Comp 533 - Assignment 6: Consensus Mechanisms

Date Assigned: April 24, 2017

Completion Date: Tue May 3, 2017

In the second part, you will use yet another way to implement our example application – the use of a system-provided replicated object. It has the advantage of being matched to the application – replicated simulations. It also provides multiple ways to perform the underlying communication, offering different degrees of safety and progress. You will configure the abstraction to experiment with these tradeoffs. You already have a way to choose between NIO, RMI, and GIPC. You will extend it to now choose five additional methods of replicating commands, all of which use different implementations of a common consensus mechanism. The assignment assumes you use integers to indicate the replication method.

In the third part, you will use the simple example presented in class to understand various aspects of Paxos. This part will require appropriate breakpoints and viewing the associated PowerPoint presentation.

Both parts require you to understand the implementation of a configurable example, which is in package: examples.gipc.consensus.paxos. In the first part, you will run it multiple times with different configurations.

If everything works as hoped, Tuesday class should be enough to do parts 1 and 2; and Thursday class should be sufficient to part 3.

Part 1: Understanding Steps n Different Consensus Mechanisms

To run the example, you need to run the session server (ASessionServerLauncher), the two proposers (PaxosProposer1Launcher and PaxosProposer3Launcher), and the acceptor (PaxosAcceptor2Launcher). The session server is used only to connect the three members, it does not communicate messages.

The proposers and acceptors are subclasses of APaxosMemberLauncher, which in turn has superclasses AnExampleProposerLauncher, and AnExampleMemberLauncher, which you should look at before starting.

APaxosMemberLauncher has the following method:

```
	protected void customizeConsensusMechanisms() {
//		simulateNonAtomicAsynchronous();
```

```
//            simulateNonAtomicSynchronous();
//            simulateCentralizedAsynchronous();
//            simulateCentralizedSynchronous();
//            simulateBasicPaxos();
              simulateSequentialPaxos();

      }
```

The commented lines show how the underlying consensus mechanism can be changed.

It also has the following code to control what values are proposed:

```
protected int meaning1() {
      return super.meaning1();
}

protected int meaning2() {
      return super.meaning2();
}
```

Choose some pair of proposed meaning values that you think will be unique and run the four processes (session server first, and then the acceptors and proposers) six times, once for each line in the customization. Study the output each time and take a screen shot of the console of proposer 1.

Part 2: Using the Abstraction and Performance Studies

You will add new methods to make the simulations consistent. These will involve running the session server of part 1. It will also involve running your simulation processes. You can also run your server to support your previous consistency mechanisms, which will not change.

GIPC Abstraction

The example code explains how to use a GIPC consensus mechanism, which involves the following:

1. Creating a consensus mechanism with a specified object name, member name, session name, session server host, and port id. The meaning mechanism is created with these parameters.
2. Creating a consensus mechanism with a specified session, object, and member name, which uses the session host name and port id of a consensus mechanism previously associated with the same session name. The greeting mechanism is created with these parameters.
3. Control which factory is used to create the consensus mechanism. The integer mechanisms is created using the sequential sequential-paxos factory and the greeting mechanism is created using the asynchronous-consensus factory. In your implementation, you want to use the sequential paxos factory for the commands and the asynchronous factory for the configuration. You can use the same factory for both, in which case you need to customize the configuration sequential mechanism appropriately.

4. Connect a mechanism to a listener. Both mechanisms are associated with listeners that simply print the new values. In response to a consensus on a value proposed to a consensus mechanism, its listeners are called on all replicas of the consensus mechanism.
5. Configure the mechanism. The Part 1 code shows how this is done.
6. Make a proposal and wait for it to reach consensus before proposing a value to avoid conflicts among pending proposals. Do not wait for consensus on the previous pending requests as that may create problems when you switch consensus mechanisms. Do not use a loop of the kind I have to retry, just propose one value and then wait.

To communicate commands, your clients will now instantiate local consensus mechanisms associated with the same session host, session name, and session server. The interface ExampleMember in examples.gipc.consensus encapsulates these data. Create a version of it for your example, choosing appropriate names for your two replicated objects. *I recommend you do not inherit from my code* and instead copy relevant aspects and implement the interfaces mentioned below.

Each of your clients will have a separate member id (which must be a short) and port id. The interface Member1, Member2, and Member 3 encapsulate the ids and port numbers. You can use them directly in your code, but make sure the port numbers do not conflict with those used in your previous mechanisms.

Associate a listener with the mechanism to receive local and remote commands. Propose a new command to the mechanism to communicate the command and wait for consensus on that proposal before issuing the next command to avoid creating multiple pending commands, as mentioned above.

Dynamic Mechanism Change

Provide a way to dynamically change the configuration to each of the configurations illustrated in the example except basic Paxos (which cannot overwrite values). Create and use a different consensus mechanism to broadcast the configuration change to others. This mechanism will allow a member to propose a new number for the command consensus and the listener for this mechanism will execute the appropriate customization method associated with the number. All five new methods will use the same consensus mechanism, the one created by ASequentialPaxosConsensusMechanismFactory, to communicate the commands. Thus, all five methods will involve proposing values to the mechanism and listening for consensus.

The sequential access configuration mode calls setSequentialAccess(true); Make sure all of the other methods set it to false.

Experiments

Compare the times required to run the 3-client 500-command experiment in the five configurations with each other and the times you observed in Assignment 3. You should create a single run to generate all parts. You are free to use the Java serializer or your own serializer.

Make sure you pause your program after each configuration change so that the accepting/listening nodes catch up with the proposing nodes – otherwise you will get buffer overflow problems.

In the centralized case, the system makes the lowest numbered node the central proposer. Choose some other process for issuing commands – such as 3 so that both processes are involved in the central case – the issuer and another process. Create screenshots of the results on the node that issues commands – such as 3.

To convince me and yourself that the experiments are running correctly, run them again, but this time in a two-command experiment, with tracing on:

port.trace.consensus.ConsensusTraceUtility.setTracing();

In the asynchronous case, you should not have any accepts, or prepares and in the synchronous case, no prepares. In the centralized cases, you should have remote proposes on a different computer, and in Paxos, you should have prepares. In all cases, you should have consensus at all sites.

Part 3: Understanding Paxos Steps in Some Depth

Run cases 4 (or 5) and 7 in the class material on Paxos implementation. I recommend you run 4 instead of 5, unless you have already run 5.

The two proposes and the acceptor are in APaxosScenariosMemberLauncher in package examples.gipc.consensus.paxos.scenarios.

In both cases, capture the console output at node 3 as text. For case 7, execute at least one re-proposal of 1 and 3 each. Thus, both 1 and 3 should make at least two proposals each.

To run these cases, you need to set breakpoints in the methods named in the class material. In general, to find some text in the project, in Eclipse you can do Search→File and ask Eclipse to search for the string in the Project or the complete workspace.

The startAcceptPhase() and startPreperePhase() methods are in consensus.paxos. APaxosConsensusMechanism.

The other methods are in examples.gipc.consensus.paxos.scenarios; APaxosMultiCaster.

For case 7 only, set in class the method customizeConsensusMechanisms() in APaxosScenariosMemberLauncher set the inherited variable, overrideRetry to false;

```
overrideRetry = false;
```

For both cases, annotate the output to show:

1. Each prepare request made at 3.
2. Each prepare request received at 3.
3. The point at which majority prepared responses are received at node 3.
4. The accept requests received at node 3.
5. The point at which majority accept responses are received at node 3.

Submission

No video this time – submit the console output instead as hard copies. As always submit the code on Sakai.