

Implementing the Consensus Object with Timing Assumptions

R. Guerraoui

Distributed Computing Laboratory



A modular approach

We implement *Wait-free Consensus (Consensus)*
through:

Lock-free Consensus (L-Consensus)

and

Registers

We implement L-Consensus through

Obstruction-free Consensus (O-Consensus)

and

$\langle \rangle$ *Leader* (encapsulating timing assumptions and
sometimes denoted by Ω)

A modular approach

Consensus

L-Consensus

Registers

0-Consensus

<>Leader

<>Synchrony

Consensus

Wait-Free-Termination: If a correct process proposes, then it eventually decides

Agreement: No two processes decide differently

Validity: Any value decided must have been proposed

L-Consensus

Lock-Free-Termination: If a correct process proposes, then *at least one* correct process eventually decides

Agreement: No two processes decide differently

Validity: Any value decided must have been proposed

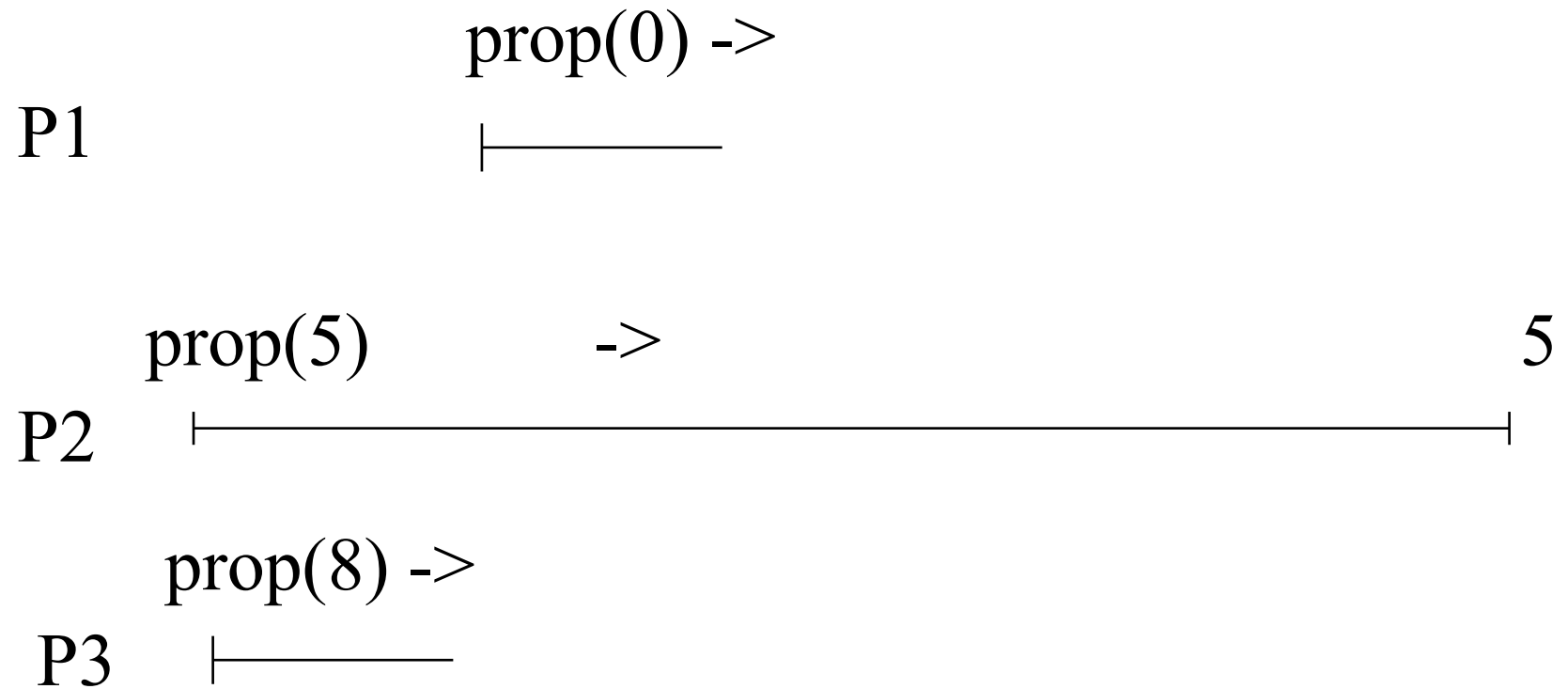
O-Consensus

Obstruction-Free-Termination: If a correct process proposes and *eventually executes alone*, then the process eventually decides

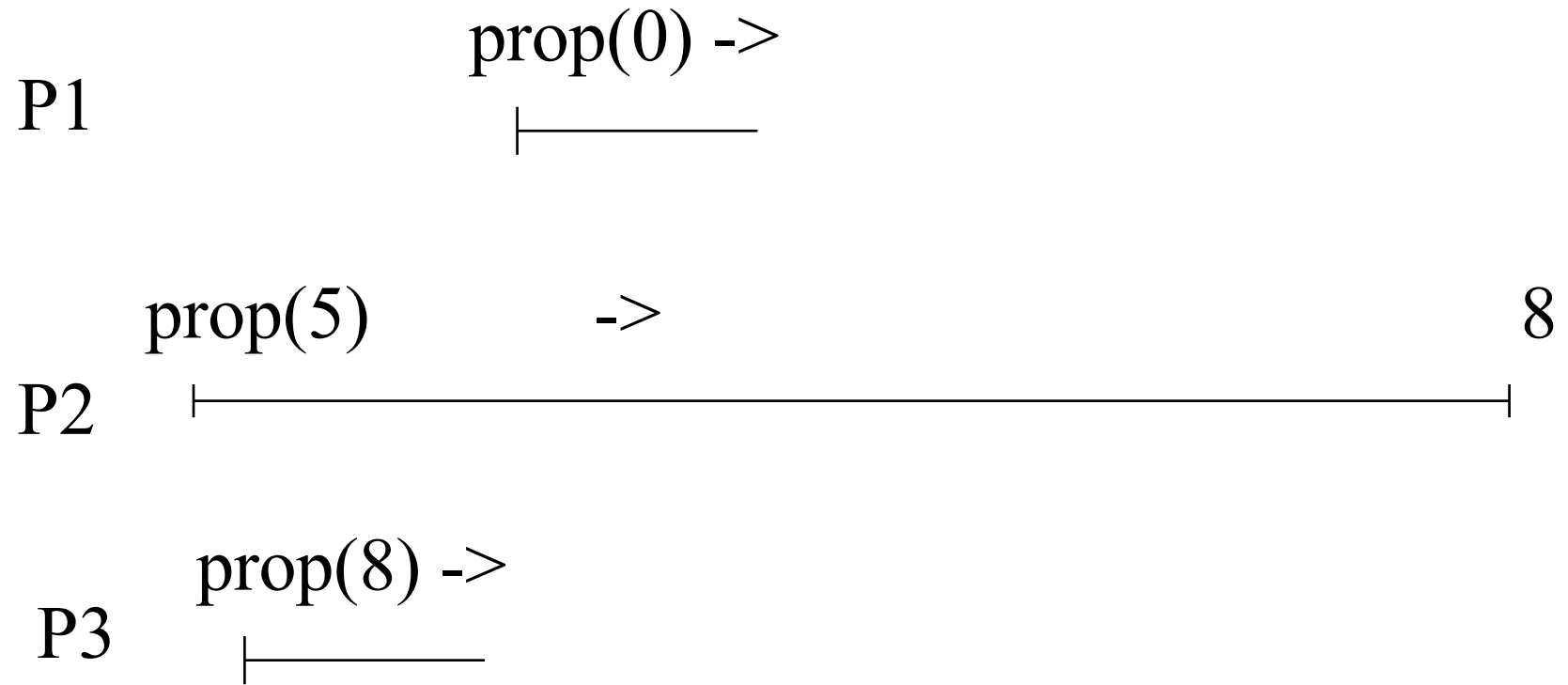
Agreement: No two processes decide differently

Validity: Any value decided must have been proposed

Example 1



Example 2



O-Consensus algorithm (idea)

- A process that is eventually « left alone » to execute steps, eventually decides
- Several processes may keep trying to concurrently decide until some unknown time: agreement and validity should be preserved during this preliminary period

O-Consensus algorithm (data)

- Each process p_i maintains a timestamp ts , initialized to i and incremented by n
- The processes share an array of register pairs ***Reg[1,..,n]***; each element of the array contains two registers:
 - ***Reg[i].T*** contains a timestamp (init to 0)
 - ***Reg[i].V*** contains a pair (value,timestamp) (init to $(\perp, 0)$)

O-Consensus algorithm (functions)

- To simplify the presentation, we assume two functions applied to $\text{Reg}[1, \dots, N]$
 - ***highestTsp()*** returns the highest timestamp among all elements $\text{Reg}[1].T$, $\text{Reg}[2].T$, .., $\text{Reg}[N].T$
 - ***highestTspValue()*** returns the value with the highest timestamp among all elements $\text{Reg}[1].V$, $\text{Reg}[2].V$, .., $\text{Reg}[N].V$

O-Consensus algorithm

- propose(v):
- while(true)
 - Reg[i].T.write(ts);
 - val := Reg[1,..,n].highestTspValue();
 - if val = \perp then val := v;
 - Reg[i].V.write(val,ts);
 - if ts = Reg[1,..,n].highestTsp() then return(val)
 - ts := ts + n

O-Consensus algorithm

- propose(v):
- while(true)
 - ☛ (1) Reg[i].T.write(ts);
 - ☛ (2) val := Reg[1,..,n].highestTspValue();
 - ☛ if val = \perp then val := v;
 - ☛ (3) Reg[i].V.write(val,ts);
 - ☛ (4) if ts = Reg[1,..,n].highestTsp() then return(val)
 - ☛ ts := ts + n

O-Consensus algorithm

- (1) p_i announces its timestamp
- (2) p_i selects the value with the highest timestamp (or its own if there is none)
- (3) p_i announces the value with its timestamp
- (4) if p_i 's timestamp is the highest, then p_i decides (i.e., p_i knows that any process that executes line 2 will select p_i 's value)

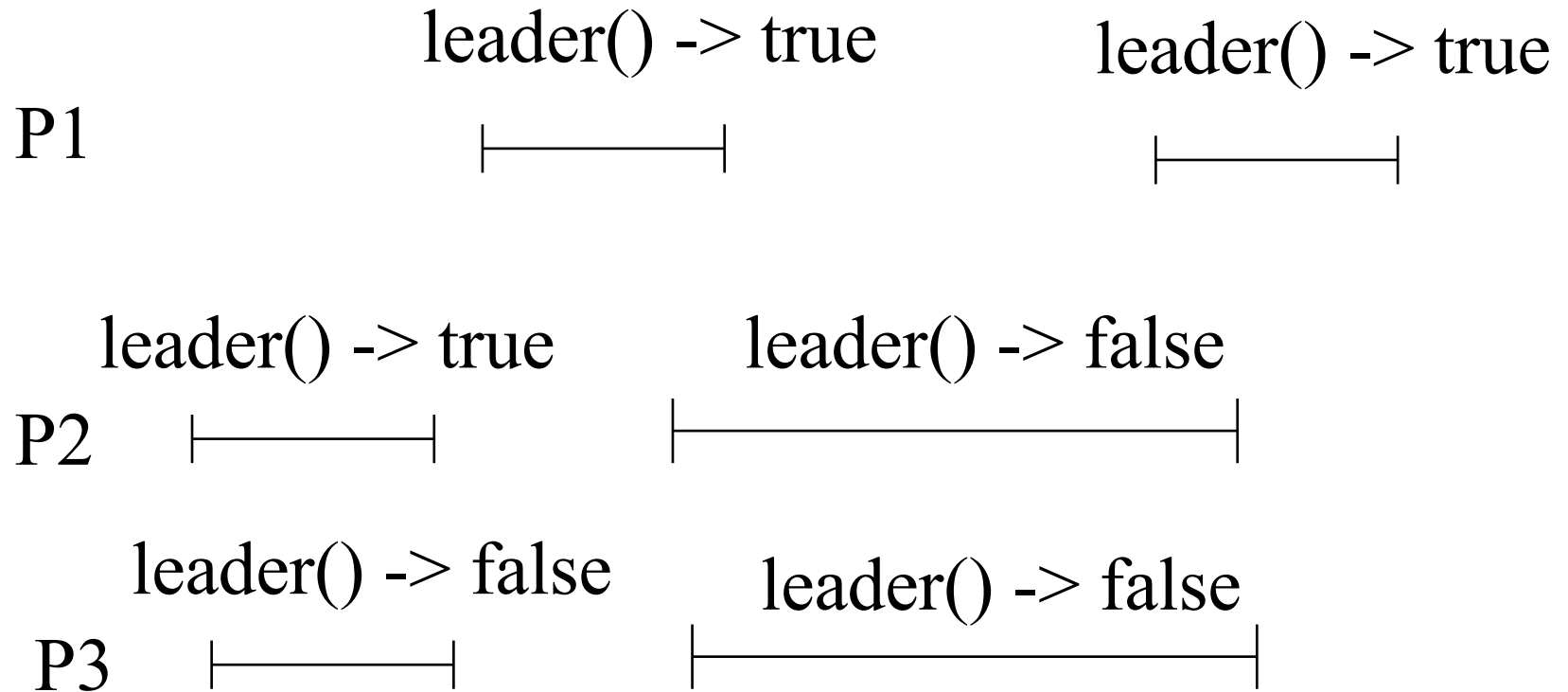
L-Consensus

- We implement L-Consensus using $\langle \rangle$ leader (leader()) and the O-Consensus algorithm
- The idea is to use $\langle \rangle$ leader to make sure that, eventually, one process keeps executing steps alone, until that process decides

<> Leader

- One operation *leader()* which does not take any input parameter and returns a boolean
 - A process considers itself leader if the boolean *leader()* is true
- ✓ **Property:** If a correct process invokes leader, then the invocation returns and *eventually*, some correct process is *permanently* the only leader

Example



L-Consensus

- propose(v): while(true)
 - if leader() then
 - Reg[i].T.write(ts);
 - val := Reg[1,..,n].highestTspValue();
 - if val = \perp then val := v;
 - Reg[i].V.write(val,ts);
 - if ts = Reg[1,..,n].highestTsp() then return(val)
 - ts := ts + n

From L-Consensus to Consensus (helping)

- Every process that decides writes its value in a register ***Dec*** (init to \perp)
- Every process periodically seeks for a value in ***Dec***

Consensus

- ☛ propose(v)
- ☛ while ($\mathbf{Dec.read() = \perp}$)
- ☛ if leader() then
 - ☛ Reg[i].T.write(ts);
 - ☛ val := Reg[1,..,n].highestTspValue();
 - ☛ if val = \perp then val := v ;
 - ☛ Reg[i].V.write(val, ts);
 - ☛ if $ts = \text{Reg}[1,..,n].\text{highestTsp}()$ then Dec.write(val)
 - ☛ $ts := ts + n$;

return($\mathbf{Dec.read()}$)

<> Leader

- One operation *leader()* which does not take any input parameter and returns a boolean
 - A process considers itself leader if the boolean is true
- ✓ **Property:** If a correct process invokes *leader()*, then the invocation returns and *eventually*, some correct process is *permanently* the only leader

<>Leader: algorithm

- We assume that the system is <>synchronous
 - ✓ There is a time after which there is a lower and an upper bound on the delay for a process to execute a local action, a read or a write in shared memory
 - ✓ The time after which the system becomes synchronous is called the global stabilization time (GST) and is unknown to the processes
- This model captures the practical observation that distributed systems are usually synchronous and sometimes asynchronous

<>Leader: algorithm (shared variables)

- Every process p_i elects (stores in a local variable leader) the process with the lowest identity that p_i considers as non-crashed; if p_i elects p_j , then $j < i$
- A process p_i that considers itself leader keeps incrementing ***Reg[i]***, i.e., claiming leadership
- Eventually, only the leader keeps incrementing ***Reg[i]***

<>Leader: algorithm (local variables)

- Every process periodically increments local variables **clock** and **check**, as well as a local variable **delay** whenever its leader changes
- Process p_i maintains **lasti[j]** to record the last value of **Reg[j]** p_i has read (p_i can hence know whether p_j has progressed)
- The next leader is the one with the smallest id that makes some progress; if no such process p_j such that $j < i$ exists, then p_i elects itself (**noLeader** is true)

<>Leader: algorithm (variables)

- *check*, and *delay* are initialized to 1
- *lasti[j]* and *Reg[j]* are initialized to 0

- The next leader is the one with the smallest id that makes some progress; if no such process p_j such that $j < i$ exists, then p_i elects itself (*noLeader* is true)

<>Leader: algorithm

leader(): return(leader=self)

- check, delay and leader init to 1
- lasti[j] and Reg[j] init to 0;

- Task:
- clock := 0;
- while(true) do
 - ✓ if (leader=self) then
 - ✓ Reg[i].write(Reg[i].read()+1);
 - ✓ clock := clock + 1;
 - ✓ if (clock = check) then
 - ✓ elect();

<>Leader: algorithm (cont'd)

elect():

- noLeader := true;
- for j = 1 to (i-1) do
 - ✓ if (Reg[j].read() > last[j]) then
 - ✓ last[j] := Reg[j].read();
 - ✓ if (leader ≠ pj) then delay:=delay*2;
 - ✓ check := check + delay;
 - ✓ leader:= pj;
 - ✓ noLeader := false; break (for);
- if (noLeader) then leader := self;

Consensus = Registers + $\langle \rangle$ Leader

- $\langle \rangle$ Leader has one operation *leader()* which does not take any input parameter and returns a boolean; a process considers itself leader if the boolean is true
 - ✓ **Property:** If a correct process invokes leader, then the invocation returns and *eventually*, some correct process is *permanently* the only leader
- $\langle \rangle$ Leader encapsulates the following synchrony assumption: there is a time after which a lower and an upper bound hold on the time it takes for every process to execute a step (eventual synchrony)

Minimal Assumptions

- Consensus is impossible in an asynchronous system with Registers (FLP83, LA88)
- Consensus is possible in an eventually synchronous system (i.e., $\langle \rangle$ Leader) with Registers (DLS88, LH95)