirmack(reject) in confirmackbuffer.get(c.ballotID):
    paxos determine order add confirm.order
    tempConfirmqueue.pop()
    reset determinedorder, hasaccepted to false
```

1.2 Propose Tracker

Propose Tracker thread behaves the same as the standard paxos algorithm.

1.3 Shutdown

To ensure that all messages are delivered before shutting down. A process need to send broadcast a shutdown message to the GCL. And it cannot shutdown until it receives all shutdownAck messages every other processes. Since the GCL is FIFO now, a process receives a shutdown message from another process implies that it has received all messages from this process. By waiting for all shutdownAck messages from all process, a process can ensure that it has received all messages from the system.

2 Questions

2.1 How you ensure that moves made by various players simultaneously eventually makes it to all the game instances.

In our approach, once you call the paxos.broadcast API, the new move you suggested will be directly broadcast to the whole group by GCL and be stored in a messageBuffer in each processes. All process are going to greedily trying to propose a sequence through paxos to deliver any message store in the messageBuffer. Thus, other processes always deliver the value no matter what.

2.2 How you ensure that every process has a fair chance in pushing its move when the rate of moves generated increases (and is not overwhelmed by a process making most of the moves)

Because paxos.broadCast is blocking. A process can not broadcast another move until its previos move is delivered to the AL. Therefore, in any confirmed sequence/order going to be delivered, there will be at most one move from each of the player. A process can only push a large number of consecutive moves to other's AL only when no other process pusing move. Those consecutive moves will be wrap in descrete proposal (one in each). Thus the fairness of pushing ones move is stricly garenteed.

2.3 How you ensure that though your broadcast API only wait for a majority to accept the value, how the rest of the process still gets the value.

Since we are paxosing sequence/order of moves instead of decrete move. We choose to implement the paxos.Broadcast to be blocking until its previous message has been delivered everywhere. Thus returning from the API means the move you suggested must be already delivered to AL. Thus the question is not a problem at all.

2.4 How you handle various failure scenarios.

When some node fails after it receives the propose, our code will work the same as the paxos algorithm. As long as no majority nodes are failed, the failed node will disconnect and the sender of the propose still waits for the majority to confirm. Until the proposer collects the confirmAck message, it will wait for a time interval, if it does not receive the corresponding confirmAck, then it is regarded as a invalid node and thus

removed from the processes, with Numprocesses -= 1. This behaviour remains for all the "acceptor failures" (Failure 1, 2)– Once the non-proposer node fails, the proposer will remain unaware of it until the time it collects confirmAck.

If a proposer disconnects after sending proposes (Failure 3) or becomes the leader and then disconnects (Failure 4), since all processes greedily proposes as long as it has a order in msgbuffer, other proposer with a higherBallotID will take over and proceed.

Similarily for Failure 5, since it cannot send out the confirm message, this propose will be taken over by other proposes, the only difference is that the new propose will proceed with the value that has been accepted previously.