ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

HS 2009 Prof. Dr. Roger Wattenhofer, Thomas Locher, Remo Meier, Benjamin Sigg

Distributed Systems
Solution 4

Assigned: November 27, 2009
Discussion: December 4, 2009

1 Multi-Paxos

It is often necessary to have a sequence of decisions rather than a single decision. One solution would be to run (standard) Paxos multiple times. This solution, however, is not very efficient.

Multi-Paxos is a version of Paxos that requires less rounds to make a sequence of decision. The first time Multi-Paxos runs, it behaves exactly like Paxos. But in every subsequent call it runs an optimized algorithm, using fewer rounds than Paxos.

Multi-Paxos does not alter or introduce new messages, it only removes unnecessary messages from Paxos.

a) How does a subsequent call of Multi-Paxos work? To answer this question have a look at slide 7/35, remove any unnecessary line of the pseudo-code.
Hint: What is not known in the start phase, but in any subsequent phase?

b) Assuming there are no failures at all: What is the minimum number of acceptors that a proposer needs to contact? And which acceptors would the proposer contact? Of course, all servers should have about the same amount of work.
Warning: all proposers contacting the same acceptor would work, but the workload would not be distributed evenly.

c) How could Multi-Paxos protect itself agains Byzantine failures happening in a subsequent call?
Hint: Byzantine Multi-Paxos requires exactly one round more than Multi-Paxos.

Solution

a) First we assume that the same proposer is used to build a sequence of decisions. If a proposer successfully brings a majority of acceptors to accept its decision, then these acceptors must have the same x_{last}, n_{last} and n_{max} (otherwise they would not acknowledge the choice). In the subsequent call the proposer can use this fact and send the `proposal` message directly, and just ignore the `prepare` message. The proposer has to reuse its last request number.

What happens if there is another proposer? The other proposer must use another request number. If the second proposer sends `prepare` messages, then the new messages just override the values used by the first proposer. The first proposer will not receive enough acknowledgments when sending `propose` to decide. But that is no problem, as the proposer may just try again (with a new sequence number and new `prepare` messsage). In the worst case the protocol never ends, but that is highly unlikely.

b) Normally (Multi-)Paxos contacts a majority of nodes. This means that if two proposers run concurrently, then there is at least one acceptor which is contacted by both proposers. This (or these) shared acceptor ensures that one of the proposers cannot collect enough acknowledgments to make a decision.

One advantage of choosing a majority set is, that it is not important how it is chosen. The intersection of two majority sets always contains an acceptor.

But if there are no failures at all, we could allow a proposer to choose a smaller set of acceptors. The only thing we have to ensure is, that if two proposers run in parallel, the intersection of their sets contains at least one acceptor. We can do this using sets of size $O(\sqrt{n})$. See figure 1 to see how the sets could be chosen.

If there are not enough acceptors to build a triangle like in figure 1 we just do not fill up the last row. The last proposers then chooses some acceptors from the second last row.

c) The easiest solution is: all acceptors compare their `propose` message by broadcasting them. If a majority has the same `propose` message, then the protocol will work, otherwise there must be Byzantine servers around. In such a case Paxos could just restart and try again.

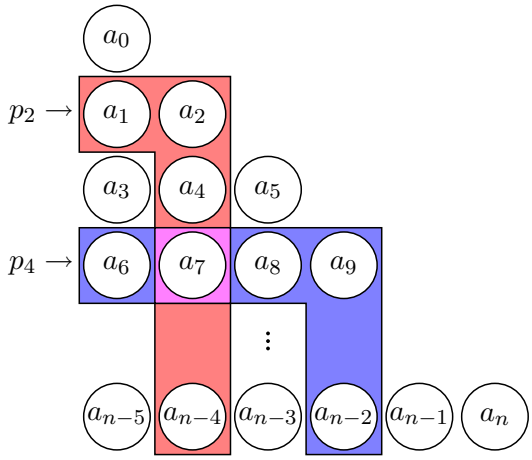

Figure 1: How proposers p have to choose sets of acceptors a such that the intersection of two sets always contains at least one acceptor. Proposer p_2 contacts acceptors in the red area $(a_1, a_2, ...a_{n-4})$ while proposer p_4 contacts acceptors in the blue area $(a_6, a_7, ..., a_{n-2})$. The acceptor a_7 is contacted by both proposers. Either a_7 accepts the proposal of p_2 or of p_4, but it is not possible for a_7 to accept both proposals.

2 PBFT

See slide 7/58 ff.

a) Why is there a prepare phase? Can you given an example where the algorithm would fail without the prepare phase?

Let us assume there are exactly $n = kf + 1$ servers, with $k \geq 3$ and f the number of faulty servers.

b) How many `(pre-)prepare` messages must a server receive until it is prepared (and sends the `commit` message)?

c) Using your result from from b), prove that if a correct backup server prepared for message m, then no other correct backup server is prepared for another message m' (unless, due of very bad luck, the hash of both messages are equal).

Currently each backup server broadcasts its `prepare` and `commit` messages. From b) you know that backup servers do not require n messages in order to decide about their next step. As a result some messages are not needed at all.

d) Modify the prepare and commit phase so that as few messages as possible are sent. If the primary is correct, then your protocol should ensure that all correct backup servers commit.

e) Prove: if the primary is correct, then your modified protocol from d) works and all correct processes commit.

f) (Optional, open question) For your protocol from d): Assume the primary is Byzantine. The primary tries to remain primary, e.g. he must ensure that at least one correct process commits, otherwise the client notices that something is wrong. How many wrong messages can the faulty primary send before too many server notice his behavior? What kind of damage can he inflict? Assume a worst case scenario for the Byzantine primary.

Solution

a) The pre-prepare and prepare phases of the algorithm guarantee that correct backup servers agree on a total order for the requests within a view.

With other words, the prepare phase synchronizes the servers such that all commit the requests in the same order.

An example without prepare phase

Remember our model, we can have message re-ordering and message delay.

1 The primary sends a request, which is received by all backup servers. Each backup server broadcasts a `commit` message. Unfortunately all `commit` messages, except $2f + 1$ heading for server x, are delayed. While server x commits, the other servers do nothing.

2 The primary sends another request, which is received by all backup servers. Each backup server broadcasts the next `commit` message. This time the messages are not delayed, and all servers commit.

3 The delayed `commit` messages from the first request arrive. All servers, with the exception of server x, commit the first request.

So different servers commit the requests in different orders, you can easily imagine some requests leading to inconsistency in the system...

How would the prepare phase help against such a problem? In order for a backup server to commit it has to receive $2f + 1$ `commit` messages. If some messages are delayed the $2f + 1$ servers remain in the commit phase. If they would now receive the next `prepare` messages, they would notice the problem and not (yet) send the next `prepare` message.

Questions and answers

Why can't we use the view-number, time-stamp or sequence-number to achieve the same effect?

- The view-number only changes if the primary changes, this might not happen often.
- The time-stamp carries no information whether there was any message before (example: you may know your train arrived at 9:14 at the station, but this information doesn't tell you whether another train arrived at 9:13)
- The sequence number could be altered by any Byzantine server.

b) A backup server must receive enough messages to build a quorum. In this case a quorum is a set of size s with $s > n/2 + f/2$.

c) Proof by contradiction: Assume backup server a is prepared for m, backup server b is prepared for m'. Both must have received quorum-many `(pre-)prepare` messages. This means, at least one correct server sent two different messages, but a correct server would never do such a thing.

d) Any correct backup server must receive s (= size of a quorum) `(pre-)prepare` and s `commit` messages in order to commit. To achieve this goal, every correct server must deliver s messages to the other correct servers. Because a correct server does not know which servers are faulty, it has to send slightly more than s messages. Every server has to send $s + f$ `(pre-)prepare` and $s + f$ `commit` messages. The i'th server sends its messages to the servers $\{i + 1, i + 2, ..., i + 1 + s + f\}$ (modulo n).

e) If the primary is correct:

- All $n - f$ correct backup servers receive `pre-prepare` and send $s + f$ many `prepare` messages.
- All $n - f$ correct backup servers receive at least s many `prepare` messages and send $s + f$ many `commit` messages.
- All $n - f$ correct backup servers receive at least s many `commit` messages and commit.

f) The exact answer depends on your protocol from d), but in general the Byzantine primary can do almost no damage.

Forge messages

If the primary sends out only one forged message to a correct server, then the correct server distributes the forged message to s many other correct servers. All of them notice something is wrong, and issue a view change request. Because s servers are enough to elect a new primary, the Byzantine primary must not send even one forged message.

Omit messages

The Byzantine primary sends the `pre-prepare` messages only to the first s correct servers. These s servers send `prepare` messages, but can only get $s + c_0$ (with c as small constant) correct servers into the prepare-state. These prepared $s + c_0$ servers then trigger only $s + c_1$ correct servers to commit.

If the primary always sends its `pre-prepare` message to the same s servers, then always the same servers commit. The other servers become useless and their data gets more and more out-dated.