Seminar Report: seminar Paxy

Shiva Toutounchiavval, Chenxi Li

December 8, 2014

1 Introduction

The seminar is mainly about the Paxos algorithm which can gain consensus in a distributed system. Paxos is a flexible and fault tolerant protocol for solving the consensus problem, where participants in a distributed system need to agree on a common value, such as the commit or rollback decision typically made using a two or three phase commit. It doesn't matter to the algorithm what this value is, as long as only a single value is ever chosen. We simulate a scene to illustrate this algorithm: there are some processes which can propose values and they will reach an agreement on the value finally.

The main topic related to distributed systems it covers is about how to reach consensus under the condition of perfect communication and faulty processes in an asynchronous system.

2 Work done

There are two key invariants in the algorithm: P1. An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having a number greater than n. P2. For any v and n, if a proposal with value v and number n is issued, then there is a set of S consisting of a majority of acceptors such that (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors S.

If these invariants hold, the algorithm can be guaranteed to work well to reach consensus. There, all design decisions are made to hold the invariants.

Acceptor and proposer have some important states: Promised: not to accept any ballot below this number, Voted: highest ballot number voted so far, Value: the value coming with Voted, Round: the current ballot of a proposer.

Acceptor: It receives two types of messages: prepare, Proposer, Round If Round is higher than Promised, the acceptor will promise not to vote for any sequence number below Round. If Round is lower than Promised, the

acceptor will not accept it. accept, Proposer, Round, Proposal If Round is higher than Promised, the acceptor will vote for it.

These two designs hold P1.

Proposer: In every round, the proposer will send a prepare message to confirm whether it can make a proposal. After receiving promise message from a majority of acceptors, the proposer will select and send the proposal with the highest sequence number. If no votes have been made, the proposer will make its own proposal. And proposals with sequence number less than the proposer will not be voted.

These designs hold p2(a) and p2(b)

3 Experiments

Experiment Strategy: Each individual experiment is done for 10 times and average round number for all proposers is calculated every time. Finally the average round number for the whole experiment in all 10 experiments is calculated.

We first run the original code and change the sleeping time of each proposer to show different results where we find it will take more rounds to reach consensus when proposers run at the same time. They are also used to compare performance of the system with the following experiments.

Table 1: proposers sleep [1000,1000,1000]

Table 2: proposers sleep [1000,2000,3000] and [1000,3000,2000] respectively

proposer sleep time	A	B	C	average
[1000,1000,1000]	0.3	1.4	1.3	1
[1000,2000,3000]	0	1	0	0.33

Table 3: round for proposers with different sleep time

Table 4: proposers sleep [2000,1000,3000] and [3000,1000,2000] respectively

Table 5: proposers sleep [2000,3000,1000] and [3000,2000,1000] respectively

3.1 Introducing Delays in the Acceptor

We do three kinds of experiments when 3 proposers and 5 acceptors in delaies and the algorithm still terminates everytime:

- only delay after receiving prepare messages

proposer sleep time	proposer A	proposer B	proposer C	average
[1000,1000,1000]	1.1	2.6	1.9	1.87
[1000,2000,3000]	0.3	1.3	0	0

Table 6: round for proposers with delay after receiving prepare messages

- only delay after receiving vote messages

proposer sleep time	proposer A	proposer B	proposer C	average
[1000,1000,1000]	1.1	3.1	2.2	2.13
[1000,2000,3000]	0.7	1.6	0.6	0.97

Table 7: round for proposers with delay after receiving vote messages

- delay after both receiving preare and receive messages

proposer sleep time	proposer A	proposer B	proposer C	average
[1000,1000,1000]	2	2.6	2.6	2.4
[1000,2000,3000]	0.1	1.1	0	0.4

Table 8: round for proposers with both delay in acceptor

The more delaies there is, the more rounds it will take to reach consensus.

3.2 Avoid Sending Sorry Messages

We do three kinds of experiments when 3 proposers and 5 acceptors in not sending sorry messages and proposers can come to an aggreement everytime:

- not sending in the 1st phase
- not sending in the 2nd phase
- not sending in both phases

As we can see from table 3.2, not sending sorry messages has no effect on the number of rounds to reach consensus.

proposer sleep time	always send	1st phase	2nd phase	both phases
[1000,1000,1000]	1	1	1	1

Table 9: round for proposers without sending sorry messages

3.3 Randomly Dropping Messages in the Acceptor

- We do three kinds of experiments when 3 proposers and 5 acceptors in dropping messages.
  - only drop 1/10 promise messages
  - only drop 1/10 vote messages
  - drop both 1/10 promise and 1/10 vote messages

proposer sleep time	no drop	1/10 promise	1/10 vote	1/10 both
[1000,1000,1000]	1	1.1	1.13	2.3

Table 10: round for proposers when dropping different messages

Dropping promise messages and vote messages have a little and similar effect. But dropping both messages have a big effect on the algorithm performance.

- We change the percentage of dropped messages based on dropping both promise and vote messages
  - drop 1/5 promise and vote messages
  - drop 1/2 promise and vote messages
  - drop all promise and vote messages

proposer sleep time	no drop	1/5 both	1/2 both	1 both
[1000,1000,1000]	1	1.23	6.47	never

Table 11: round for proposers when dropping different percent of messages

The effect of dropping a percentage of both promise and vote messages is expoentially increasing.

It is possible that the algorithm reports conflicting answers when a majority of accepters have voted for a proposal and some of the vote messages are dropped. The proposer should have reach consensus if no messages are droped. But in the case of dropping messages, the proposer will continue.

3.4 Increase the Number of Acceptors and Proposers

We do three kinds of experiments when 3 proposers and 5 acceptors on increasing the number of acceptors and proposers:

-
- only increase the number of acceptors 1,2,3, which means there are 3 proposers and 6,7,8 acceptors.

proposer sleep time	[3,6]	[3,7]	[3,8]
[1000,1000,1000]	1	1	1

Table 12: round for proposers with different proposers and accepters

- only increase the number of proposers 1,2,3, which means there are 4,5,6 proposers and 5 acceptors.

proposer sleep time	[4,5]	[5,5]	[6,5]
[1000,1000,1000]	1.5	1.68	0

Table 13: round for proposers with different proposers and accepters

- increase the number of both acceptors and proposers {1,1},{2,2},{3,3} which means there are 4 proposers 6 acceptors, 5 proposers 7 acceptors and 6 proposers 8 acceptors.

proposer sleep time	[4,6]	[5,7]	[6,8]
[1000,1000,1000]	1.5	1.92	2.38

Table 14: round for proposers with different proposers and accepters

As is shown in the table 3.4 and3.4, the number of acceptors has no effect on the number of rounds it takes for proposers to reach consensus. But the number of proposers has a big effect, and the larger the number is, the more rounds it will need to reach consensus.

3.5 Split the Paxy Module

we replace paxy.erl with paxy_acceptor.erl and paxy_proposer.erl. With distributed programming in Erlang, processes can communicate with each other.

When starting Erlang, we make it network aware by providing a name:

- erl -sname acceptor -setcookie 123456
- erl -sname proposer -setcookie 123456

Then proposers can send messages to accepter a in a different machine via {a,acceptor@node_name} ! message.

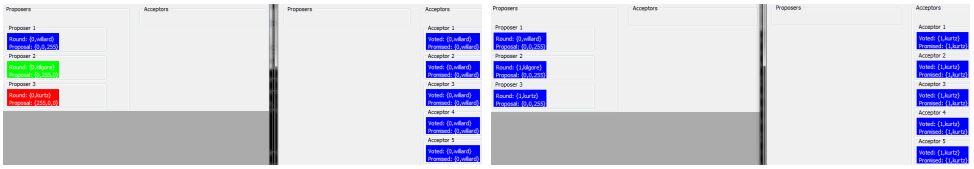

Table 15: after splitting the paxy module

4 Fault Tolerance

pers:read() should be added in init(Name,PaneId)
pers:store() should be added right after sending promise message and vote message

In order to make sure the acceptor really recovers, we do two experiments:

- not restart the acceptor after exiting
- restart the acceptor after exiting

Table 16: **left:** not restarted after crashing **right:**restarted after crashing

When acceptor 5 is not restarted after exiting, it remains Voted{0,willard} and never recovers after crashing. And when acceptor 5 is restarted after

exiting, it recovers from crashing. Therefore, we can conclude that our fault tolerance version works well after crashing.

5 Improvement Based on Sorry Message

After receiving sorry messages from a majority of acceptors, it means the proposer will never receive promise messages or vote messages from a majority of acceptors. Therefore, the ballot will abort in order to get better efficiency.

We do experiments

```
collect(_,0,_,_,_)->
    abort;
collect(0,_, _, _, Proposal) ->
    {accepted, Proposal};
collect(N,Ns, Round, MaxVoted, Proposal) ->
    receive
        {promise, Round, _, na} ->
            collect(N-1,Ns, Round, MaxVoted, Proposal);
        {promise, Round, Voted, Value} ->
            case order:gr(Voted, MaxVoted) of
                true ->
                    collect(N-1,Ns, Round, Voted, Value);
                false ->
                    collect(N-1,Ns, Round, MaxVoted, Proposal)
            end;
        {promise, _, _, _} ->
            collect(N,Ns, Round, MaxVoted, Proposal);
        {sorry, {prepare, Round}} ->
            collect(N,Ns-1, Round, MaxVoted, Proposal);
        {sorry, _} ->
            collect(N,Ns, Round, MaxVoted, Proposal)
    after ?timeout ->
            abort
    end.

vote(_,0,_)->
        abort;
vote(0,_, _) ->
    ok;
vote(N,Ns, Round) ->
    receive
        {vote, Round} ->
            vote(N-1,Ns, Round);
```

```
        {vote, _} ->
            vote(N,Ns, Round);
        {sorry, {accept, Round}} ->
            vote(N,Ns-1, Round);
        {sorry, _} ->
            vote(N,Ns, Round)
    after ?timeout ->
            abort
    end.
```

6 Personal Opinion

Provide your personal opinion of the seminar, including whether it should be included in next year's course or not.