How to implement a write once register?

- The mapping is recorded in "Paxos register" at a set of state machines called acceptors

Note: Though state machines, acceptors are NOT replicas of each other!

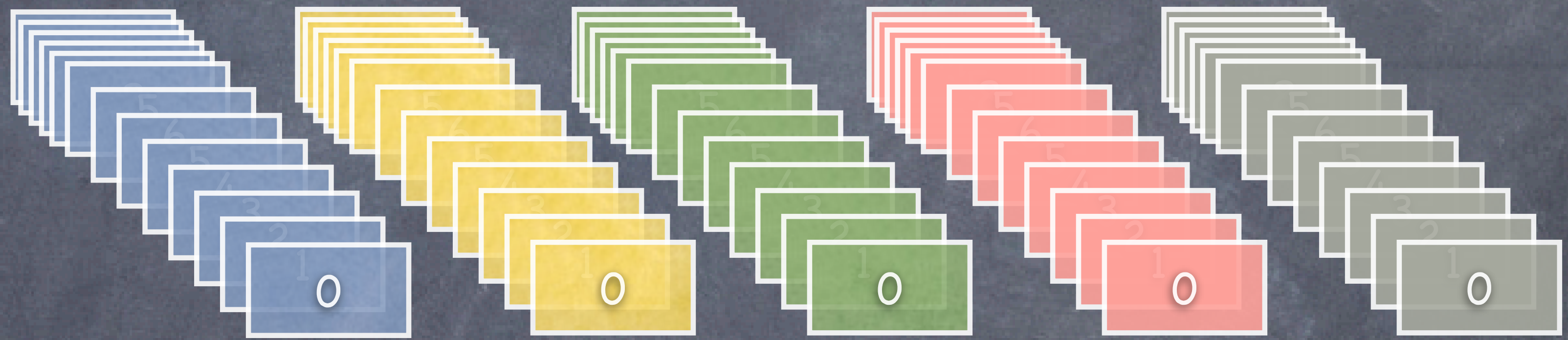
- A leader never proposes a map that may conflict with what is stored in Paxos register

- A leader, before attempting to create a new map between a slot number for which it knows not a decision and a proposal, "reads" the Paxos register to check whether such map may already exist

- Once a leader learns that a new mapping has become permanent, it informs the replicas

Ballots

- Each leader has an infinite supply of ballots



- The set of ballots of different leaders are disjoint

Ballots

- Each leader has an infinite supply of ballots

- The set of ballots of different leaders are disjoint
- Ballots are lexicographically ordered pairs

$$\langle seq\_no, LId \rangle$$

Acceptors

- Send messages only when prompted
- Can crash...
- ...but we assume no more than a minority will
- Need at least $2f+1$ acceptors to tolerate f faults

- Each acceptor α maintains two variables:
  - $\alpha.ballot\_num$, initially $\bot$
  - $\alpha.accepted$, a set of pvalues, initially empty
- A pvalue $e = \langle b, s, p \rangle$ is a triple
  - b: ballot number
  - s: slot number
  - p: a proposal
- α accepts $e \equiv e \in \alpha.accepted$
- α adopts $b \equiv \alpha.ballot\_num := b$

A mapping is forever...

- ...once it is accepted by a majority of acceptors – it is then chosen
- α accepts a pvalue only if it includes the ballot most recently adopted by α
- To make mapping $\langle s, p \rangle$ permanent, λ needs a majority of acceptors to adopt the ballot of the pvalue that contains $\langle s, p \rangle$

```
process Acceptor()
  var ballot_num := ⊥, accepted := ∅;
   for ever
    switch receive();
     case <p1a, λ, b> :
       if b > ballot_num then
         ballot_num := b;
       end if
       send(λ, ⟨p1b, self(), ballot_num, accepted⟩);
     case <p2a, λ, ⟨b, s, p⟩> :
       if b ≥ ballot_num then
         ballot_num := b;
         accepted := accepted ∪ {⟨b, s, p⟩}
       end if
       send(λ, ⟨p2b, self(), ballot_num⟩);
    end switch
   end for
end process
```

Acceptor

- On receiving <p1a, λ, b >
  - adopts b iff larger than $ballot\_num$
  - returns to λ all accepted pvalues (i.e., "partially reads the Paxos register")
- On receiving <p2a, $\lambda, \langle b, s, p\rangle$>
  - adopts b iff larger than $ballot\_num$
  - accepts e if b equal to $ballot\_num$
  - returns to λ the current $ballot\_num$

Invariants

A1. An acceptor can only adopt strictly increasing ballot numbers

A2. An acceptor can only accept $\langle b, s, p\rangle$ if $b = ballot\_num$

A3. An acceptor α can not remove entries from $\alpha.accepted$

```
process Acceptor()
  var ballot_num := ⊥, accepted := ∅;
    for ever
      switch receive();
        case <p1a, λ, b > :
          if b > ballot_num then
            ballot_num := b;
          end if
          send(λ, ⟨ p1b, self(), ballot_num, accepted⟩);
        case <p2a, λ, ⟨b, s, p⟩> :
          if b ≥ ballot_num then
            ballot_num := b;
            accepted:=accepted ∪ {⟨b, s, p⟩}
          end if
          send(λ, ⟨p2b, self(), ballot_num⟩);
      end switch
    end for
end process
```

Acceptor

- On receiving <p1a, λ, b >
  - adopts b iff larger than $ballot\_num$
  - returns to λ all accepted pvalues
- On receiving <p2a, $\lambda, \langle b, s, p\rangle$>
  - adopts b iff larger than $ballot\_num$
  - accepts e if b equal to $ballot\_num$
  - returns to λ the current $ballot\_num$

Invariants

A4. For a given b and s, at most one proposal can be under consideration by the acceptors: $\langle b, s, p\rangle \in \alpha.accepted \wedge \langle b, s, p'\rangle \in \alpha'.accepted \implies p = p'$?

A5. Suppose a majority of acceptors has $\langle b, s, p\rangle \in \alpha.accepted$. If $b' > b$ and $\langle b', s, p'\rangle \in \alpha'.accepted$, then $p = p'$?

Commander

- A leader holding $ballot\_num = b$ and trying to map slot s to proposal p spawns a new commander thread for $\langle b, s, p \rangle$
- A commander's mission has two possible outcomes:
  - success: the leader learns that the proposed mapping has been permanently established
  - failure: the leader learns that b may no longer be acceptable to a majority of acceptors

Commander invariants

C1. For any b and s, at most one commander is spawned

A4. For a given b and s, at most one proposal can be under consideration by the acceptors

Commander invariants

C2. Suppose a majority of acceptors has $\langle b, s, p \rangle \in \alpha.accepted$. If a commander is spawned for $\langle b', s, p' \rangle$: $b' > b$, then $p = p'$?

A5. Suppose a majority of acceptors has $\langle b, s, p \rangle \in \alpha.accepted$. If $b' > b$ and $\langle b', s, p' \rangle \in \alpha'.accepted$, then $p = p'$

Commander

```
process Commander(λ, acceptors, replicas, ⟨b, s, p⟩)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p2a, self(), ⟨b, s, p⟩⟩);
   for ever
     switch receive();
       case <p2b, α, b'> :
         if b' = b then
           waitfor := waitfor − {α};
           if |waitfor| < |acceptors|/2 then
             ∀ρ ∈ replicas :
               send(ρ, ⟨decision, s, p⟩);
             exit();
           end if;
         else
           send(λ, ⟨preempted, b'⟩)
           exit();
         end if
     end switch
   end for
end process
```

Must enforce

R1. For any given slot, replicas decide the same command

A5. Suppose a majority of acceptors has $\langle b, s, p\rangle \in \alpha.accepted$. If $b' > b$ and $\langle b', s, p'\rangle \in \alpha'.accepted$, then $p = p'$

C2. Suppose a majority of acceptors has $\langle b, s, p\rangle \in \alpha.accepted$. If a commander is spawned for $\langle b', s, p'\rangle : b' > b$, then $p = p'$

Commander

```
process Commander(λ, acceptors, replicas, ⟨b, s, p⟩)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p2a, self(), ⟨b, s, p⟩⟩);
   for ever
     switch receive();
        case <p2b, α, b′> :
          if b′ = b then
             waitfor := waitfor − {α};
             if |waitfor| < |acceptors|/2 then
                ∀ρ ∈ replicas :
                   send(ρ, ⟨decision, s, p⟩);
                exit();
             end if;
          else
             send(λ, ⟨preempted, b′⟩)
             exit();
          end if
     end switch
   end for
end process
```

A higher ballot b' is active: a majority of acceptors may no longer be willing to accept b

Commander

process $Commander(\lambda, acceptors, replicas, \langle b, s, p\rangle)$
  var $waitfor := acceptors, pvalues := \emptyset$
  $\forall \alpha \in acceptors : send(\alpha, \langle$ p2a $, self(), \langle b, s, p\rangle\rangle);$
   for ever
    switch $receive();$
      case <p2b $, \alpha, b'$> :
        if $b' = b$ then
          $waitfor := waitfor - \{\alpha\};$
          if $|waitfor| < |acceptors|/2$ then
            $\forall \rho \in replicas :$
              $send(\rho, \langle$ decision $, s, p\rangle);$
            $exit();$
          end if;
        else
          $send(\lambda, \langle$ preempted $, b'\rangle)$
          $exit();$
        end if
    end switch
  end for
end process

Notify the leader and exit

Scout

- Before spawning commanders for ballot b, leader invokes a scout

- Scouts read the Paxos memory to help leaders propose mappings that satisfy C2.

- A scout's mission has two possible outcomes:
  - success: the leader learns that the proposed ballot has been adopted by a majority of acceptors and receives all pvalues accepted by that majority
  - failure: the leader learns that b may no longer be acceptable to a majority of acceptors

```
process Scout(λ, acceptors, b)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p1a, self(), b);
   for ever
    switch receive();
      case <p1b, α, b′, r > :
       if b′ = b then
          pvalues := pvalues ∪ r;
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            send(λ, ⟨adopted, b, pvalues⟩);
           exit();
         end if;
       else
          send(λ, ⟨preempted, b′⟩)
          exit();
       end if
     end switch
   end for
end process
```

Scout

Scout

- gets a majority of acceptors to adopt b
- collects all pvalues that acceptors have accepted while adopting ballots no larger than b

Scout

```
process Scout(λ, acceptors, b)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p1a, self(), ⟨b⟩⟩);
   for ever
     switch receive();
        case <p1b, α, b′, r > :
          if b′ = b then
             pvalues := pvalues ∪ r;
             waitfor := waitfor − {α};
             if |waitfor| < |acceptors|/2 then
                  send(λ, ⟨adopted, b, pvalues⟩);
               exit();
             end if;
          else
            send(λ, ⟨preempted, b′⟩)
            exit();
          end if
     end switch
   end for
end process
```

A higher ballot b' is active: a majority of acceptors may no longer be willing to accept b

Scout

```
process Scout(λ, acceptors, b)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p1a, self(), ⟨b⟩⟩);
   for ever
     switch receive();
        case <p1b, α, b′, r> :
          if b′ = b then
              pvalues := pvalues ∪ r;
              waitfor := waitfor − {α};
              if |waitfor| < |acceptors|/2 then
                  send(λ, ⟨adopted, b, pvalues⟩);
                exit();
              end if;
          else
            send(λ, ⟨preempted, b′⟩)
            exit();
          end if
     end switch
   end for
end process
```

Notify the leader and exit

Leader

- Spawns a scout for initial ballot number
- Enters a loop waiting for one of three messages:
  - $\langle$propose$, s, p\rangle$ from a replica
  - $\langle$adopted$, ballot_num, pvals\rangle$ from a scout
  - $\langle$preempted$, \langle r', \lambda'\rangle\rangle$ from a commander or a scout
- Each leader λ maintains three variables:
  - $\lambda.ballot\_num$, initially 0
  - $\lambda.active$, boolean, initially false
  - $\lambda.proposals$, an initially empty map $\langle slot\_number, proposal\rangle$
- Leader moves between active and passive mode
  - in passive mode is waiting for $\langle$adopted$, ballot_num, pvals\rangle$
  - in active mode spawns commanders for each of the proposal it holds

How a leader enforces

C2. Suppose a majority of acceptors has $\langle b, s, p \rangle \in \alpha.accepted$. If a commander is spawned for $\langle b', s, p' \rangle : b' > b$, then $p = p'$

- Suppose λ learns that a majority of acceptors has adopted its ballot b ($\langle$adopted$, b, pvals \rangle$)
  - CASE 1: if for some slot s there is no value in $pvals$, then it is impossible that a permanent mapping for a smaller ballot already exists or will ever exist for s: any proposal by λ will satisfy C2

How a leader enforces

C2. Suppose a majority of acceptors has $\langle b, s, p \rangle \in \alpha.accepted$. If a commander is spawned for $\langle b', s, p' \rangle : b' > b$, then $p = p'$

- Suppose λ learns that a majority of acceptors has adopted its ballot b ($\langle$adopted$, b, pvals \rangle$)

How a leader enforces

C2. Suppose a majority of acceptors has $\langle b, s, p \rangle \in \alpha.accepted$. If a commander is spawned for $\langle b', s, p' \rangle : b' > b$, then $p = p'$

- Suppose λ learns that a majority of acceptors has adopted its ballot b ($\langle$adopted$, b, pvals\rangle$)
  - CASE 2: let $\langle b', s, p \rangle$ be the pvalue with the maximum ballot number b' for s .
    - by induction, no pvalue other than p could have been chosen for s when $\langle b', s, p \rangle$ was proposed
    - since a majority of acceptors has adopted b, no pvalues between b' and b can be chosen
    - by proposing p with ballot b, λ enforces C2

Leader

process $Leader(acceptors, replicas)$

  var $ballot_num := (0, self()), active = $ false, $proposals := \emptyset$

  $spawn(Scout(self(), acceptors, ballot\_num);$

  for ever

    switch $receive();$

      case <propose, s, p > :

        if $\nexists p' : \langle s, p' \rangle \in proposals$ then

          $proposals := proposals \cup \{\langle s, p \rangle\}$

          if $active$ then

            $spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, p \rangle));$

          end if

        end if

      end case

      case <adopted, $ballot\_num, pvals$ >

        $proposals = proposals \oplus pmax(pvals)$

        $\forall \langle s, p \rangle \in proposals : spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, p \rangle));$

        $active :=$ true

      end case

      case <preempted, r', λ' >

        if $(r', \lambda') > ballot\_num$ then

          $active :=$ false;

          $ballot\_num := (r' + 1, self());$

          $spawn(Scout(self(), acceptors, ballot\_num);$

        end if

$$x \oplus y \equiv \{\langle s, p \rangle \mid \langle s, p \rangle \in y \vee (\langle s, p \rangle \in x \wedge \nexists p' : \langle s, p' \rangle \in y)\}$$

$$pmax(pvals) \equiv \{\langle s, p \rangle \mid \exists b : \langle b, s, p \rangle \in pvals \wedge \forall b', p' : \langle b', s, p' \rangle \in pvals \Rightarrow b' \leq b\}$$

      end case

    end switch

  end for

end process

Implementing State Machine Replication

- Implement a sequence of separate instances of consensus, where the value chosen by the i^{th} instance is the i^{th} message in the sequence.
- Each server assumes all roles in each instance of the algorithm.
- Assume that the set of servers is fixed

The role of the leader

- In normal operation, elect a single server to be a leader. The leader acts as a distinguished proposer in all instances of the consensus algorithm.
  - Clients send commands to the leader, which decides where in the sequence each command should appear.
  - If the leader, for example, decides that a client command is the k^{th} command, it tries to have the command chosen as the value in the k^{th} instance of consensus.

What if a new λ is elected?

- Since λ serves also as a replica in all instances of consensus, it should know most of the commands that have already been chosen. For example, it might know commands for slots 1-10, 13, and 15.
  - It executes phase 1 for slots 11, 12, and 14 and of all slots 16 and larger.
  - λ may find that some value was already accepted for slots 14 and 16 and that slots 11, 12 and all slots after 16 have accepted no command.
  - λ then executes phase 2 of 14 and 16, using the value with the highest ballot it retrieved for those slots

Stop-gap measures

- All replicas now can execute commands 1-10, but not 13-16 because 11 and 12 haven't yet been chosen.
- λ can either take the next two commands it receives by clients to be commands 11 and 12, or can propose immediately that 11 and 12 be no-op commands.
  - this is what happens on "Olive Day"!
- λ runs phase 2 of consensus for slots 11 and 12.
- Once consensus is achieved, all replicas can execute all commands through 16.

To infinity, and beyond

- λ can efficiently execute phase 1 for infinitely many instances of consensus! (e.g. command 16 and higher)
  - λ just sends a message with a sufficiently high proposal number for all instances
  - An acceptor replies non trivially only for instances for which it has already accepted a value

Paxos and FLP

- Paxos is always safe–despite asynchrony
- Once a leader is elected, Paxos is live.
- 👋 "Ciao ciao" FLP?
  - To be live, Paxos requires a single leader
  - "Leader election" is impossible in an asynchronous system (gotcha!)
- Given FLP, Paxos is the next best thing: always safe, and live during periods of synchrony

Atomic Commit

The objective

Preserve data consistency for distributed transactions in the presence of failures

Model

- For each distributed transaction T:
  - one coordinator
  - a set of participants
- Coordinator knows participants; participants don't necessarily know each other
- Each process has access to a Distributed Transaction Log (DT Log) on stable storage

The setup

- Each process p_i has an input value $vote_i$:

$$vote_i \in \{\text{Yes, No}\}$$

- Each process p_i has output value $decision_i$:

$$decision_i \in \{\text{Commit, Abort}\}$$

AC Specification

AC-1: All processes that reach a decision reach the same one.

AC-2: A process cannot reverse its decision after it has reached one.

AC-3: The Commit decision can only be reached if all processes vote Yes.

AC-4: If there are no failures and all processes vote Yes, then the decision will be Commit.

AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide.

Comments

AC-1: All processes that reach a decision reach the same one.

AC-2: A process cannot reverse its decision after it has reached one

AC-3: The Commit decision can only be reached if all processes vote Yes

AC-4: If there are no failures and all processes vote Yes, then the decision will be Commit

AC-5: If all failures are reported and there are no more failures, then all processes will eventually decide

AC1:

- We do not require all processes to reach a decision
- We do not even require all correct processes to reach a decision (impossible to accomplish if links fail)

AC4:

- Avoids triviality
- Allows Abort even if all processes have voted yes

NOTE:

- A process that does not vote Yes can unilaterally abort

Liveness & Uncertainty

- A process is uncertain if it has voted Yes but does not have sufficient information to commit
- While uncertain, a process cannot decide unilaterally
- Uncertainty + communication failures = blocking!

Liveness & Independent Recovery

- Suppose process p fails while running AC.
- If, during recovery, p can reach a decision without communicating with other processes, we say that p can independently recover
- Total failure (i.e. all processes fail) - independent recovery = blocking

A few character-building facts

Proposition 1

If communication failures or total failures are possible, then every AC protocol may cause processes to become blocked

Proposition 2

No AC protocol can guarantee independent recovery of failed processes

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

II. sends $vote_i$ to Coordinator

if $vote_i$ = NO then

$decide_i$:= ABORT

halt

2-Phase Commit

Coordinator c | **Participant p_i**

I. sends VOTE-REQ to all participants

II. sends $vote_i$ to Coordinator
    if $vote_i$ = NO then
        $decide_i$:= ABORT
        halt

III. c votes
    if all vote YES then
    $decide_c$:= COMMIT
    send COMMIT to all
else
    $decide_c$:= ABORT
    send ABORT to all who voted YES
halt

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

II. sends $vote_i$ to Coordinator
  if $vote_i$ = NO then
    $decide_i$:= ABORT
    halt

III. c votes
  if all vote YES then
  $decide_c$:= COMMIT
  send COMMIT to all
else
  $decide_c$:= ABORT
  send ABORT to all who voted YES
halt

IV. if received COMMIT then
    $decide_i$:= COMMIT
  else
    $decide_i$:= ABORT
  halt

Notes on 2PC

- Satisfies AC-1 to AC-4
- But not AC-5 (at least “as is”)
  i. A process may be waiting for a message that may never arrive
     - Use Timeout Actions
  ii. No guarantee that a recovered process will reach a decision consistent with that of other processes
     - Processes save protocol state in DT-Log

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Step 3 Coordinator is waiting for vote from participants

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Since it is has not cast its vote yet, p_i can decide ABORT and halt.

Step 3 Coordinator is waiting for vote from participants

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Since it is has not cast its vote yet, p_i can decide ABORT and halt.

Step 3 Coordinator is waiting for vote from participants

Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Since it is has not cast its vote yet, p_i can decide ABORT and halt.

Step 3 Coordinator is waiting for vote from participants

Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

p_i cannot decide: it must run a termination protocol

Termination protocols

I. Wait for coordinator to recover
   - It always works, since the coordinator is never uncertain
   - may block recovering process unnecessarily

II. Ask other participants

Cooperative Termination

- c appends list of participants to VOTE-REQ
- when an uncertain process p times out, it sends a DECISION-REQ message to every other participant q
- if q has decided, then it sends its decision value to p, which decides accordingly
- if q has not yet voted, then it decides ABORT, and sends ABORT to p
- What if q is uncertain?

Logging actions

1. When c sends VOTE-REQ, it writes START-2PC to its DT Log
2. When p_i is ready to vote YES,
    i. p_i writes YES to DT Log
    ii. p_i sends YES to c (p_i writes also list of participants)
3. When p_i is ready to vote NO, it writes ABORT to DT Log
4. When c is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
5. When c is ready to decide ABORT, it writes ABORT to DT Log
6. After p_i receives decision value, it writes it to DT Log

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT
- otherwise, p is a participant:
  - if DT Log contains a decision value, then decide accordingly
  - else if it does not contain a Yes vote, decide ABORT
  - else (Yes but no decision) run a termination protocol

2PC and blocking

- Blocking occurs whenever the progress of a process depends on the repairing of failures
- No AC protocol is non blocking in the presence of communication or total failures
- But 2PC can block even with non-total failures and no communication failures among operating processes!