Implementing the Consensus Object with Timing Assumptions

R. Guerraoui
Distributed Programming Laboratory

A modular approach

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

Lock-free Consensus (L-Consensus)

and

Registers

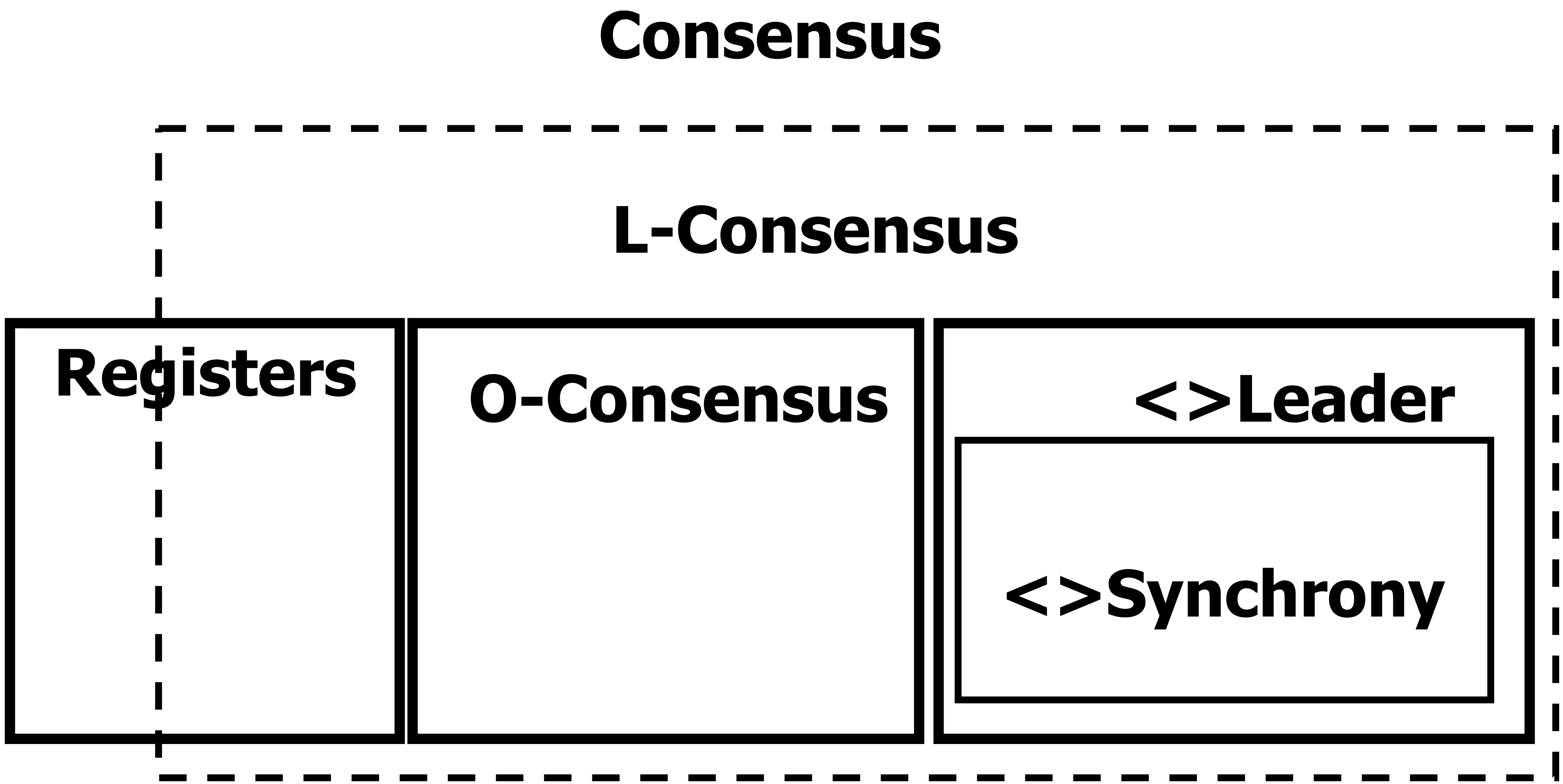
We implement L-Consensus through

Obstruction-free Consensus (O-Consensus)

and

<>*Leader* (encapsulating timing assumptions and sometimes denoted by ***Ω***)

A modular approach



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

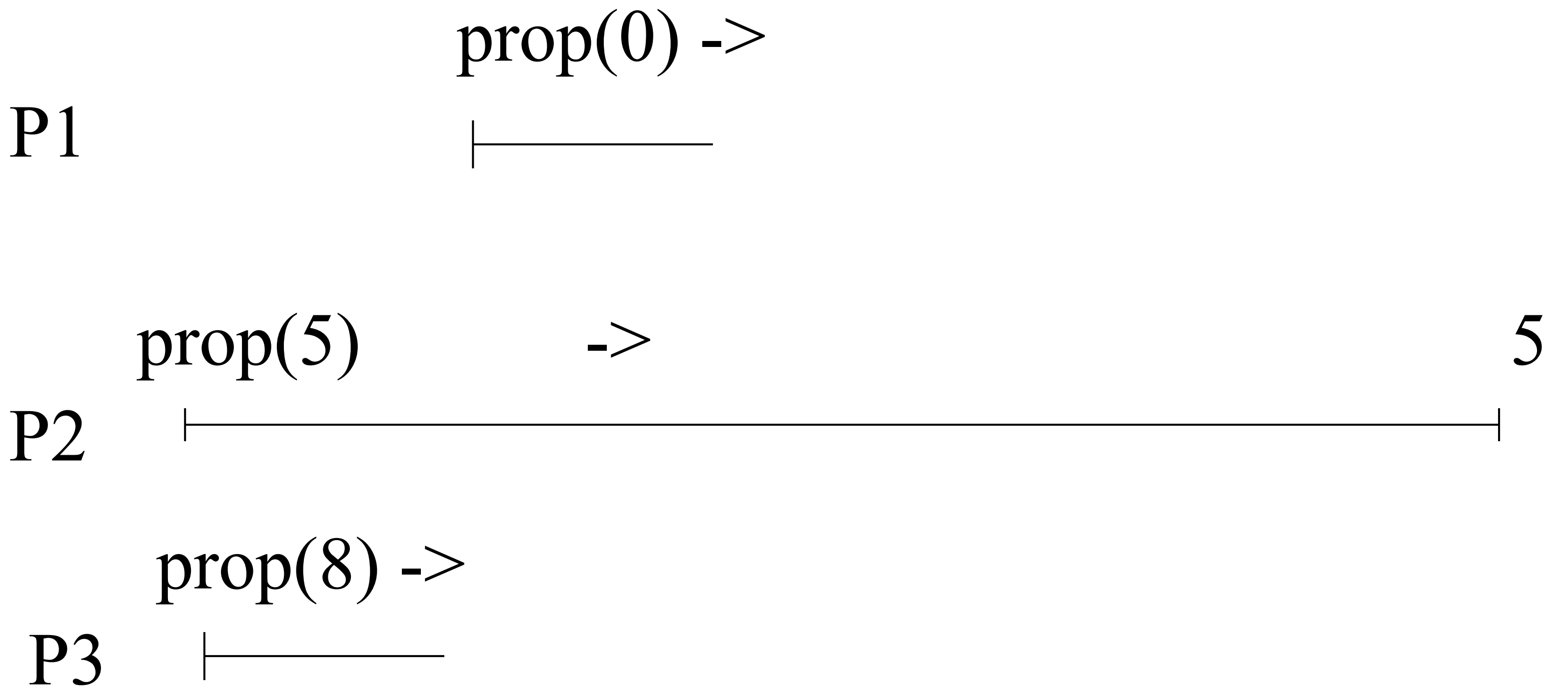
O-Consensus

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

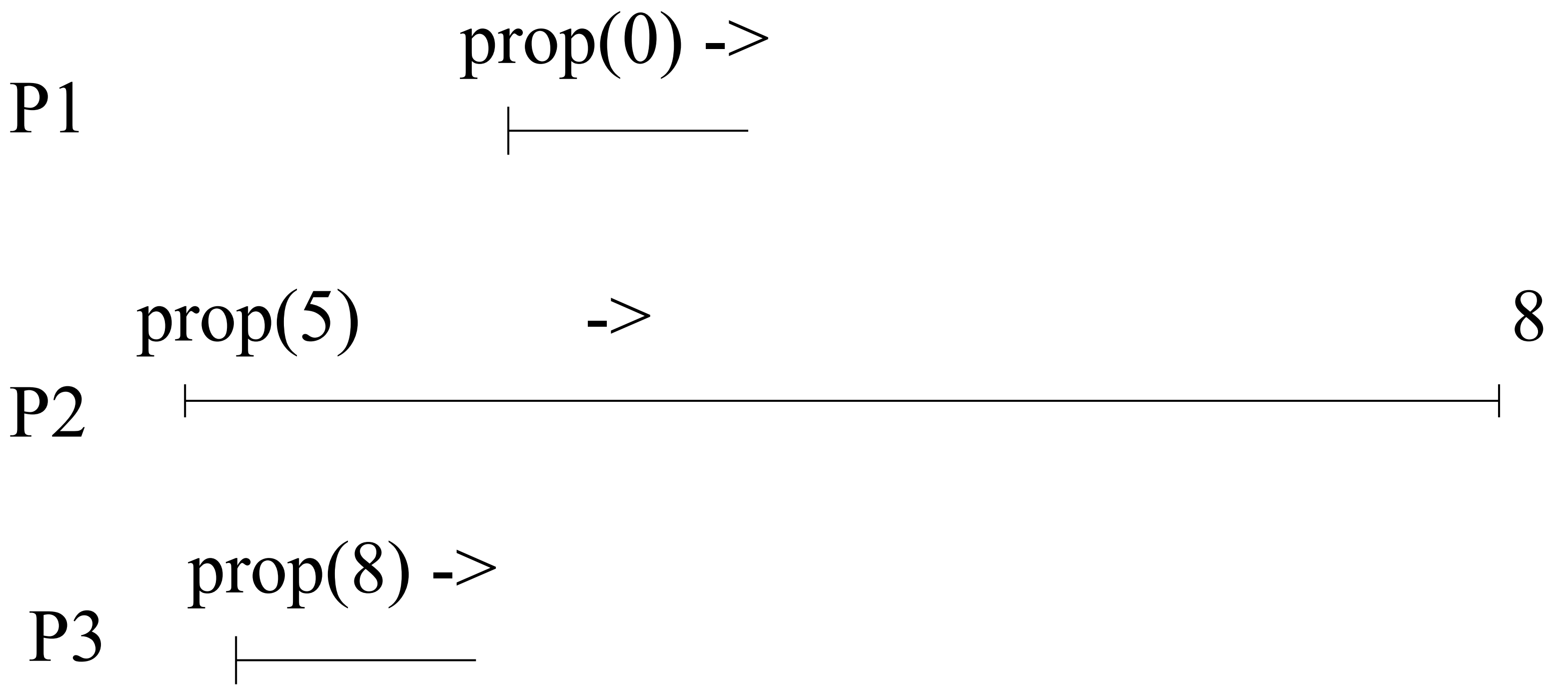
Agreement: No two processes decide differently

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 might keep trying to concurrently decide until some unknown time: agreement (and validity) should be ensured during this preliminary period

O-Consensus algorithm (data)

- Each process pi 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 (⊥,0))

O-Consensus algorithm (functions)

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

O-Consensus algorithm

- propose(v):
- while(true)
  - Reg[i].T.write(ts);
  - val := Reg[1,..,n].highestTspValue();
  - if val = ⊥ 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 = ⊥ 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) pi announces its timestamp
- (2) pi selects the value with the highest timestamp (or its own if there is none)
- (3) pi announces the value with its timestamp
- (4) if pi’s timestamp is the highest, then pi decides (i.e., pi knows that any process that executes line 2 will select pi’s value)

L-Consensus

- We implement L-Consensus using <>leader (leader()) and the O-Consensus algorithm

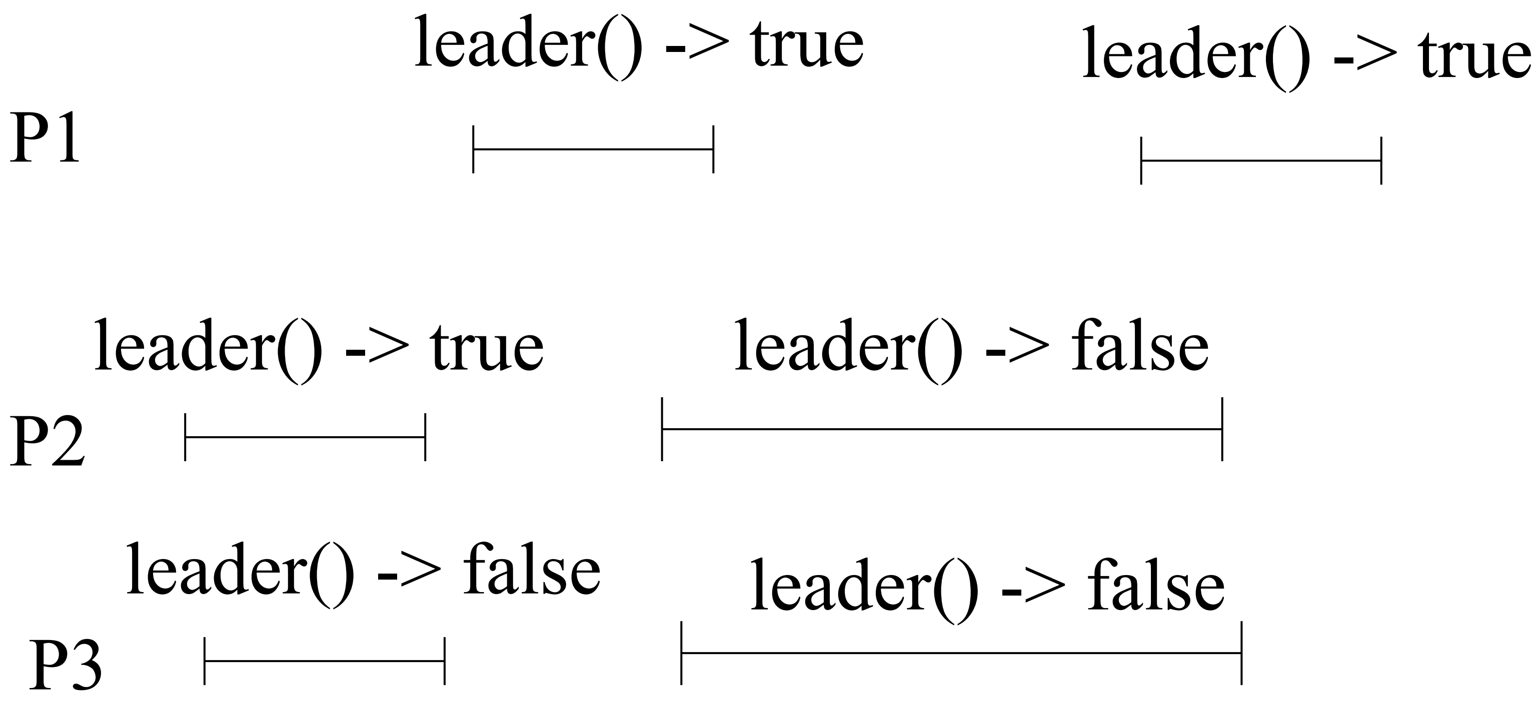
- The idea is to use <>leader to make sure that, eventually, one process keeps executing steps alone, until it decides

<> Leader

- One operation ***leader()*** which does not take any input parameter and returns, as an output parameter, 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

Example


leader() -> true
leader() -> true
P1
leader() -> true
leader() -> false
P2
leader() -> false
leader() -> false
P3

L-Consensus

- propose(v): while(true)
  - if leader() then
    - Reg[i].T.write(ts);
    - val := Reg[1,..,n].highestTspValue();
    - if val = ⊥ 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 (***Dec***.read() = ⊥)
- if leader() then
  - Reg[i].T.write(ts);
  - val := Reg[1,..,n].highestTspValue();
  - if val = ⊥ then val := p;
  - Reg[i].V.write(val,ts);
  - if ts = Reg[1,..,n].highestTsp()
  -             then Dec.write(val)
  - ts := ts + n;

return(***Dec***.read())

<> Leader

- One operation ***leader()*** which does not take any input parameter and returns, as an output parameter, 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 pi elects (stores in a local variable leader) the process with the lowest identity that pi considers as non-crashed; if pi elects pj, then $j < i$

- A process pi that considers itself leader keeps incrementing ***Reg[i]*** ; pi claims that it wants to remain leader

- NB. Eventually, only the leader keeps incrementing the shared register ***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 pi maintains ***lasti[j]*** to record the last value of ***Reg[j]*** pi has read (pi can hence know whether pj has progressed)
- The next leader is the one with the smallest id that makes some progress; if no such process pj such that $j<i$ exists, then pi 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 pj such that j<i exists, then pi elects itself (***noLeader*** is true)

<>Leader: algorithm

leader(): return(leader)

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

- Task:
- while(true) do
  - ✓ clock := 0;
  - ✓ 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 + <> Leader

- <>Leader has one operation ***leader()*** which does not take any input parameter and returns, as an output parameter, 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 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., <> Leader) with Registers (DLS88, LH95)
- What is the minimal synchrony assumption needed to implement Consensus with Registers?
- Is there any weaker timing abstraction than <>Leader that help Registers solve Consensus

Failure detector

- A ***failure detector*** is a distributed (wait-free) oracle that provides processes with information about the ***crashes*** of processes
- Examples: ***P, ◊P, ◊S, ◊W,*** Ω***,*** ◊**`Leader`**
- NB. A failure detector does ***only*** provide information about crashes (CT96)

Failure detector relations

- We say that a failure detector D ***implements*** abstraction A (e.g., object O) if there is an algorithm that implements A using D
- We say that a failure detector D is ***weaker*** than a failure detector D' if D' ***implements*** D (D ≤ D')
- If D is weaker than D' and D' is not weaker than D, then D is said to be ***strictly weaker*** than D' (D < D')
- We say that two failure detectors are ***equivalent*** if each is weaker than the other (D ≅ D')

Failure detector Ω

- Failure detector Ω outputs a process q at every process p (we say that p ***trusts*** q) and ensures the following property:
  - Eventually, the ***same correct*** process is ***permanently trusted*** by every process
  - NB. Note that the process that is trusted might keep changing until some eventual time

<>Leader ≅ Ω

- To implement <>Leader using Ω, every process simply returns true if it is leader (the process emulates the output of <>Leader)

- To implement <>Leader using Ω, every process writes its name in a shared register ***L*** when leader() returns true; all processes periodically read L and elect the process in ***L*** (eventually, only one process is elected)

Failure detector example

- Failure detector Ω outputs a process q at every process p (we say that p ***trusts*** q) and ensures the following property:
  - ◇ ***unique leader:*** eventually, the ***same correct*** process is ***permanently trusted*** by every process
  - NB. Note that the process that is trusted might keep changing until some eventual time

Questions

- (1) Show that Ω is the weakest failure detector to implement consensus with Registers (i.e., give an algorithm that implements Ω with any failure detector that implements Consensus with Registers)
- (2) What is the weakest failure detector to implement Consensus with objects of consensus number k and Registers?
- (3) What is the weakest failure to implement an object with consensus number k using Registers?