1. Introduction

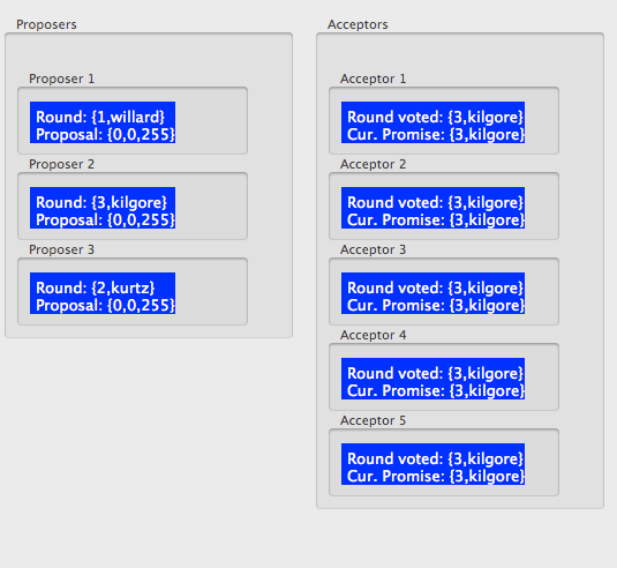
This seminar is about implementing the Paxy algorithm for how to gain consensus amongst processes in a distributed system. The algorithm is composed of proposers sending out ballots and trying to gain a majority consensus from the available acceptors who can themselves send back messages saying whether they can vote in a given round based on past promises and if they can in fact vote, if they'll vote for the proposal sent by a given proposer. We'll simulate the algorithm at first by filling in the code provided to us, and then run some experiments to see how delay changes, message dropping, changes in numbers of acceptors/providers, fault tolerance, etc. We will additionally extended the system by decoupling the acceptors and providers and allowing them to communicate from different Erlang instances locally with the intention of to simulating distributed systems and then finally we implement a final improvement on the implementation of the proposer to make the ending of a ballot which has already been negatively decided more efficient.

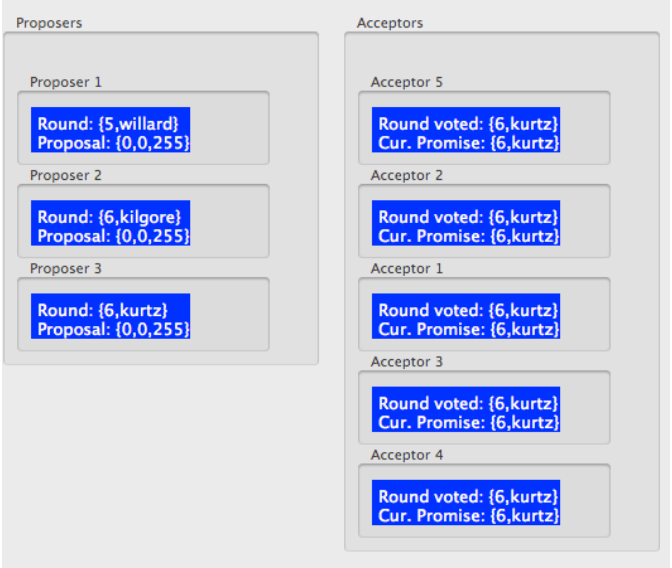
2 Work done

Our source code files are included as a zip file & include the original files with the code filed in and the following:
acceptor.erl.fault-tolerance (the original completed code and Part 3),
acceptor-w-delay-no-sorry-dropped.erl (code from experiments 1,2,3),
paxy-6props-3accepts.erl & paxy-6props-9accepts.erl (for experiment 4),
paxyacceptor.erl, paxyproposer.erl, and proposer.erl-distributed (for experiment 5)
proposer.erl-preemptive-abort (for Part 4)

3 Experiments

Initially, the system as is, with no delay, gains consensus very fast so after experiment 1 we retain a delay of 2000ms to slow the system down some for observance. Because each experiment has this limitation, the overall comparisons among experiments remains valid.
In Experiment 1, we add a delay of 2000ms after both the prepare and accept messages are received by the acceptors, and although they take longer to reach a consensus, they still do after 3 rounds as opposed to one round before. Removing the delay from the accept message similarly terminates, this time with 4 rounds but still similar. We similarly drop the delay from after the prepare message is received, and double the amount of time delay after the accept, and consensus is still reached, this time after a longer period and 6 rounds.





In Experiment 2, we stop returning sorry messages from the acceptor for both the prepare and accept messages it receives. Effectively the acceptors don't tell the proposers when they are unable to vote on something because the Round in the given proposal, for either the prepare or accept messages, is lower than the Round the acceptor has currently already promised. This actually won't change the behavior of the system and agreement will still be reached, because although acceptors will no longer send sorry messages, its still possible for them to all vote for a given Round/proposal and for a given Proposer to eventually get a majority of votes consensus. This modification will make the process of reaching consensus take longer because as the acceptor doesn't send a sorry message, the proposer will continually wait until the time out is reached either in his collect or vote methods.

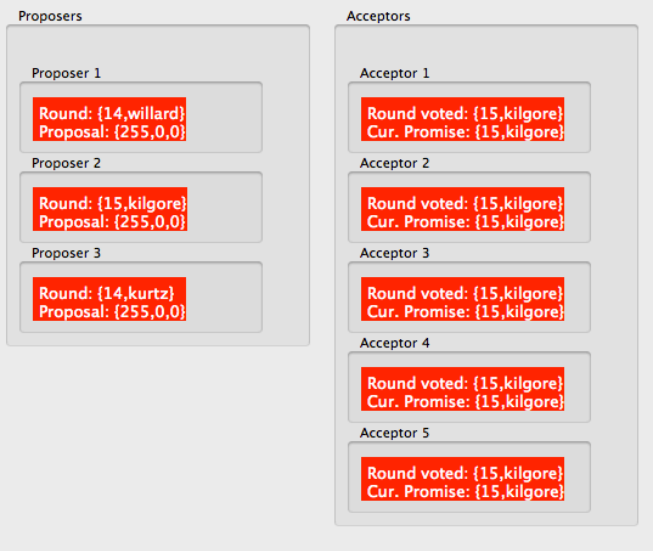


fig: acceptors no longer sending sorry messages results in longer time to reach consensus

In Experiment 3, we randomly allow acceptors to drop sending vote messages so that 1 in 10 times they don't vote when sent an accept message from a Proposer. This modification makes it so that the process for consensus takes occasionally slightly longer overall because on the occasion that an acceptor receives a round number higher than their last promise and last vote and hence, would normally vote for the proposed Round, but don't because they drop the message, the Proposer waiting for the vote will timeout and hope for better luck the next round. This will result in the algorithm reporting conflicting algorithms because when the message is dropped, the acceptor will keep his current state when in actuality he should have voted for the proposal.

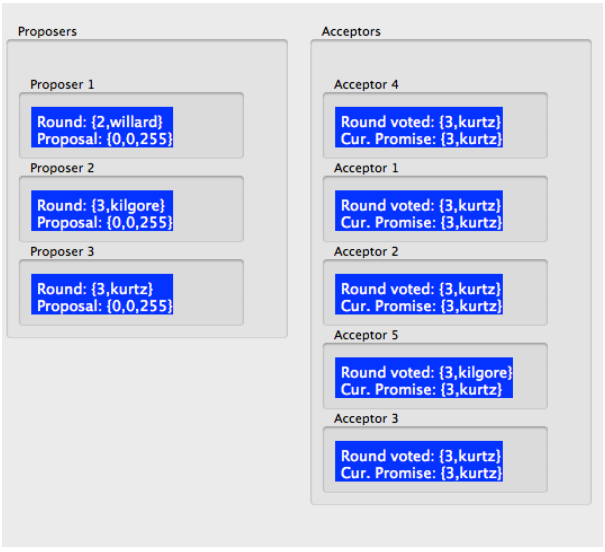


fig: acceptors dropping vote messages results in longer time to reach consensus than normal

In Experiment 4, we increase the number of acceptors and proposers.
First we make the number of proposers 6 and the number of acceptors 9 and get the expected consensus, just after a longer time of 14 rounds.

fig: initial state of 6 proposers and 9 acceptors

fig: end state of new system after agreement

Setting the number of proposers to 6 and the number of acceptors to 3 also similarly reaches consensus though with greater time again.

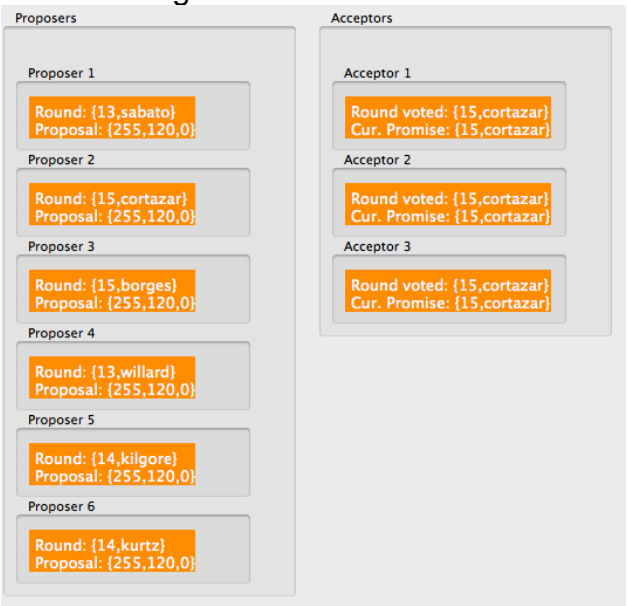


fig: final state of 6 proposers and 3 acceptors

In Experiment 5, we split the paxy module in two parts and make the needed adaptations to enable the acceptors (with the gui) to run in one machine and the proposers in a different one.

as an identifier to communicate between different two erlang instances. We first register the process name in an instance, and refer to another instance by making a call to full name of the other.(process@instanceid). Then once we have established that method of communication, we can refer to the registered names in the other one. So in the paxyacceptor.erl module we'll have the acceptor logic and gui running with the following acceptors:

Register Acceptor a with AccId <0.48.0>
Register Acceptor b with AccId <0.47.0>
Register Acceptor c with AccId <0.46.0>
Register Acceptor d with AccId <0.45.0>
Register Acceptor e with AccId <0.44.0>

Then instead of starting erl, and then calling paxyacceptor:start() as we normally would, we do something like the following:
erl -name paxya@diego -setcookie cookiemonster
p = spawn(paxya@diego, paxyacceptor, start, [123213]).

At this point, the acceptors should be waiting to receive messages from the proposers we will set up now. As the proposers will have knowledge of the names of these acceptors, they should be able to send messages of the sort { a , paxya@diego} ! "some message" .
To do so we would make paxyproposer.erl only start off proposers with the list of acceptor names from before and then in the definition of send() in proposer.erl change it to include the context of the acceptor so:

```
{Name, paxya@diego} ! Message.
```

Then similarly to how we started paxyacceptor, we would do the following in a new terminal window (to simulate a different machine):
erl -name paxyp@diego -setcookie cookiemonster
p2 = spawn(paxyp@diego, paxyproposer, start, [123213]).

This should then start sending prepare and accept messages to the acceptors, which should then in turn return the same values, but we kept receiving errors and weren't able to exactly see what the problem with paxyproper.erl was, but we think it has to do with the manner in which proposers are being generated because of not calling the gui directly, which is what gives them their PropIds in the original paxy module. Due to these problems we encountered, we were not able to complete this experiment fully.

Fault tolerance

In order to make the implementation fault tolerant the acceptor needs to remember what it promises and what it votes for. The module pers, allows us to recover our state to the state we had when we crashed and store state changes as we make promises, thus when prepare and accept messages are received and after changing the acceptors state on the gui, we should store our state using the store(Name, Pr, Vt, Ac, Pn) function in pers with our current state. We then need to be able to simulate a crash so we place the crash procedure in the paxy controller module, and then call it when erlang is running. Before that we need a way to actually load our state in the first place, and for that we use the read(Name) function in pers which will return a null state or our previously stored state based on if it finds an object for that Name. Then we run the paxy:start(231). and during the simulation, we run paxy:crash(b,12). to crash acceptor b. Screenshots from our session show this:

```
[Acceptor b] set gui: voted {11,willard} promise {11,willard} color {0,0,255}
[Acceptor c] set gui: voted {11,kilgore} promise {11,willard} colouur {0,0,255}
[Acceptor a] set gui: voted {11,kilgore} promise {11,willard} colouur {0,0,255}
[Acceptor e] set gui: voted {11,willard} promise {11,willard} color {0,0,255}
4> paxy:crash(b,12).
Call Crash! on btrue
[Acceptor b] gets RE-LOADed FROM CRASH with values promise: {11,willard},voted: {11,willard}!!!!!!!!!
[Acceptor d] set gui: voted {11,willard} promise {11,willard} color {0,0,255}
[Proposer willard] [a,b,c,d,e] decided {0,0,255} in round {11,willard}
[Acceptor c] set gui: voted {11,willard} promise {11,willard} color {0,0,255}
[Acceptor a] set gui: voted {11,willard} promise {11,willard} color {0,0,255}
```

After this crash and reloading of the acceptor instance, the program continues and achieves a consensus as expected.

Part 4:

In this part, we implement improvements that could be made in the proposer to pre-emptively abort the ending of a ballot that has already been negatively decided. We do so by keeping track of how many "sorries" must be received before we can end the ballot (for either collect or actually voting messages received). The number of sorries turns out to be Quorum – 1. This does actually shorten the overall time of consensus, but seems to have some initial lag time starting up (though this could be due to the prior fault tolerance instances not being deleted properly). Below is a screen shot of the code print out, depicting a simulation run that where Proposer Kurtz gets his proposal collection accepted, but then fails to gain enough votes in a ballot which is preemptively ended because of too many sorries being received.

```
[Acceptor b] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,kurtz} colouur {0,0,0}
[Acceptor c] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,kurtz} colouur {0,0,0}
[Proposer kurtz] GETS ACCEPTED and should now send accept out.. set gui: Round {0,kurtz} Proposal {255,0,0}
[Acceptor d] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,kurtz} colouur {0,0,0}
[Acceptor e] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,kurtz} colouur {0,0,0}
[Acceptor c] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,willard} colouur {0,0,0}
[Proposer kilgore] has BALLOT REJECTED due to sorries {0,255,0}
[Proposer kilgore] set gui: Round {1,kilgore} Proposal {0,255,0}
[Acceptor c] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {1,kilgore} colouur {0,0,0}
[Acceptor a] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,willard} colouur {0,0,0}
[Acceptor b] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,willard} colouur {0,0,0}
[Proposer willard] GETS ACCEPTED and should now send accept out.. set gui: Round {0,willard} Proposal {0,0,255}
[Acceptor d] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,willard} colouur {0,0,0}
[Acceptor e] received PREPARE and ROUND > PROMISE so set gui: voted {0,0} promise {0,willard} colouur {0,0,0}
[Proposer kurtz] has NOT ENOUGH VOTES due to sorries {255,0,0}
```

5 Personal Opinion

Although it seemed possibly unnecessary for comparison purposes, one could theoretically profile the time of experiments as follows:

eprof:start().
eprof:start_profiling([self()]).
paxy:start(123123).
eprof:stop_profiling().
eprof:log(experiment3).
eprof:analyze(total).

This will create a log file "experiment3" which will contain a table of the results of the profiling, but deciphering that file becomes a task in and of itself unnecessary for general comparison. Additionally, timer:tc(module,function,arguments) could be used through out, but that would only give us individual function times which is not necessary. It would have been interesting to see how to do these types of comparisons in the assignment because alone we weren't able to find a simple way to get a global view of time that would be consistent between experiments, and giving round counts themselves didn't seem to be truly representative as delays caused things to differ quite a bit in reality, but were necessary to me to get simulation runs that were varied. The assignment seemed interesting and fair to me so I would recommend it for next year, though it may help to give the acceptors and proposers some imaginary context like