Paxos programming practical: instructions for coding
Richard Connor
February 2025

I have provided Java code which runs a number of *PaxosProcesse*s in parallel. The package as provided should execute as it is, although it doesn't do anything very useful other than sending some useless messages around the initial processes. Before proceeding make sure you understand all the necessary details of these classes. I have *deliberately* not put any comments in the code!

The Java code for the eight classes is available on studres at Coursework/paxos.

The core process model in this implementation is a little like that of the FLP proof. There is a single message pool. Each *PaxosProcess* defines only a single step, where a step comprises: fetching a message from the pool (which may be null if there isn't one); performing some appropriate actions, and finally ending with sending one or more messages to other processes via the *MessagePool*. That is all built into the template.

The practical requires editing the *Proposer*, *Acceptor* and *Learner* classes so that they implement the Paxos protocol. You may also choose to edit other classes, or to add more classes, but you don't need to.

The classes provided are as follows:

Main : contains a *main* method that creates lists of *Proposers*, *Acceptors* and *Learners*, and sends a list containing them all to be run in parallel threads. You might want to make changes to this, particularly:

1. You can change the numbers of proposers, acceptors and learners as you wish for different demonstrations of Paxos message interleaving.
2. The *maxProcessSteps* puts a pragmatic bound on the number of steps executed before the whole program stops; the larger this number, the longer it will run for, but you may need to increase it for a more complex interaction scenario.
3. The different *proposals* are not very interesting….!
4. Similarly, the *ballotIds* built in are quite uninteresting, but are unique.

Notice that the order of acceptors used to initialise the proposers and learners is randomised, you might want to do something different (or you might not…)

ParallelRunner: this is tricky Java that you may not have encountered; you don't need to change anything in here, or understand how it works in detail. *runProcessesInParallel* runs the list of *PaxosProcesses* provided in parallel, i.e. one thread per process, with random delays on each process step. The min and max delay values (in milliseconds) are given in the parameters. The idea is to simulate wide-area delays; if there were no delays the set of processes would probably do the same thing each time, which wouldn't be very interesting.

PaxosProcess: an abstract class from which Proposer, Acceptor and Learner all inherit. The *nextTask* method needs to be written in each, and some classes will need some added state.

The *nextTask* method returns a Boolean; if it's true, then the process is essentially removed from the scheduler (it isn't actually, but it won't take up any significant processor time.) This is not really necessary but will shorten the wait time for the program to end!

MessagePool: a single parallel-safe message pool used to send messages among the Paxos processes via *send* and *receive* methods. I don't think you'll want to change this, except maybe to add unreliability to the message delivery.

Message: a utility message type, I don't expect you will need to change this. There is a fixed set of message types encoded in an enum. Note there are three different constructors, depending on how much info the message needs to carry.

Proposer, *Acceptor*, *Learner*: over to you! On each message step, all of these Paxos process types should (a) attempt to receive a message destined for them; if there is one, handle it appropriately, and then do something else including other message sends. This framework template is already coded in, you shouldn't change that template. If the process has logically finished, it should return true, otherwise false. One Java detail: if you haven't used a Java *switch* statement before, note that every case needs to be terminated with an ugly break instruction, otherwise the control will flow on to the next case. Bad language design, but we're stuck with it!

The core practical task

1. Add state and logic to the three PaxosProcess types to implement the core Paxos protocol, and demonstrate its operation/interleaving via calls to the log functions. Each proposer making a single proposal is sufficient for this, although of course it is possible for no consensus to be reached.
2. You might like to add failure modes to the processes and/or messages, to test its resilience, by e.g. randomly pausing threads and/or stopping or delaying messages.
3. You might like to refine the Proposers so that, if it seems that a consistent learned state hasn't been reached, they will make another proposal. Probably only if you're particularly interested, or are aiming for a very high grade!

Appendix: possible output

Just an example, this is one output that my example system produces. It will do something different each time it is executed, including possibly failing to reach a consensus.

threads set up
Acceptor 2: message <prepare> received from process 3
Acceptor 2: ballot id 1000 accepted
Acceptor 1: message <learn> received from process 6
Proposer 3: message <acknowledge> received from process 2
Acceptor 2: message <prepare> received from process 4
Acceptor 2: ballot id 1001 accepted

Proposer 4: message <acknowledge> received from process 2
Acceptor 0: message <prepare> received from process 5
Acceptor 0: ballot id 1002 accepted
Acceptor 1: message <learn> received from process 7
Proposer 5: message <acknowledge> received from process 0
Acceptor 0: message <learn> received from process 6
Acceptor 2: message <learn> received from process 7
Acceptor 1: message <prepare> received from process 4
Acceptor 1: ballot id 1001 accepted
Acceptor 2: message <prepare> received from process 5
Acceptor 2: ballot id 1002 accepted
Acceptor 1: message <learn> received from process 7
Proposer 4: message <acknowledge> received from process 1
proposer 4 received ballot majority
Acceptor 0: message <learn> received from process 7
Proposer 5: message <acknowledge> received from process 2
proposer 5 received ballot majority
Acceptor 1: message <prepare> received from process 3
Acceptor 1: ballot id 1000 declined
Acceptor 2: message <learn> received from process 7
Acceptor 1: message <learn> received from process 6
Acceptor 0: message <learn> received from process 6
Acceptor 2: message <accept> received from process 4
Acceptor 1: message <accept> received from process 4
Acceptor 2: message <accept> received from process 5
Acceptor 0: message <learn> received from process 7
Acceptor 2: message <learn> received from process 7
Acceptor 0: message <accept> received from process 4
Acceptor 1: message <learn> received from process 6
Learner 7: message <value> received from process 2
Learner 7: learned 1002:third
Acceptor 2: message <learn> received from process 6
Acceptor 0: message <accept> received from process 5
Learner 6: message <value> received from process 1
Learner 6: learned 1001:second
Acceptor 1: message <accept> received from process 5
Acceptor 2: message <learn> received from process 6
Acceptor 0: message <learn> received from process 7
Learner 6: message <value> received from process 2
Learner 6: learned 1002:third
Acceptor 0: message <learn> received from process 6
Learner 7: message <value> received from process 0
Learner 7: learned 1002:third
Learner 7: *** Consensus! value is <third> ***
Acceptor 1: message <learn> received from process 7
Learner 6: message <value> received from process 0
Learner 6: learned 1002:third

Learner 6: *** Consensus! value is <third> ***
Acceptor 0: message <prepare> received from process 4
Acceptor 0: ballot id 1001 declined
Acceptor 0: message <prepare> received from process 3
Acceptor 0: ballot id 1000 declined
executed 200 steps, processes terminating
done