School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2025

Programming Assignment 2: Total Order Using Paxos

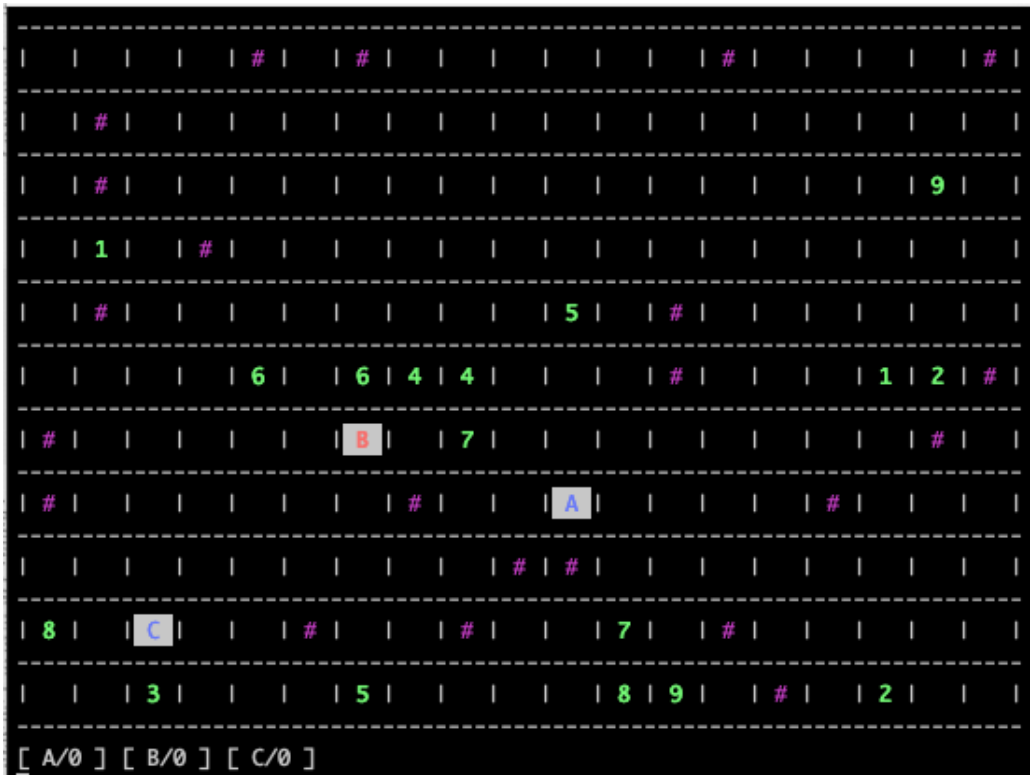
Due date: Report is due November 4, Demos on the following days

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 multicast 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**.

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 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. That is, if currently two players' processes multicast their moves nearly simultaneously, it is possible that different game instances will see the moves in different order.

The task of this assignment is that you implement a total order for all moves using Paxos, and that the game is able to handle failure of processes. You should implement an update anywhere approach where each process is proposer for the moves coming from its own application. You do NOT have to consider recovery of processes. Your Paxos logic will have to ensure that all game instances see the messages in the same order, execute the moves in the

same order, and thus see the game progress in the same way. In particular, when a player wants to move, the player's process should initiate a Paxos instance with the aim that this move is the next in the total order of moves. If several players attempt to move at the same time, only one can succeed to be the next move in the global order. The other players will have to attempt to make their moves to be the next moves in the order by initiating a new Paxos instance. That is, a game play will be a sequence of Paxos instances. Each instance represents one successful execution of the Paxos algorithm agreeing on one value that represents one move in the game play. One successful execution of the Paxo algorithm is also often referred to as a **turn** in the following.

How to move from one turn (Paxos instance) to the next has been shortly discussed in class. For instance, when proposing to become the leader, the proposer explicitly states the position in the total order this Paxos round aims at. At the beginning everybody aims at position 1. For those whose moves do not succeed for this position 1, a node can initiate a new Paxos instance by proposing to become leader for position 2, etc.

Note:- It is extremely important that no moves pushed down by the application are "discarded". For instance, if both player 1 and player 2 submit move requests at the same time and both processes try to become proposers to have their moves agreed on by everyone, only one can succeed in this instance. The other player's process should then push its move for the next instance (and so forth until it eventually succeeds). If your Paxos module drops such moves, it will impact the project grades significantly.

Below is 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

You will be using an existing group communication library to exchange messages among processes. The terminology used in this library is slightly different to what was discussed in class. As outlined in document GCL.pdf, our implementation has the concept of only a single group. Within this group, a group member can send a *point-to-point message* to one other group member, it can send a message to a *subgroup* of processes and it can send a message to *all* group members. We distinguish the latter two by using *multicast* if we send

to a subgroup of processes and *broadcast* if we send to all processes. For this assignment, you will mainly use the point-to-point and broadcast functionality.

Based on this, the Paxos-based total order algorithm has to provide

`void broadcastTOMsg(Object val)`

This API is called by the application layer to broadcast its move to all processes in the group - 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) until 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 sent 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 (promise or refuse) 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 its 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.

Provide an analysis of the following:

- How do the number of players and the interval between moves impact the time until a move is accepted and the rate at which moves are accepted (the number of moves the system processes per time unit). - You will have to work by varying the number of players and interval between moves to see how this pans out.
- Do 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 containing a description of the testings you performed, figures, analysis, etc. For each performance figure that you show, the text should have a description of what can be observed in the figure and a discussion why this happens.

The report should contain max. 2 pages of text (Times New Roman, 11pt, single-column). Figures can take additional space.

Question 3: Project Report and Demonstration (10 Points)

Project Description (5 Points)

This is separate from your performance analysis report. Write a report detailing your architecture and design. The report should contain max. 2 pages of text (Times New Roman, 11pt, single-column). Again, figures can take additional space.

Your report should also indicate (at a high-level) as to:

- How you ensure that moves made by various players simultaneously eventually make 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 although your broadcast API only waits for a majority to accept the value, how the rest of the processes still gets the value.
- How you handle various failure scenarios (specifically the ones discussed in your project description).

Collaboration As last section of your report, provide a description of the contributions of each of the group members. That is, for each of the tasks, indicate how much and what each of the team members has contributed. Furthermore, describe your collaboration efforts. Each team member is expected to contribute significantly to the project work. If we notice a pattern of certain members not contributing consistently, they may receive a lower grade letter at the end of the course.

Demos (5 Points)

To grade your implementation (i.e., the 65 points of the first section) we will use live demonstrations with the TA where you show your running system.

Furthermore, the way you present and demonstrate your system will be evaluated by a further 5 Points. We recommend a very short presentation that highlights your system and

implementation strategy. This can be shown on a few slides, but also simply a well-thought narrative where some aspects are illustrated by pointing to architecture figures in the report.

The demonstrations of all groups will take place over two or three days after the deadline. 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. The TA will ask you to perform certain tasks that show the working of the system. Expect also questions about your architecture, code design, implementation details, etc. 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 the 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.

Copyright