School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2022

Programming Assignment 2: Total Order Using Paxos

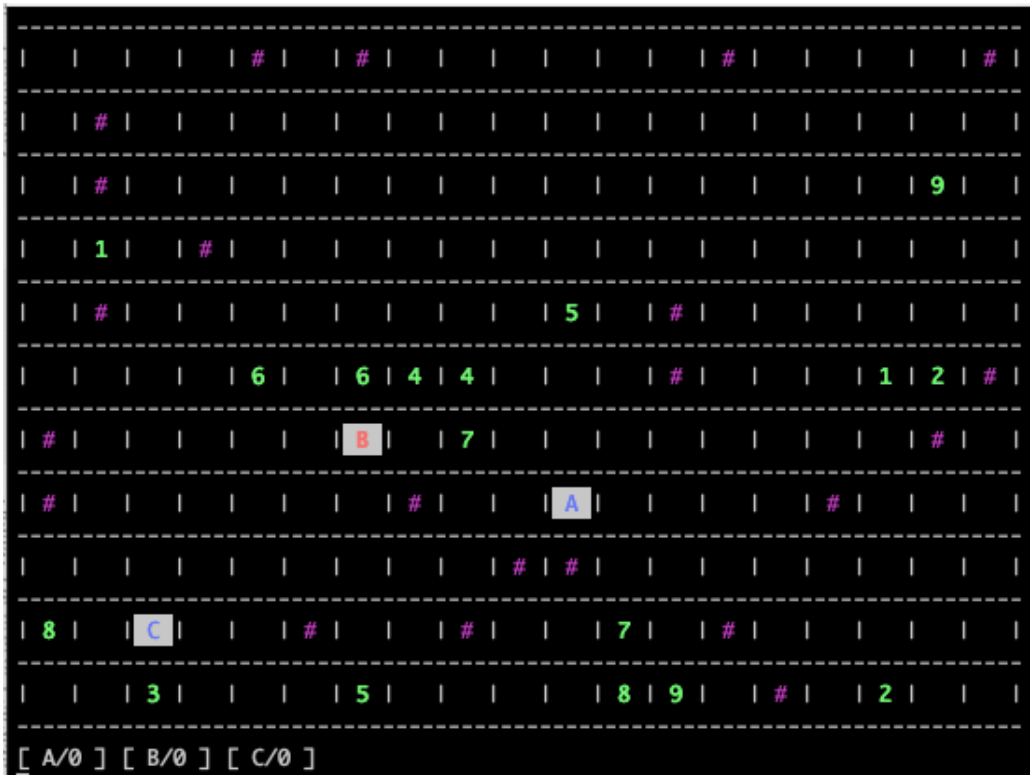
Due date: October 25 @ 6PM, Demo date October 26/27/28

Your task in this programming assignment is to develop a Paxos module that will ensure total order delivery of messages to the application layer. You are already given a group communication library (GCL) that you can use to send both multicast and point-to-point messages. You will build your Paxos layer over this, utilizing this GCL. Conceptually this will be similar to how we discussed various networking / group communication algorithms in class where we constructed a more refined layer over an existing primitive layer.

You will use the Paxos module that you built to implement total order communication between multiple players in a distributed multi-player game that is given to you. This programming assignment is an example of how we can streamline the events to a standalone application through a total order communication layer to implement replicated states.

The "Treasure Island"

The famous island of Paxos has fallen from its glory days and is now outrun by pirates. It also holds hoards of treasures from its glory days.



You and your teammates are pirates in this "Treasure Island" and have to roam around and claim the treasures (valued 1-9 points) before your teammates do. There could be 1-9

pirates in the island (A-J characters).

Each player has their own copy of the island. Therefore, it is important that whenever a player makes a move they broadcast it to all the players and then each player update their island's copy in the same order. Basically, we need the moves to be made in total order.

You can find how the game itself works by referring to the **UserGuide.pdf**.

There is no late turn-in for programming assignments.

Question 1: Implementation (65 Points)

Follow the instructions provided in **GettingStarted.pdf** in order to extract, compile and run the code.

```
comp512.jar
build_tiapp.sh
comp512st/
  |-paxos
  |---- Paxos.java
  |-tests
  |---- run_tiapp_auto.sh
  |---- TreasureIslandAppAuto.java
  |-tiapp
  |---- run_tiapp.sh
  |---- TreasureIslandApp.java
```

The `TreasureIslandApp.java` contains the logic to run the game (the fun, interactive version), and interact with the Paxos module. You should not ideally have to do much modifications to this code's core logic besides perhaps ensuring that any outstanding messages are processed before the application is shutdown (which it currently does not bother) and any additional debugging, etc., that you need.

`Paxos.java` contains the template of the code that you will work on. This class and its currently public members must be the ONLY public items in your code. You are free to add other classes, methods, etc., as long as their access scope is restricted to the `comp512st.paxos` package.

Currently this code does not do any Paxos ordering work and instead just pushes the messages directly to the GCL and retrieves the messages from GCL and sends it back to the application. You will have to modify it to ensure it provides total ordering using Paxos, is able to handle failure of individual processes, etc. For example, if currently two players' processes broadcast their moves near simultaneously, it is possible that different game instances will see the moves in different order. Your Paxos logic will have to ensure that all game instances see the messages in the same order.

Note:- It is extremely important that no moves pushed down by the application is "discarded" .I.e., because player 1's process did not become a leader for this turn (because say player 2's move got selected), does not mean that player 1's move is "discarded", it should then try to push the same move as the next turn (and so forth till it eventually succeeds). If your Paxos module drops such moves, it will impact the project grades significantly.

Below discuss the public APIs that you need to implement in **Paxos.java**.

Constructor

```
Paxos(String myProcess, String[] allGroupProcesses, Logger logger, FailCheck failCheck)
```

This is the Paxos constructor, as called by the application. Refer to GCL constructor definition in **GCL.pdf** to understand what `myProcess` and `allGroupProcesses` are. You should be able to leverage some of those concepts for your own work. `logger` could be used to add your own event logging for debugging purposes. `failcheck` object is to be invoked at various points in your Paxos logic to see if the process has been asked to fail at specific situations. See the discussion under **Enabling Fail Points** for details.

Broadcasting messages

```
void broadcastTOMsg(Object val)
```

This API is called by the application layer to broadcast its move - the argument is some serializable object. This is where your Paxos implementation kicks in and makes sure that the `val` is delivered (see next API) in the same order in every member process. In our case, the `val` is basically an object that encodes a player's move. Once again, make sure that no moves are "discarded" because one process did not become the leader for a turn and some other player's move was selected. It should try to push this move in as part of the next turn.

This call MUST block until Paxos has successfully had a majority accept this value. I.e., we **DO NOT** want the application pushing in the next move even before the previous move made by this process has been placed in the right order. However, this API must also **NOT** wait for EVERY process to accept (only a majority is required). On the other hand, make sure that every correct process eventually gets that move (in the right order).

Accepting messages

```
Object acceptTOMsg()
```

This API is called by the application layer to read a message (move) from the Paxos layer. If there are no messages, it **MUST** block. Also, remember that just because Paxos has a

message to deliver does not mean the application layer is ready to read it. You might have to internally buffer messages (possibly multiple) till a read is invoked (and pass them one by one with each read operation). The messages here are delivered in total order (basically the `val` objects send out as part of `broadcastTOMsg`).

This API can throw an `InterruptedException`. This could be useful in some situations. For example a thread is blocked waiting for the read (no messages), and a shutdown gets initiated. You can then interrupt the thread waiting on the read so that it can perform its own cleanup instead of being blocked.

Shutdown

`shutdownPaxos`

Called by the application layer to indicate that it is shutting down. You can do any cleanup as required.

Enabling Fail Points

While you do not have to worry about recovery, your Paxos implementation must handle failures of individual processes, as long as a majority of them are up and running.

Below is a list of failure points that you will have to install in your code, by calling the corresponding APIs on the `failCheck` object originally given by the application to your Paxos constructor.

- `failCheck.checkFailure(FailCheck.FailureType.RECEIVEPROPOSE);` to be invoked immediately when a process receives a propose message.
- `failCheck.checkFailure(FailCheck.FailureType.AFTERSENDVOTE);` to be invoked immediately AFTER a process sends out its vote for leader election.
- `failCheck.checkFailure(FailCheck.FailureType.AFTERSENDPROPOSE);` to be invoked immediately AFTER a process sends out its proposal to become a leader.
- `failCheck.checkFailure(FailCheck.FailureType.AFTERBECOMINGLEADER);` to be invoked immediately AFTER a process sees that it has been accepted by the majority as the leader.
- `failCheck.checkFailure(FailCheck.FailureType.AFTERVALUEACCEPT);` to be invoked immediately by the process once a majority has accepted it's proposed value.

How to test using these fail points is discussed in **GettingStarted.pdf**

Question 2: Performance Analysis (25 Points)

Using the testing framework given to you (see **GettingStarted.pdf**), run a performance analysis of your implementation. You are of course free to adapt it to your own needs.

The testing framework uses the automated playing program given in `TreasureIslandAppAuto.java` which is fairly similar to the code in `TreasureIslandApp.java`, except for the fact that it generates its own moves instead of reading it from the standard input where the user might type in their moves.

Try to understand the impact of the following factors on your performance.

- How does increasing the number of players impact the rate at which their moves are being accepted? - You will have to work by varying the number of players and interval between moves to see how this pans out.
- Does all of your processes have the same rate at which their moves are being accepted (especially when the intervals become shorter)? - try with more number of players and shorter time intervals. Is some process consistently performing better than the others? Can you figure out what aspect of your algorithm might be contributing to it?
- What is the maximum number of moves that your system can process per time unit when the interval approaches 0? - does it even has to reach 0 before this rate flattens out? Does this change as you increase the number of players?

For this section, prepare a short analysis report (around 2 pages) containing a description of the testings you performed, figures, analysis, etc.

Question 3: Project Report (10 Points)

This is separate from your performance analysis report. Write a short (1.5 - 2 pages) report detailing your architecture and design. You report should also indicate (at a high-level) as to:

- How you ensure that moves made by various players simultaneously eventually makes it to all the game instances.
- 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).
- 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.
- how you handle various failure scenarios (specifically the ones discussed in your project description).

Your report should also contain a small paragraph at the end as to the contribution of each of the group members. If we notice a patter of certain members not contributing consistently, they may receive a lower grade letter at the end of the course.

The report is due by **6 PM on October 25th** to give the TA enough time to read before the demos.

Demos

To grade your implementation we will use online demonstrations with the TA where you show your running system. Slides are optional, as long as you have some effective way of discussing your architecture with the TA (could be the diagrams in your report).

The demonstrations of all groups will take place over three days (October 26/27/28). A sign-up sheet will be put in place so that you can reserve a time-slot. Show up early and have your system running. You may use the same Unix host for all your players if necessary. Expect questions! Ensure the person who is doing the demo is efficient with Unix command prompt. Practice a bit on your own before showing up for the demo. Questions can be directed at anyone and not just the person who is executing the commands. **ALL team members must be present for the demo. Absentees will not receive the grade.**

Please remember that demo is not the time to debug your code. If it did not work for some scenario does not mean you get to rerun it a second time and show it is working the second time - the fact that it did not work the first time means there is some bug in your code that shows up itself every now and then. The concurrent nature of distributed systems can lead to buggy code that works some times and fails at other times.

All code is due on MyCourses by the time of your demo.

Restrictions

All of your inter-process communication must be through the GCL module given to you, no other communication paradigms (RMI, TCP sockets, etc., must be employed). Not conforming to this will significantly impact your grades.