Caltech, CMS. CS/IDS 142: Lecture 9.1
Paxos, Byzantine, Snapshots: Review
Mani Chandy
25 November 2019

1. Review and proof of Paxos.
2. Review of Byzantine
3. Review of Global Snapshots
4. Thoughts about the course

Review of Paxos and Proof of Correctness.

- Phase 1:
  - **Prepare(t)** message from proposer to acceptor
  - **promise(promise_t, accepted_t, value)** reply from acceptor to proposer
- Phase 2:
  - **Request(t, value)** message from proposer to acceptor
  - **Accept**(**t**, **value**) message from acceptor to learner
- *Learner learns the proposal chosen by majority of acceptors.*
- Different proposers never use the same t value.

Break ties lexicographically

Algorithm for proposer P:

State: (P.t, P.value) Initially (-1, x)
Start timer
While not timed_out:

choose t greater than P.t and set P.t = t
PHASE 1
1. send prepare(P.t) to all acceptors
2. wait for promise(prepare_t, accept_t, value) replies from (at least) a majority of acceptors where prepare_t == P.t
3. If value is not None for one or more of these promise messages then set P.value to the value in the promise message with the largest accept_t

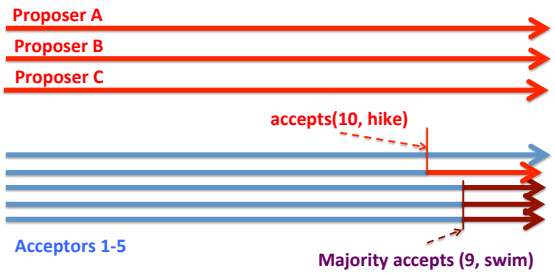
PHASE 2
4. send request(P.t, P.value) to all acceptors.
5. Wait for accepted(t, value) replies from majority of acceptors where (t, value) == (P.t, P.value)

Algorithm for acceptor A:

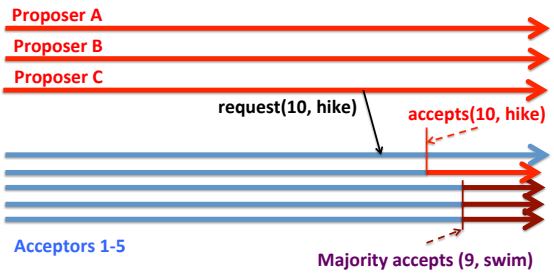
State: (A.t, A.accepted_t, A.value) Initially (None, None, None)
Start timer
While not timed_out:
upon receiving prepare(t):
if t >= A.t:
A.t = t
reply with promise(t, A.accepted_t, A.value)
upon receiving request(t, value):
If t >= A.t:
A.t = t ⟵ correction
reply with accepted(t, value)

Theorem: If majority accepts (t, v) at some point then no process at the same point has accepted (t', v') where t' >= t and v' != v

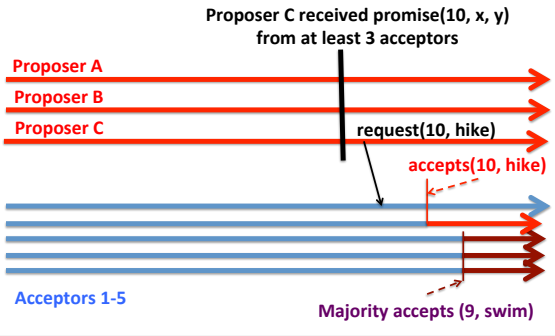
The situation shown in the diagram cannot happen. Why not?



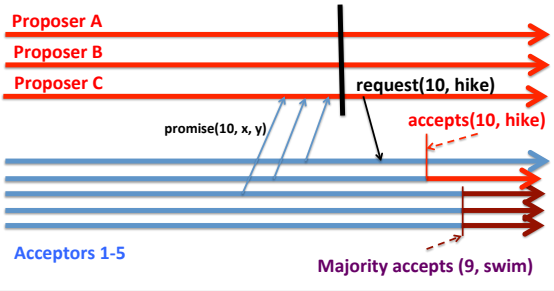
If an acceptor accepted (10, hike) at a point p then it received request(10, hike) at an earlier point p'



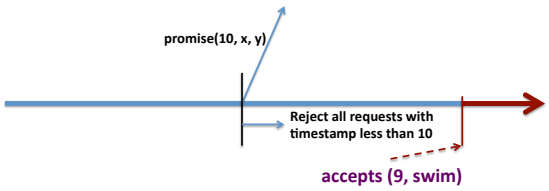
If a proposer sent request(10, hike) at point p'' then it received promise(10, x, y) from a majority of acceptors.



Proposer C received promise(10, x, y) from at least 3 acceptors, *and at least one of these 3 is in the majority that accepted (10, hike)*

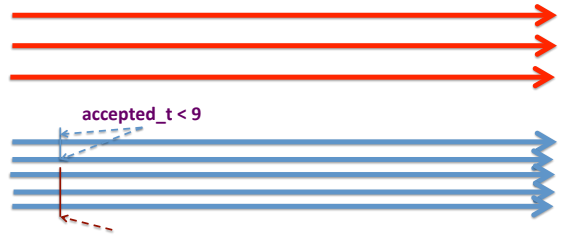


Contradiction!
An acceptor that sent promise(10, x, y) will reject request(9, swim) because 9 < 10.

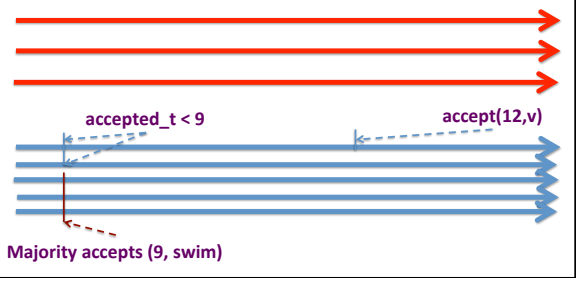


Theorem: If a majority of acceptors accept (T, V) at some point then from then on a majority of acceptors accept value V.

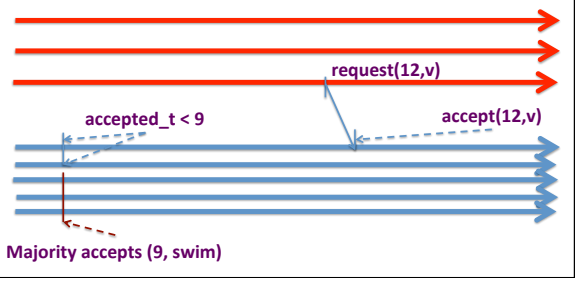
(Note the acceptance timestamps of these acceptors may change, but the value, V, remains unchanged.)



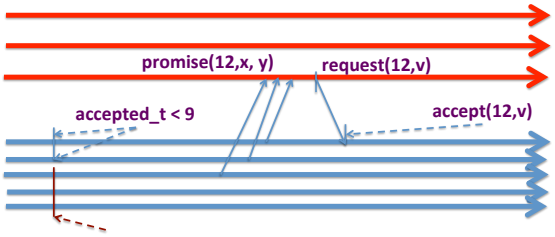
Consider the next acceptance accept(t, v), say accept(12, v), after this point.



Consider the next acceptance, accept(12, v), after this point. This was preceded by a proposer sending request(12, v).

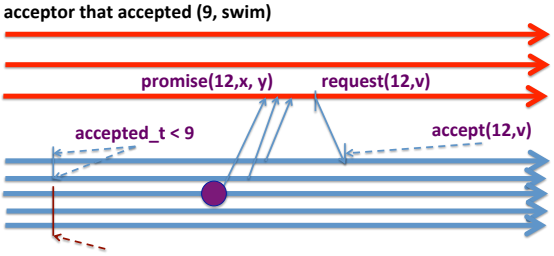


Consider the next acceptance accept(12, v) after this point.
This was preceded by a proposer sending request(12, v).
Which was preceded by that proposer receiving promise(12, x, y) from a majority of acceptors.



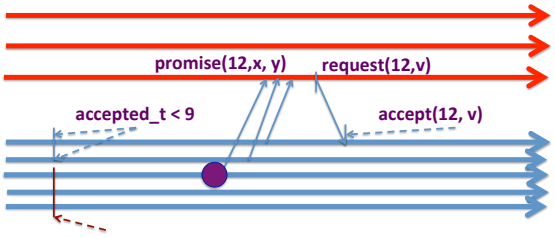
Majority accepts (9, swim)

Consider the next acceptance accept(12, v) after this point.
This was preceded by a proposer sending request(12, v).
Which was preceded by that proposer receiving promise(12, x, y) from a majority of acceptors.
At least one of the promise(12, x, y) messages was sent by an acceptor that accepted (9, swim)



Majority accepts (9, swim)

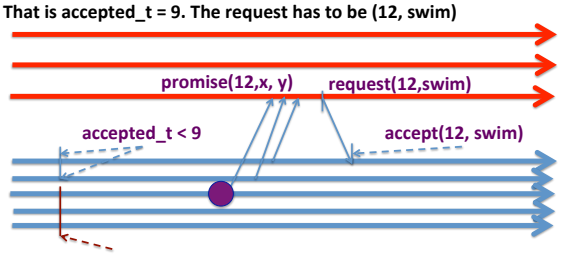
At least one of the promise(12, x, y) messages was sent by an acceptor that accepted (9, swim). So, the promise messages are:
1. accepted_t = 9, value = swim, or
2. accepted_t < 9



Majority accepts (9, swim)

At least one of the promise(12, x, y) messages was sent by an acceptor that accepted (9, swim). So, the promise messages are:
1. accepted_t = 9, value = swim, or
2. accepted_t < 9

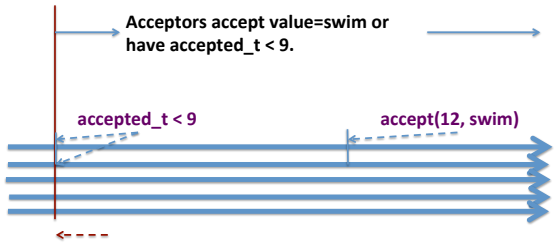
So, the proposer requests the promise with the largest accepted_t . That is accepted_t = 9. The request has to be (12, swim)



Majority accepts (9, swim)

Induction hypothesis: At each point after majority accepted [T, V], all acceptors have accepted value V or have accepted_t < T.

After majority accepted (9, swim) all acceptors have either accepted value V or have accepted_t < 9.



Byzantine Generals Review

Consider system with 1000,002 officers where up to 1000,000 are faulty. So at least 2are non-faulty.

If the general is non-faulty then there is at least 1 non-faulty commander, who must obey the general's command.

If the general is faulty, then there are at least 2 non-faulty commanders and they must come to a consensus.

Byzantine Generals Review

Consider system with 1000,002 officers where up to 1000,000 are faulty. So at least 2are non-faulty.

If the general is non-faulty then there is at least 1 non-faulty commander, who must obey the general's command.

If the general is faulty, then there are at least 2 non-faulty commanders and they must come to a consensus.

Case: General is non-faulty.

Round 1. General sends attack or retreat message.
Non-faulty commanders get this message at the end of round 1.
Round 2: If the general sent attack, then non-faulty commanders commit to attack for rounds 2, 3, …. , 1000,001.
If the general sent retreat, then the general NEVER sends an attack message. Since messages cannot be forged, no commander ever gets an attack message, or a copy of an attack message, with the general's signature. So no non-faulty commander commits to attack on rounds 1, 2, 3,…. , 1000,001.

Case: General is faulty.
If on any round n (where n = 1, 2, …) a non-faulty commander gets an attack message signed by the general and attack messages signed by at least n-1 commanders, then the commander sends copies of the n attack messages it received and sends one more attack message that it signs for a total of n+1 and this commander commits to attack in round n+1, n+2, n+3, …….

Case: General is faulty.

So, if any non-faulty commander commits to attack by the end of round n where n = 1, 2, …., 1000,000 then ALL non-faulty commanders commit to attack on rounds n+1, n+2, ……, 1000,001.

Case: General is faulty.

So, if any non-faulty commander commits to attack by the end of round n where n = 1, 2, …., 1000,000 then ALL non-faulty commanders commit to attack on rounds n+1, n+2, ……, 1000,001.

If no non-faulty commander has committed to attack by the end of round 1000,000 then no non-faulty commander commits to attack at end of round 1000,001. This is because a commander can get at most 1000,000 attack messages on round 1000,0001.

What goes wrong when messages can be forged?

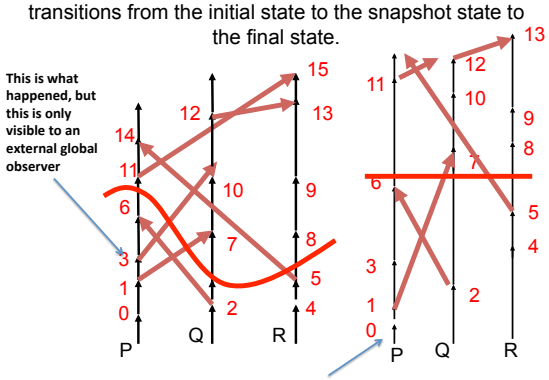
Case with t faulty officers and 2t non-faulty ones. Diagram with t=2

4 good officers

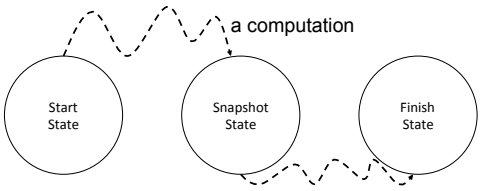
2 bad officers

The 2 (i.e. t) bad officers split the 4 (i.e., 2t) good officers into symmetric halves. Neither half can break the symmetry

Key idea of snapshots: There is a sequence of transitions from the initial state to the snapshot state to the final state.



Key Idea of Proof



There exists a computation from the start state to the snapshot state, and from the snapshot state to the final state.

25

Working on distributed algorithms is enjoyable because:
- the algorithms have massive impact on industry
- The code for each algorithm is often small
- The proofs are tricky. (Many algorithms that seemed reasonable to me initially were wrong.)

Working on distributed algorithms is enjoyable because:
- the algorithms have massive impact on industry
- The code for each algorithm is often small
- The proofs are tricky. (Many algorithms that seemed reasonable to me initially were wrong.)

In 10 weeks we can't describe all the important distributed algorithms. So we chose exemplar algorithms and emphasized a few concepts.

Working on distributed algorithms is enjoyable because:
- the algorithms have massive impact on industry
- The code for each algorithm is often small
- The proofs are tricky. (Many algorithms that seemed reasonable to me initially were wrong.)

In 10 weeks we can't describe all the important distributed algorithms. So we chose exemplar algorithms and emphasized a few concepts.

Take away concepts from the course:
- State transition systems.
- Always nothing bad happens.
- Progress. The system eventually gets closer to its goal.
- Map global view to local agent state
- Data structures, e.g. graphs, that change with computation.

Please provide feedback in the TQR.
Your notes are especially welcome.
Send messages to Richard or me (mani@cs.caltech.edu)

Also, let me know if you are interested in a CS 81 projects course with me.