Fault Tolerance

Consensus

Pedro F. Souto (pfs@fe.up.pt)

November 9, 2017

Distributed Agreement: Informal

Problem How to ensure that the processes in a group agree on the actions to take?

Observation This is rather ambiguous. What do we mean by **agree**?

- Doesn't atomic commitment require agreement?
- There are actually a few problems very similar, but that are nevertheless different
  - We need to be more rigorous.

Consensus: Formal

Def. Consider a set of processes $1, \ldots, n$

1. Each process starts with an input from a fixed value set V
2. The goal is that each process choose a value in V
3. If a process chooses a value v, that decision is irreversible
4. The values chosen by each process must satisfy the following assertions

   Agreement Only a single value can be chosen by all processes

   Validity If all processes have the same input value v, no value different from v can be chosen

   Termination In any failure free execution, eventually all processes chose a value

The Synod Algorithm: A solution for Consensus (1/6)

Assumptions

Processes

- Operate asynchronously
- May fail and recover
- Have access to stable storage

Messages

- Delays are unbounded
- Messages may be lost or duplicated
- Messages are not corrupted

But Given the impossibility of consensus in asynchronous systems is well known (FLP)

> "We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value."

The Synod Algorithm: Roles and Structure (2/6)

Process Roles

Proposers Send **proposals** to the acceptors

Proposal is a pair (n, v), where n is a unique number (a proposal identifier) and v is some value from V;

Acceptors **Accept** the proposals

- A proposal is accepted if it is **valid** (to be defined)
- A value is **chosen** if a **majority** acceptors accept a proposal with a given value

Learners Learn the **chosen** values

Structure

Phase 1 Find a number that makes the proposal likely to be accepted

- And the value that has been chosen, if any

Phase 2 Submit a proposal

The Synod Algorithm: Phase 1 (3/6)

Phase 1

Proposer Selects a proposal number n and sends a PREPARE(n) request to a majority of acceptors

Acceptor Upon receiving a PREPARE. If:

n is larger than any previous PREPARE request to which it has already responded, then it responds with:

1. a **promise** not to accept any more proposals numbered less than n and
2. with the highest-numbered proposal (if any) that it has **accepted**

Note If n is not larger than that of a previous PREPARE request to which it has already received, then the acceptor does not need to respond

- How does then a proposer learn about a larger number?

The Synod Algorithm: Phase 2 (4/6)

Phase 2

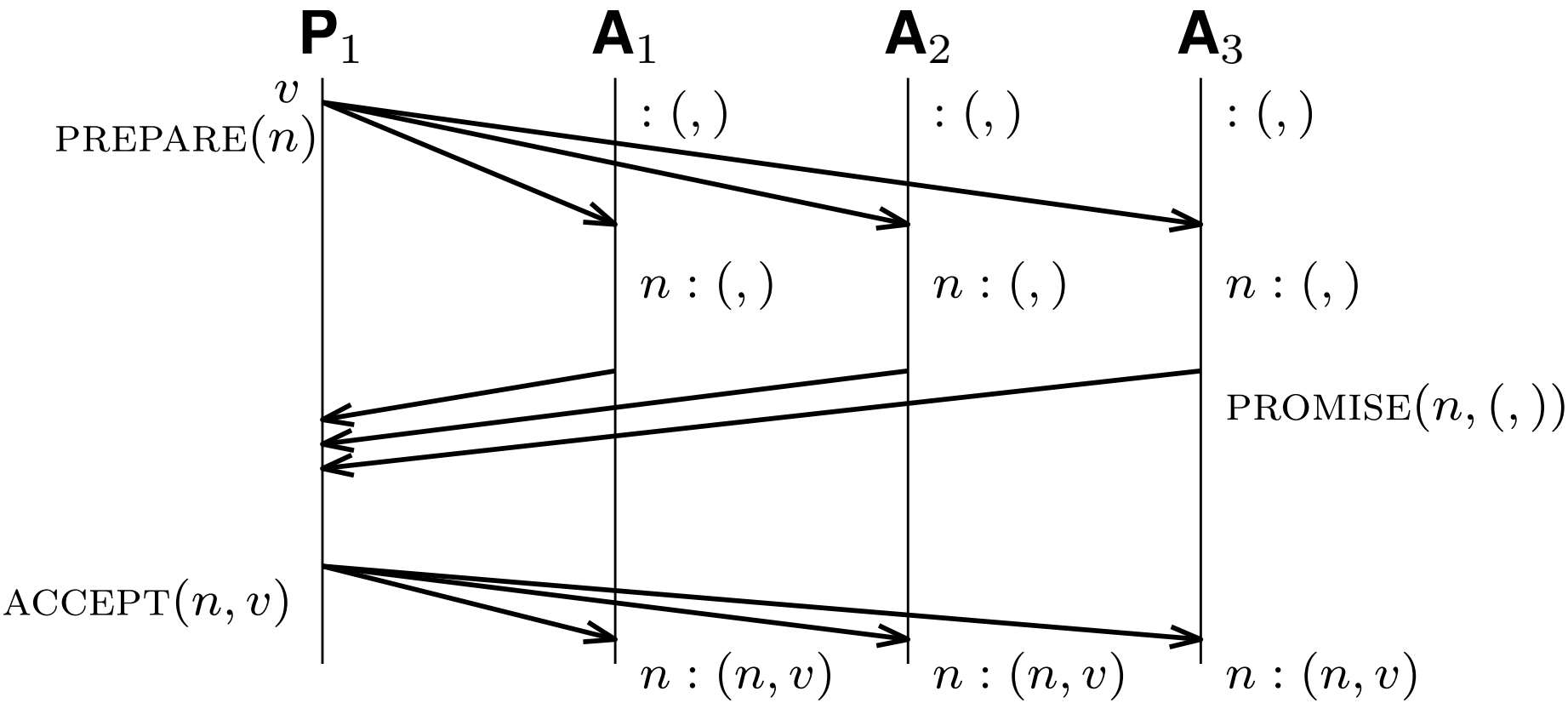
Proposer If it receives a response to its PREPARE requests (numbered n) from a **majority** of acceptors, then it sends an ACCEPT(n, v) request to each of those acceptors for a **proposal** numbered n with value v, where v is:

- the value of the highest-numbered proposal among the responses received in phase 1
- or is the proposer's input value, if the responses reported no proposals

Acceptor If an acceptor receives an ACCEPT request for a proposal numbered n, it accepts the proposal **unless** it has already responded to a PREPARE request having a number greater than n

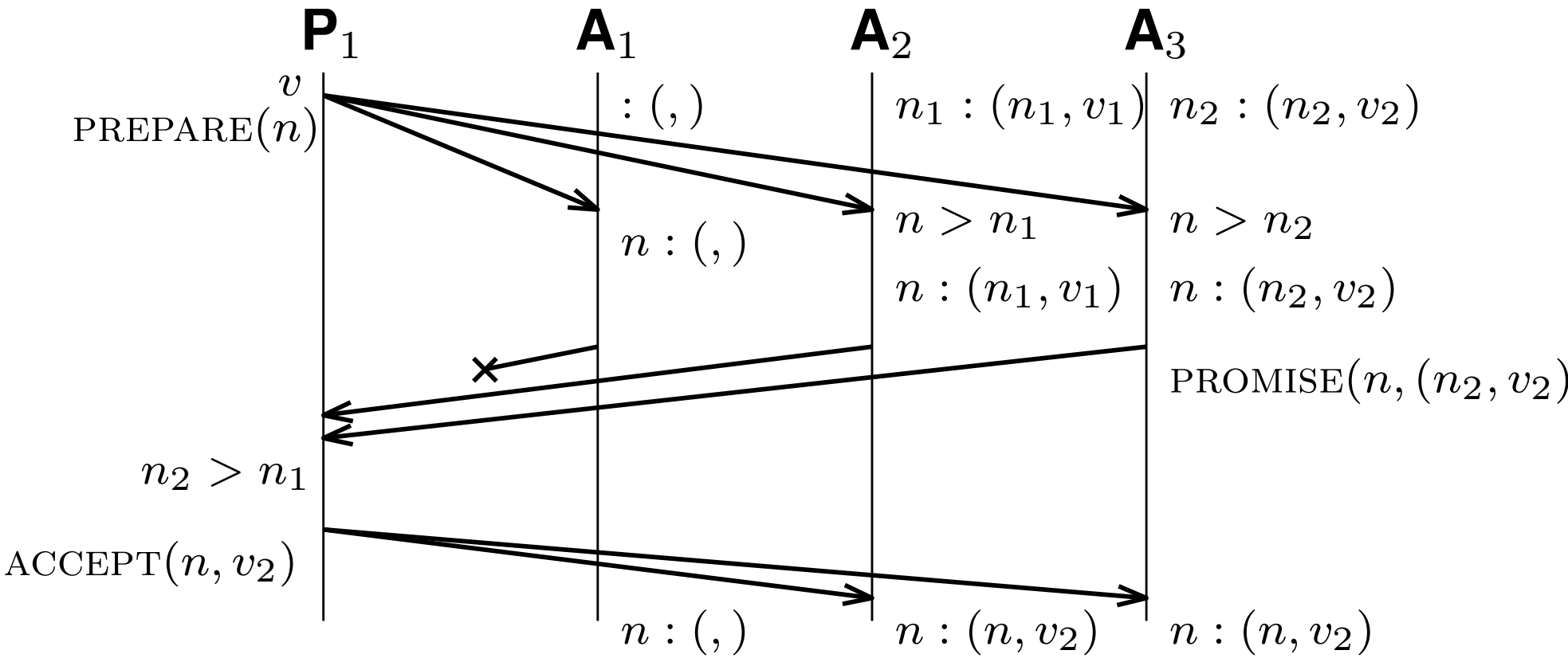
Note The rule to choose the value of a proposal is crucial to ensure that if a value has been chosen then higher-numbered proposals will propose that value

Synod's Execution: Simple



- The PREPARE message does not include any proposed value

Synod's Execution: More Interesting



- Should acceptor $\mathbf{A}_1$, also accept the proposed value?
  - If the proposer also sent $\mathbf{A}_1$ the ACCEPT request

The Synod Algorithm: Correctness Arguments

- If a value is **chosen**, it must have been accepted by a **majority** of the acceptors
- **Once a value has been chosen**, if a proposer gets a response to its PREPARE message from a majority, then the **highest-numbered proposal** it receives in the responses will have the **chosen value**
  - This can be proven by induction
- Therefore, **once a value has been chosen**, **every proposal** submitted (in phase 2), will have the value chosen
- Hence, **once a value has been chosen, every accepted proposal** will have the chosen value

The Synod Algorithm: Learning a Chosen Value (5/6)

- To learn the chosen value, the learner must find out that a **proposal** has been accepted by a majority of acceptors
  1. An acceptor may notify all learners every time it accepts a proposal
     - This minimizes the learning delay
     - But may generate too much traffic
  2. A learner may learn about the acceptance of the proposal from another learner
     - We are assuming non-Byzantine failures
     - Acceptors may notify a distinguished learner, which in turn notifies the other learners
     - The distinguished learner may be chosen by some election algorithm
     - Alternatively, a set of distinguished learners could be used
  3. Because of the loss of messages a learner may not learn about a chosen value.
     - The learner can ask the acceptors what proposals they have accepted, but the failure of an acceptor may make it impossible for a learner to find out if a proposal was accepted by a majority
     - An alternative, is to have a proposer to issue a proposal, using the algorithm described

The Synod Algorithm: Ensuring Progress (6/6)

- Two or more proposers may enter a kind of livelock situation in which the PREPARE from one prevents the other's proposal from being accepted
- To ensure progress, a distinguished proposer must be selected as the only one to try issuing proposals
  - If it finds out that its proposal number is not high enough, e.g. in case there is more than one leader, it can eventually choose a high enough proposal number
- FLP's impossibility result implies that leader election must use either randomness or real time, e.g. timeouts
  - However, the synod algorithm ensures safety regardless of success or failure of the election

Paxos: an Implementation of the Synod Algorithm

- Each process plays the role of proposer, acceptor and learner
- The algorithm chooses a leader, which plays the role of the:
  - The distinguished proposer
  - The distinguished learner
- The messages exchanged are those described
  - Responses are tagged with the corresponding proposal mumber
- Stable storage is used to keep state used by acceptors and that must survive crashes
  - The largest number of any PREPARE to which it responded
  - The highest-numbered proposal it has accepted
  - Updates to stable storage must occur before sending the response
- Uniqueness of proposal numbers is ensured by using a pair (cnt, id) where id is the the proposer's id and cnt is a local counter
  - Each proposer stores in stable storage the highest-numbered proposal it has tried to issue

State Machine Replication

What is? A generic approach to develop a fault-tolerant system, originally proposed by Lamport in his 1978 paper, "Time ..."

Idea

1. Design a service as a determinitistic state machine, which
   - Changes its state (variables)
   - Produces an output

   upon execution of an (atomic) operation
2. Replicate that state machine in different nodes
3. Set all replicas to the same initial state
4. Execute the same sequence of operations in all SM replicas

Challenge How do you ensure that all replicas execute the same sequence of operations?

- Clients may submit different operations "simultaneously"

Solutions

1. Use atomic (total order) multicast
2. Execute Paxos (consensus)

State Machine Replication with Paxos (1/5)

Idea Run Paxos to decide which command should be command n

- The i^{th} instance is used to decide the i^{th} command in the sequence
- Different executions may be run concurrently
- But must ensure that if the same command is chosen by more than one execution, it is executed only once

Normal Operation

- A single server is elected to be the leader, wich plays the role of distinguished proposer
- Clients send commands to the leader
- The leader:
  - Chooses a sequence number for the command
  - Runs an instance of Paxos for that sequence number, proposing the execution of that command
- The leader may not succeed, if:
  - It fails
  - Another server believes itself to be leader

State Machine Replication with Paxos (2/5)

- Initially, the servers elect a leader
- The **first** leader executes phase 1 for **all instances**
  - This is a particular case. We'll see the general case below.
- This can be done using a single very short message
  - Just use the **same proposal number**, e.g. 1, for all instances
- An acceptor will respond with a simple OK message
  - This is a particular case. We'll see the general case below
- After that, as the leader receives clients commands it can run phase 2 for the **next** instance
- A leader may start phase 2 of instance i before it learns the value chosen by instance $i - 1$
  - An implementation may bound the number of **pending** phase 2 instances, i.e. phase 2 instances for which the leader has not learned the chosen value
- A server may execute command i iff:
  1. It learned the command chosen in the i^{th} instance
  2. Has executed **all** commands up to command i

State Machine Replication with Paxos (3/5)

Observation Because of the concurrent execution of multiple phase 2 instances, when a leader fails, the new leader may have not learned the value chosen for some instances:

- It may have learned the command chosen for instance i but not for instance $i - k$, where k is less than the size of the window of the pending phase 2 instances
- It may even be the case that no value has been chosen yet

State Machine Replication with Paxos (4/5)

General Case Initially, a new leader executes phase 1 for **all** instances whose chosen values it has not learned yet

- This can be done using a single PREPARE message, which:
  - specifies all instances for which the new leader has not learned the chosen value
  - specifies the same proposal number for all of them
- An acceptor responds with a single message which
  - includes any accepted proposal for the instances specified in the request
- If no value has been chosen to some of the pending phase 2 instances yet, the new leader may propose a NO-OP operation for those instances

Observation If failure of a leader is a rare event, the cost of executing a state machine command is essentially the cost of executing only phase 2.

- This is the best we can hope for (it has been proven): Paxos is **almost optimal**

State Machine Replication with Paxos (5/5)

What if leader election fails?

No leader is elected no new commands will be proposed

Multiple servers think they are leaders they can all propose values in the same instance of the consensus algorithm

- This may prevent any command from being chosen
- But safety is preserved: two servers will never disagree on the value chosen as the i^{th} state machine command

Conclusion

- Election of a single leader is needed only to ensure progress

Observations

1. Actually, this solution is essentially an implementation of atomic reliable broadcast using consensus (Paxos)
2. Given the assumption of process recovery, this description is somewhat incomplete:
   - The messages sent while a process is down must be delivered in order when the process recovers

Further Reading

- Leslie Lamport, **Paxos Made Simple** in *ACM SIGACT News (Distributed Computing Column)* 32, 4 (December 2001) 51-58.
- T. Chandra, R. Griesemer and J. Redstone, **Paxos Made Live - An Engineerign Perspective**, in *ACM PODC'07*, 2007, 398-407
- R. van Renesse, **Paxos Made Moderately Complex**, 2011
- F. Schneider, **Implementing fault-tolerant services using the state machine approach: A tutorial**, in *ACM Computing Surveys 22, 4 (December 1990), 299–319*