The Paxos Algorithm
Consensus among Agents in a Faulty Distributed System
Distributed Systems Model

• Modifications to usual model:
  – Agents may halt forever, or halt and restart (from previously saved state).
  – Messages may be lost or duplicated.

• Assumptions: No Byzantine operations
  – Messages are not corrupted
  – Agents are not malicious
Proposers, Acceptors, Learners
Overview

1. Proposers propose values for the consensus value.
2. Acceptors choose among proposed values.
3. Learners learn the value (if any) chosen by the majority of acceptors.
Safety

1. Learners learn at most one value.
   Associated with learner $z$ is a variable $\text{learned}[z]$ where $\text{learned}[z]$ is either None (representing undefined) or some non-None value.
   For $v \neq \text{None}$: stable($\text{learned}[z] = v$)

2. Learners learn only proposed values
   ($\text{learned}[z] \neq \text{None}$) $\Rightarrow$ $\text{learned}[z]$ in set of proposals

3. All learners learn the same unchanging value
   For all learners $z, z'$: ($\text{learned}[z]=\text{None}$) or ($\text{learned}[z']=\text{None}$) or ($\text{learned}[z]=\text{learned}[z']$)
Specification

Progress (NOT GUARANTEED BY ALGORITHM!)

1. Learners eventually learn a value. For all learners \( z \): eventually(\text{learned}[z] \neq \text{None})

   Fischer, Lynch, Patterson (FLP) theorem says that \textit{consensus cannot be achieved with a single faulty process}. More about this later.

   We cannot prove progress; but we will discuss best effort algorithms.
Algorithm Overview

• Phase 1:
  – **prepare** message from proposer to acceptor
  – **granted** reply from acceptor to proposer.

• Phase 2:
  – **accept** message from proposer to acceptor
  – **accepted** message from acceptor to learner

• Final
  – Learner learns the proposal that is chosen.
Algorithm Phase 1 (2 steps)

Proposer sends prepare(n)
actuator replies: granted(n, latest_accepted, latest_accepted_n)

Local variable of proposer:
latest_n: The latest (and highest) value of n send in prepare(n)

Local variables of acceptor:
• highest_prepare_n This value is initially an arbitrarily negative. The highest n received.
• latest_accepted, latest_accepted_n: the latest proposal that was accepted and its n-value.
Algorithm: Phase 1, Step 1
proposers send prepare messages

1. When triggered, a proposer sends a \textit{prepare}(n) message to at least a majority of acceptors where:
   \[ n > \text{latest}_n \]
   and makes \[ \text{latest}_n = n \]

- n could be a pair (t, id) where t is strictly increasing, id is the id of the process, and comparison between pairs are made lexicographically as in our homework. e.g. \((5, 4) > (5, 3)\) and \((5, 4) > (1, 10)\)
- triggering described later
Algorithm: Phase 1, Step 2
acceptors reply with granted messages

2. When an acceptor y receives a prepare(n) message from a proposer x:
   if n > highest_prepare_n:
     send granted(n, latest_accepted, latest_accepted_n) to x
     highest_prepare_n = n
   else: send nack_prepare(n) to y
     # Ignore nacks: used for efficiency not safety.
Ideal Case

The next slides show an ideal scenario. This is merely to explain the algorithm.

The key challenge is to prove the algorithm in “bad situations” where messages are delayed or lost, and agents fail.
Proposers 0 and 1 send \textit{prepare}
Acceptors reply with granted messages

granted([1,0], None, None)
granted([1,1], None, None)
Phase 2: 2 steps

1. Proposer may send accept message to acceptors.
2. Acceptors may send accepted message to learners.
Phase 2, Step 1
proposers send accept messages

A proposer sends an **accept** message if it receives a granted message from a majority of acceptors where n in each of these granted messages equals latest_n

Recall format: granted(n, latest_proposal, latest_proposal_n)
An accept message is (latest_n, my_proposal) where my_proposal is as follows:

• **Proposer’s choice**: If each granted message has latest_proposal = None then my_proposal is any value.
• **Acceptor’s choice**: else: my_proposal is the latest_proposal in a granted message with the highest latest_proposal_n
Proposers send accept messages

Proposer's choice for proposal = v[0,1]
Phase 2, Step 2
acceptors send accepted messages to learners

if acceptor receives an
   accept(latest_n, my_proposal) message
from a proposer where:
   latest_n >= latest_prepare_n
then acceptor accepts proposal v and the acceptor sends an
accepted(my_proposal) message to all learners and sets
   latest_accepted = my_proposal
   latest_accepted_n = latest_n

(Note: acceptor may accept any number of proposals.)
Acceptors send accepted messages to learners

Diagram showing a network of nodes labeled 0, 1, 2, 3, and 4, with arrows connecting them and labels such as `accepted(p[1,0])`.
Final: Learners learn the chosen proposal

A learner learns that proposal $v$ has been chosen if it receives $\text{accepted}(v)$ from a majority of acceptors.
Proof of Safety

• (Recall, algorithm may not progress!)
• Safety: A learned value is a proposed value: Proof is obvious.
• Safety: A learner learns at most one value.
  
  For $v \neq \text{None}$: stable(learned$[z] = v$)
  
  -- Proof hinges on two majority requirements:
    • For proposer’s transition from prepare to accept
    • For learner to determine chosen proposal.
Proof of Safety

Lemma: for v != None and any set S of a majority of acceptors:

\[
\text{stable}(\text{for all acceptors } p \in S: \text{latest}\_\text{accepted} = v)
\]

Proof:

Acceptors accepts v only if it gets accept (latest_n, my_proposal) from a proposer p where latest_n >= highest_prepare_n, and my_proposal = v

Proposer p sends gets accept (latest_n, my_proposal) only if it received from each of a majority set S’ of acceptors:

\[
\text{granted}(n, \text{latest}\_\text{proposal}, \text{latest}\_\text{proposal}\_n) \text{ where } \text{latest}_n = n
\]
Proof of Safety (continued)

There exists at least one acceptor q that is common to S and S’ because both are majority sets.

acceptor q sends granted(n, latest_proposal, latest_proposal_n) where, latest_proposal = v

By the proposer’s choice rule (see slide 15 for phase 2, step 1), proposer p replies to this granted message by sending: accept (latest_n, my_proposal) where my_proposal = v

Hence q’s latest_proposal remains unchanged when q receives this accept message.
Scenario without progress

1. Proposer 0 sends prepare([1,0])
2. Acceptor replies granted([1,0], -, -)
3. Before proposer 0’s accept message gets to the acceptor it gets prepare([1, 1]) from proposer 1.
4. Acceptor replies granted([1, 1], _, _)  
5. Before proposer 1’s accept message gets to the acceptor it gets prepare([2, 0]) from proposer 0
6. Acceptor replies granted([2,0], -, -)
7. .............