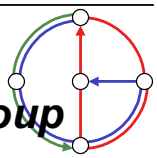
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group



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

Distributed Systems
Theory exercise 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.

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.