School of Computer Science, McGill University
COMP-512 Distributed Systems

TreasureIsland : Getting Started

The following is a description of the steps to run the template programming assignment code. You should get started as soon as possible, as there may be issues to work out before you can start the program implementation. This code is setup for compiling and executing in the SOCS Unix systems (specifically `lab2-XX` servers). Trying to build/execute them elsewhere is not guaranteed to work.

1 TreasureIsland Components

You are given a jar file, `comp512.jar` that consists of the group communication (GCL) functionality and the actual TreasureIsland game itself. You will be using this component as-is. This is contained within `comp512p2.tar.gz`.

In addition, this package also contains templates for the code to both start your work and test your work. Start by untaring this file, you will then get the following directory structure.

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

Compiling

Now edit the `build_tiapp.sh` script to change the `BASEDIR` variable to point to where ever you have extracted and placed these contents (i.e., their parent directory).

You are now ready to build the source code by executing

```
$ ./build_tiapp.sh
```

This should build any class files needed, etc.

Playing

Now edit the `run_tiapp.sh`.

Once again, update the `BASEDIR` variable as above.

Edit `group` to make it your group number instead of `XX` - (`03` if you are group 3, `30` if you are group 30, etc.).

The next step is important, it determines how many players are going to play now. The default script is configured for 3 players. But you can configure it from 1 player all the way through 9 players. Below, we assume that we are only playing with two players.

Decide on the name of the server and port that each of the two players will be using. Update this information on `process1` and `process2`. For simplicity, we recommend that you leave out the ports as-is and just update the server names that you are planning to use. Make sure that any unused process variables, `process3`, etc., are commented in the script. (The script automatically figures out the number of players based on the number of these variables defined.).

We are finally read to play!!

Decide who is going to be player 1, player 2, etc.

player 1 should login to the Unix server that was given for `process1`, go to the folder containing `run_tiapp.sh` and execute the following.

```
$ ./run_tiapp.sh 1
```

Similarly, player 2 will also login to their Unix server (for `process2`) and execute

```
$ ./run_tiapp.sh 2
```

The same philosophy is applied to all the other players as needed when more players are present.

Get the game going !, for more information on how to play, see **UserGuide.pdf**.

Note:- If you want to change the island's map, just edit the script and change the value of `gameid` to something else. Make sure that all the players have the same `gameid`.

Stress Testing

While it is fun to play the game manually, it is not going to be "fast" enough for some of your testing scenarios (like what happens when two processes try to send their moves at the same time, etc.). The purpose of the tests module is to provide some help with this, to automatically generate the moves and broadcast them at specific intervals.

For this, edit the `run_tiapp_auto.sh`.

As before, update the `BASEDIR`.

Edit `group` to make it your group number instead of `XX` - (`03` if you are group 3, `30` if you are group 30, etc.).

Update the `autotesthost` to point to the Unix server that you will be using for the tester (all the tester players will be started from the same host, but with different port numbers).

The next step is important, it determines how many players are to be used by the tester. The default script is configured for 3 players. But you can configure it from 1 player all the way through 9 players. Below, we assume that we are only playing with two players.

Decide on the ports that each of the players will be using. Update this information on `process1` and `process2`. For simplicity, we recommend that you leave out the ports as-is unless you run into some conflicts. Make sure that any unused process variables, `process3`, etc., are commented in the script. (The script automatically figures out the number of players based on the number of these variables defined.).

Update the `maxmoves` variable to indicate how many moves must be made by each player.
Update the `interval` variable to indicate how much time (in milliseconds) the automatic move generator should pause between generating its moves.
Update the `randseed` variable to have the automatic move generator create a different set of move pattern. (Any string/numeric values can be used).

If you are testing failures as part of your test run, set corresponding variables. For example, if you set `failmode_2` to `AFTERBECOMINGLEADER` then the second player's process is set to fail in **one of the** instances (randomly decided by the auto move generator) of it becoming the leader. (Also see **USerGuide.pdf** and **p2.pdf** on this).

Note:- In most cases, a player configured to fail will do so before they reach `maxmoves`. In some cases it may not happen (as failure events are randomly generated), in such cases, either increase the value for `maxmoves` or change the value for `randseed`.

It is recommended that when you start testing, do so without any failures enabled. Once you have the basic consensus working in the absence of failures, you can start failing processes to see how your Paxos module is handling them.

The Catch: The problem with simulating failures is that in an asynchronous, concurrent, environment, the order of events are non-deterministic. Especially, given that GCL sends are non-blocking, it is possible that a vote request may not have been actually send out by a process' GCL layer before an `AFTERSENDVOTE` failure event crashes it. We therefore, recommend that you start with the manual, interactive game mode described in the previous section as your primary mode of testing failures (see **UserGuide.pdf** on invoking those failures). The other option is to use a larger interval in the automatic tester (that simulates the delays between moves closer to a real human player interaction). This is because if the GCL has less messages to push out, there is a higher chance that such messages are sent out before the process is crashed.

If you want to change the island's map, just edit the script and change the value of `gameid` to something else.

You can execute the script as indicated below

```
$ ./run_tiapp_auto.sh
```

Be mindful to start with fairly large values for `interval` in the beginning. This is because if there is a bug in your implementation, it could produce a lot more messages than that would be fairly expected in a consensus environment (say for example, you are starting to work on the next move from a process even before you have pushed its previous move, etc.). And that could overwhelm the GCL layer and your java process might run out of memory (as the GCL is non-blocking and internally queues the messages it will not stop Paxos from producing a lot of messages on its own.)

If the `interval` is set to 300ms and it took 100ms for the previous move to be processed, the move generator will wait another 200ms before producing the next move. However, if it took 400ms for the previous move to be processed, it is not going to wait anymore and will process the next move immediately (in this case the time between generating the moves will be 400ms - approximately).

Be careful if you happen to kill this process explicitly (or gets terminated as a result of you getting disconnected, etc.), it might leave the java program running in the background and as a result you might not be able to reuse the same port. In such circumstances, check using the `ps` command for any left over java programs in your account and explicitly terminate them.

2 The Log Files

The modules and scripts generate a set of logs which help you understand what is going on, and in some cases have a quick peek at whether the outcome is correct or not. All the logs are generated in the "current directory" and are overwritten for every run.

The Islands's Log. The first of these logs is generated by the TreasureIsland itself, to indicate the order in which it received the moves. As such, it is an important component in verifying if your Paxos module is working correctly. In a correct implementation, all the player's logs would be identical (except the first, header line). The objective is that all players get the same ordering of moves (facilitated by your Paxos module). However, remember that just because the logs are identical does not necessarily mean your implementation is correct. It could be that specific situations that can cause your Paxos module to behave incorrectly did not occur yet. Therefore it is important to use the auto-testing module to stress test your implementation with different configurations. These logs have the name of the form *gameid-playernum*. So `game-90-99-1.log` is the island log file for the game id `game-90-99` by player `1` and it should be identical in the order of moves to those from other players, `game-90-00-2`, etc.

Start/End state of Auto Tester Runs. This log is generated *ONLY* when you use the auto tester. Since you can make thousands of moves when auto testing, it is not feasible to manually verify the map of the island after each individual move. So this log provides a quick view into what the island map looked like at the initialization and what it looked like at the very end. Ideally every auto player's maps should look the same. However, you must still look into the island's log to double confirm that they are still agreeing between different players. But if this log already shows that the end maps are not identical, then you already know the island's log will not be identical. The log file has the name format *gameid-playernum-display.log*. So `game-90-99-3-display.log` belongs to the game id `game-90-99` of player `3`.

The Events Log. This a detailed log of all the events happening in that process (as recorded by the **logger** object) and will be useful for debugging. Right now, among other things it will show when the GCL received a request, when it was sent, received, delivered, etc. Make sure that you are also using the **same** logger in your Paxos module implementation and call the logger to report events that you need to debug / understand what is happening in your code. Please refer to the online Java documentation for `java.util.logging.Logger` if you are not familiar with the logging module. This log file is of the name form *gameid-process-playernum-processinfo-.log*. For example, `game-90-99-lab2-12.40390-3-processinfo-.log` belongs to the process which was playing the game with game id `game-90-99`, with a process identifer `lab2-12:40390` - i.e., executing on the Unix server `lab2-12` and using the port `40390`. It was also representing the player number `3`.

Some of the important log messages are briefly described below for a quick reference.

```
.. broadcastMsg FINE: Broadcast message request received. Translating to point-to-point messages ..
```

This means GCL has received a broadcast message from the application layer.

```
.. sendMsg FINE: Request received to send message to ..
```

A point to point invocation (either directly from AL or within GCL)

```
..comp512.gcl.GCL$GCLOutBoxDespatcher run FINE: lab2-12:40190: Sending message to. ..
```

The message is actually being sent by the GCL to the other process.

```
.. Message sent. message: ...
```

The message was actually sent by the GCL to the other process. (You may need to enable *FINER* level in logging to see these messages.

```
.. Received a message from. message: ..
```

The GCL layer of the process has received a message.

```
.. readGCMessage INFO: Delivering message ...
```

The AL has requested a read and the GCL layer of the process is now handing over a message.