The Limitations of Registers (cont'd)

Julien Stainer

Concurrent Algorithms
Distributed Programming Laboratory
julien.stainer@epfl.ch

Outline

1. Consensus from Stacks and Registers
2. Immediate Snapshots
3. The Iterated Immediate Snapshot Model
4. Set Agreement in $\mathcal{IIS}$
5. k-Set Agreement from Registers

1 Consensus from Stacks and Registers
Problem Statement
The Case of Two Processes
The Case of Three Processes

2 Immediate Snapshots

3 The Iterated Immediate Snapshot Model

4 Set Agreement in $\mathcal{IIS}$

5 k-Set Agreement from Registers

Is it possible to wait-free implement a consensus object from stacks and registers in a system of 2 processes?

A consensus object offers an operation PROPOSE(v) that returns a value. It fulfills the following properties:

Termination: Any invocation of PROPOSE by a correct process terminates.

A consensus object offers an operation PROPOSE(v) that returns a value. It fulfills the following properties:

Termination Any invocation of PROPOSE by a correct process terminates.

Agreement At most one value is decided.

A consensus object offers an operation PROPOSE(v) that returns a value. It fulfills the following properties:

- **Termination** Any invocation of PROPOSE by a correct process terminates.
- **Agreement** At most one value is decided.
- **Validity** A decided value is a proposed value.

Two Processes Consensus from a Stack and Registers

```
initialization
    REG[0] ← ⊥; REG[1] ← ⊥
    S.push(loser); S.push(winner)
operation PROPOSE(v)
    REG[id] ← v
    if S.pop() = winner then
        return v
    else
        return REG[1 − id]
```

The Case of Three Processes

- With 3 processes, the losers cannot easily know which value to adopt.

The Case of Three Processes

- With 3 processes, the losers cannot easily know which value to adopt.
- Even with several stacks and more registers, how to organize?

Is it possible to wait-free implement a consensus object from stacks and registers in a system of 3 processes?

Structure of the Proof

- Suppose that there exists an algorithm solving 3 processes consensus from stacks and registers.

- Suppose that there exists an algorithm solving 3 processes consensus from stacks and registers.
- Show that there is a schedule in which a process takes an infinite number of steps but does not decide.

Structure of the Proof

- Suppose that there exists an algorithm solving 3 processes consensus from stacks and registers.
- Show that there is a schedule in which a process takes an infinite number of steps but does not decide.
- This contradicts the termination property. Consequently, there is no such algorithm.

Lemma 1

The initial configuration $C(0, 1, 0)$ is bivalent.

Lemma 1

The initial configuration $C(0,1,0)$ is bivalent.

- Starting from $C(0,1,0)$, if p_1 executes alone, it has to decide 0 because it cannot distinguish between this execution and the one starting from $C(0,0,0)$ where it executes alone.

Lemma 1

The initial configuration $C(0,1,0)$ is bivalent.

- Starting from $C(0,1,0)$, if p_1 executes alone, it has to decide 0 because it cannot distinguish between this execution and the one starting from $C(0,0,0)$ where it executes alone.
- Starting from $C(0,1,0)$, if p_2 executes alone, it has to decide 1 because it cannot distinguish between this execution and the one starting by $C(1,1,1)$ where it executes alone.

Bivalent Initial Configuration

Lemma 1

The initial configuration $C(0, 1, 0)$ is bivalent.

- Starting from $C(0, 1, 0)$, if p_1 executes alone, it has to decide 0 because it cannot distinguish between this execution and the one starting from $C(0, 0, 0)$ where it executes alone.
- Starting from $C(0, 1, 0)$, if p_2 executes alone, it has to decide 1 because it cannot distinguish between this execution and the one starting by $C(1, 1, 1)$ where it executes alone.
- Consequently, $C(0, 1, 0)$ is bivalent.

Maximal Schedule Leading to a Bivalent Configuration

- Consider a schedule Σ such that

Maximal Schedule Leading to a Bivalent Configuration

- Consider a schedule Σ such that
  - $\Sigma(C(0,1,0))$ is bivalent;

Maximal Schedule Leading to a Bivalent Configuration

- Consider a schedule Σ such that
  - $\Sigma(C(0,1,0))$ is bivalent;
  - $\forall i \in \{1,2,3\} \ : \ p_i(\Sigma(C(0,1,0)))$ is monovalent.

Maximal Schedule Leading to a Bivalent Configuration

- Consider a schedule Σ such that
  - $\Sigma(C(0,1,0))$ is bivalent;
  - $\forall i \in \{1,2,3\}\ :\ p_i(\Sigma(C(0,1,0)))$ is monovalent.
- Necessarily, there are two processes p_i and p_j such that $p_i(\Sigma(C(0,1,0)))$ is 0-valent while $p_j(\Sigma(C(0,1,0)))$ is 1-valent.

Maximal Schedule Leading to a Bivalent Configuration

- Consider a schedule Σ such that
  - $\Sigma(C(0,1,0))$ is bivalent;
  - $\forall i \in \{1,2,3\} \; : \; p_i(\Sigma(C(0,1,0)))$ is monovalent.
- Necessarily, there are two processes p_i and p_j such that $p_i(\Sigma(C(0,1,0)))$ is 0-valent while $p_j(\Sigma(C(0,1,0)))$ is 1-valent.
- Let op_i (resp. op_j) be the next step executed by p_i (resp. p_j) in $\Sigma(C(0,1,0))$.

Maximal Schedule Leading to a Bivalent Configuration

- Consider a schedule Σ such that
  - $\Sigma(C(0,1,0))$ is bivalent;
  - $\forall i \in \{1,2,3\}$: $p_i(\Sigma(C(0,1,0)))$ is monovalent.
- Necessarily, there are two processes p_i and p_j such that $p_i(\Sigma(C(0,1,0)))$ is 0-valent while $p_j(\Sigma(C(0,1,0)))$ is 1-valent.
- Let op_i (resp. op_j) be the next step executed by p_i (resp. p_j) in $\Sigma(C(0,1,0))$.
- If op_i and op_j commute, then processes cannot distinguish between $p_i(p_j(\Sigma(C(0,1,0))))$ and $p_j(p_i(\Sigma(C(0,1,0))))$ while one is 0-valent and the other is 1-valent. This is a contradiction.

Possible Values for op_i and op_j

- Since op_i and op_j do not commute, they are both invocations of operations on the same stack or register.

Possible Values for op_i and op_j

- Since op_i and op_j do not commute, they are both invocations of operations on the same stack or register.
- Two reads on the same register commute, so if op_i and op_j are both accesses to the same register, at least one of them is a write.

Possible Values for op_i and op_j

- Since op_i and op_j do not commute, they are both invocations of operations on the same stack or register.
- Two reads on the same register commute, so if op_i and op_j are both accesses to the same register, at least one of them is a write.
- If they are both writes to the same register, then p_i cannot distinguish between $p_i(\Sigma(C(0, 1, 0)))$ and $p_i(p_j(\Sigma(C(0, 1, 0))))$, while one is 0-valent and the other 1-valent. This is a contradiction.

Possible Values for op_i and op_j

- If op_i is a write to a register and op_j a read to the same register, then p_i cannot distinguish between $p_i(p_j(\Sigma(C(0,1,0))))$ and $p_i(\Sigma(C(0,1,0)))$ while the former is 1-valent and the latter is 0-valent. This is a contradiction.

Possible Values for op_i and op_j

- If op_i is a write to a register and op_j a read to the same register, then p_i cannot distinguish between $p_i(p_j(\Sigma(C(0,1,0))))$ and $p_i(\Sigma(C(0,1,0)))$ while the former is 1-valent and the latter is 0-valent. This is a contradiction.
- Symmetric arguments apply when inverting the valence of $p_i(\Sigma(C(0,1,0)))$ and $p_j(\Sigma(C(0,1,0)))$ or when op_i and op_j are respectively a read and a write to the same register.

- If op_i is a write to a register and op_j a read to the same register, then p_i cannot distinguish between $p_i(p_j(\Sigma(C(0,1,0))))$ and $p_i(\Sigma(C(0,1,0)))$ while the former is 1-valent and the latter is 0-valent. This is a contradiction.
- Symmetric arguments apply when inverting the valence of $p_i(\Sigma(C(0,1,0)))$ and $p_j(\Sigma(C(0,1,0)))$ or when op_i and op_j are respectively a read and a write to the same register.
- It follows that op_i and op_j are necessarily invocations of operations on the same stack.

Possible Values for op_i and op_j

- If both op_i and op_j are pop operations on the same stack, then $p_k, k \in \{1,2,3\} \setminus \{i,j\}$ cannot distinguish between $p_i(p_j(\Sigma(C(0,1,0))))$ and $p_j(p_i(\Sigma(C(0,1,0))))$ while one is 0-valent and the other 1-valent. This is a contradiction.

- If both op_i and op_j are pop operations on the same stack, then $p_k, k \in \{1,2,3\} \setminus \{i,j\}$ cannot distinguish between $p_i(p_j(\Sigma(C(0,1,0))))$ and $p_j(p_i(\Sigma(C(0,1,0))))$ while one is 0-valent and the other 1-valent. This is a contradiction.
- If op_i is a push operation and op_j a pop operation on the same stack, then we have two cases:

Possible Values for op_i and op_j

- If both op_i and op_j are pop operations on the same stack, then $p_k, k \in \{1, 2, 3\} \setminus \{i, j\}$ cannot distinguish between $p_i(p_j(\Sigma(C(0, 1, 0))))$ and $p_j(p_i(\Sigma(C(0, 1, 0))))$ while one is 0-valent and the other 1-valent. This is a contradiction.
- If op_i is a push operation and op_j a pop operation on the same stack, then we have two cases:
    - If in $\Sigma(C(0, 1, 0))$ the stack is empty, then p_k cannot distinguish between $p_j(p_i(\Sigma(C(0, 1, 0))))$ and $p_j(\Sigma(C(0, 1, 0))))$ while the former is 0-valent and the latter is 1-valent. This is a contradiction.

Possible Values for op_i and op_j

- If both op_i and op_j are pop operations on the same stack, then $p_k, k \in \{1, 2, 3\} \setminus \{i, j\}$ cannot distinguish between $p_i(p_j(\Sigma(C(0, 1, 0))))$ and $p_j(p_i(\Sigma(C(0, 1, 0))))$ while one is 0-valent and the other 1-valent. This is a contradiction.
- If op_i is a push operation and op_j a pop operation on the same stack, then we have two cases:
  - If in $\Sigma(C(0, 1, 0))$ the stack is empty, then p_k cannot distinguish between $p_j(p_i(\Sigma(C(0, 1, 0))))$ and $p_j(\Sigma(C(0, 1, 0))))$ while the former is 0-valent and the latter is 1-valent. This is a contradiction.
  - If in $\Sigma(C(0, 1, 0))$ the stack is not empty, then we need a further analysis.

Possible Values for op_i and op_j

In this case op_i is a push operation and op_j a pop operation on the same stack that is not empty in $\Sigma(C(0,1,0))$.

- If p_i runs alone from $p_j(p_i(\Sigma(C(0,1,0))))$, it necessarily eventually pops the item z that was on top of the stack in $\Sigma(C(0,1,0))$ or it would not distinguish between the situation when it runs alone from $p_i(p_j(\Sigma(C(0,1,0))))$ while it has to decide 0 in the first situation and 1 in the second.

In this case op_i is a push operation and op_j a pop operation on the same stack that is not empty in $\Sigma(C(0,1,0))$.

- If p_i runs alone from $p_j(p_i(\Sigma(C(0,1,0))))$, it necessarily eventually pops the item z that was on top of the stack in $\Sigma(C(0,1,0))$ or it would not distinguish between the situation when it runs alone from $p_i(p_j(\Sigma(C(0,1,0))))$ while it has to decide 0 in the first situation and 1 in the second.
- Let Σ_i be the schedule in which p_i executes solo from $p_j(p_i(\Sigma(C(0,1,0))))$ until just after it pops the item z.

Possible Values for op_i and op_j

In this case op_i is a push operation and op_j a pop operation on the same stack that is not empty in $\Sigma(C(0,1,0))$.

- If p_i runs alone from $p_j(p_i(\Sigma(C(0,1,0))))$, it necessarily eventually pops the item z that was on top of the stack in $\Sigma(C(0,1,0))$ or it would not distinguish between the situation when it runs alone from $p_i(p_j(\Sigma(C(0,1,0))))$ while it has to decide 0 in the first situation and 1 in the second.
- Let Σ_i be the schedule in which p_i executes solo from $p_j(p_i(\Sigma(C(0,1,0))))$ until just after it pops the item z.
- Since until this pop operation p_i cannot distinguish if it started from $p_j(p_i(\Sigma(C(0,1,0))))$ or $p_i(p_j(\Sigma(C(0,1,0))))$, it also takes the same steps while running alone from the later configuration. The only difference is the value it pops at the last step of Σ_i.

Possible Values for op_i and op_j

- p_k cannot distinguish between $\Sigma_i(p_j(p_i(\Sigma(C(0,1,0)))))$ and $\Sigma_i(p_i(p_j(\Sigma(C(0,1,0)))))$ because the stack is in the same state in both configurations. The former configuration being 0-valent while the latter is 1-valent, this is a contradiction.

- p_k cannot distinguish between $\Sigma_i(p_j(p_i(\Sigma(C(0,1,0)))))$ and $\Sigma_i(p_i(p_j(\Sigma(C(0,1,0)))))$ because the stack is in the same state in both configurations. The former configuration being 0-valent while the latter is 1-valent, this is a contradiction.
- The same reasoning applies if the roles of p_i and p_j swapped.

- If op_i and op_j are both push operations on the same stack, then, when running alone from $p_i(p_j(\Sigma(C(0,1,0))))$, p_i necessarily eventually pops the item it pushed at op_i or it would not be able to distinguish this execution from the one when it runs alone from $p_j(p_i(\Sigma(C(0,1,0))))$.

- If op_i and op_j are both push operations on the same stack, then, when running alone from $p_i(p_j(\Sigma(C(0,1,0))))$, p_i necessarily eventually pops the item it pushed at op_i or it would not be able to distinguish this execution from the one when it runs alone from $p_j(p_i(\Sigma(C(0,1,0))))$.
- Let Σ'_i be the schedule in which p_i executes alone from $p_i(p_j(\Sigma(C(0,1,0))))$ until just after it pops the value pushed by op_i.

Possible Values for op_i and op_j

- If op_i and op_j are both push operations on the same stack, then, when running alone from $p_i(p_j(\Sigma(C(0,1,0))))$, p_i necessarily eventually pops the item it pushed at op_i or it would not be able to distinguish this execution from the one when it runs alone from $p_j(p_i(\Sigma(C(0,1,0))))$.
- Let Σ'_i be the schedule in which p_i executes alone from $p_i(p_j(\Sigma(C(0,1,0))))$ until just after it pops the value pushed by op_i.
- With the same reasoning, starting from $\Sigma'_i(p_i(p_j(\Sigma(C(0,1,0)))))$ or from $\Sigma'_i(p_j(p_i(\Sigma(C(0,1,0)))))$, p_j necessarily take the same steps until it eventually pops the value pushed by op_j (in the first situation) or by op_i (in the second one). Let us denote Σ'_j its steps until just after this pop.

- p_k is not able to distinguish between $\Sigma'_j(\Sigma'_i(p_i(p_j(\Sigma(C(0,1,0))))))$ and $\Sigma'_j(\Sigma'_i(p_i(p_j(\Sigma(C(0,1,0))))))$. This is a contradiction because the former is 1-valent while the latter is 0-valent.

- p_k is not able to distinguish between $\Sigma'_j(\Sigma'_i(p_i(p_j(\Sigma(C(0,1,0))))))$ and $\Sigma'_j(\Sigma'_i(p_i(p_j(\Sigma(C(0,1,0))))))$. This is a contradiction because the former is 1-valent while the latter is 0-valent.
- In all cases we reach a contradiction. It follows that there exists a schedule such that a process takes an infinite number of steps without deciding, which concludes the proof.

It is impossible to wait-free implement consensus among 3 processes from stacks and registers.

1 Consensus from Stacks and Registers

2 Immediate Snapshots
- Immediate Snapshot Specification
- Set-Linearizability
- Immediate Snapshot Algorithm

3 The Iterated Immediate Snapshot Model

4 Set Agreement in $\mathcal{IIS}$

5 k-Set Agreement from Registers

Immediate Snapshot Specification

An immediate snapshot object offers an operation $\textsc{write-snapshot}(v)$ that can be invoked at most once by each process. It returns a set *view* of pairs (j, v_j) where j is a process identifier and v_j a value. If we denote by $view_i$ the set returned to process i, we have the following properties:

Termination
: Any invocation of $\textsc{write-snapshot}$ by a correct process terminates.

Immediate Snapshot Specification

An immediate snapshot object offers an operation WRITE-SNAPSHOT(v) that can be invoked at most once by each process. It returns a set *view* of pairs (j, v_j) where j is a process identifier and v_j a value. If we denote by $view_i$ the set returned to process i, we have the following properties:

Termination Any invocation of WRITE-SNAPSHOT by a correct process terminates.

Validity If $(j, v_j) \in view_i$, then process j invoked WRITE-SNAPSHOT(v_j).

An immediate snapshot object offers an operation WRITE-SNAPSHOT(v) that can be invoked at most once by each process. It returns a set *view* of pairs (j, v_j) where j is a process identifier and v_j a value. If we denote by $view_i$ the set returned to process i, we have the following properties:

Termination Any invocation of WRITE-SNAPSHOT by a correct process terminates.

Validity If $(j, v_j) \in view_i$, then process j invoked WRITE-SNAPSHOT(v_j).

Self-Inclusion $(id, v_{id}) \in view_{id}$.

An immediate snapshot object offers an operation $\textsc{write-snapshot}(v)$ that can be invoked at most once by each process. It returns a set *view* of pairs (j, v_j) where j is a process identifier and v_j a value. If we denote by $view_i$ the set returned to process i, we have the following properties:

Termination Any invocation of $\textsc{write-snapshot}$ by a correct process terminates.

Validity If $(j, v_j) \in view_i$, then process j invoked $\textsc{write-snapshot}(v_j)$.

Self-Inclusion $(id, v_{id}) \in view_{id}$.

Containment $\forall i, j \; : \; view_i \subseteq view_j \vee view_j \subseteq view_i$.

An immediate snapshot object offers an operation WRITE-SNAPSHOT(v) that can be invoked at most once by each process. It returns a set *view* of pairs (j, v_j) where j is a process identifier and v_j a value. If we denote by $view_i$ the set returned to process i, we have the following properties:

- **Termination** Any invocation of WRITE-SNAPSHOT by a correct process terminates.
- **Validity** If $(j, v_j) \in view_i$, then process j invoked WRITE-SNAPSHOT(v_j).
- **Self-Inclusion** $(id, v_{id}) \in view_{id}$.
- **Containment** $\forall i, j \;:\; view_i \subseteq view_j \vee view_j \subseteq view_i$.
- **Immediacy** $\forall i, j \;:\; (j, v_j) \in view_i \implies view_j \subseteq view_i$.

Theorem

$$((i, v_i) \in view_j \wedge (j, v_j) \in view_i) \implies view_i = view_j$$

Theorem

$$((i, v_i) \in view_j \wedge (j, v_j) \in view_i) \implies view_i = view_j$$

Consequence

The calls to an immediate snapshot object can be set-linearized by ordering the processes according to the size of their views.

One by one:
$view_1 = \{(1, v_1)\} \subsetneq view_2 = \{(1, v_1), (2, v_2)\} \subsetneq view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$

Two then one:
$view_1 = view_2 = \{(1, v_1), (2, v_2)\} \subsetneq view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$

Three together:
$view_1 = view_2 = view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$

Immediate Snapshot Algorithm

```
initialization
    REG[1, ..., n][1, ..., n] ← [[⊥, ..., ⊥], ..., [⊥, ..., ⊥]]
operation WRITE-SNAPSHOT(v)
    return REC_WRITE-SNAPSHOT(n, v)
operation REC_WRITE-SNAPSHOT(x,v)
    REG[x][id] ← v
    for i ∈ {1, ..., n} do scan[j] ← REG[x][j] end for
    view ← {(j, scan[j]) | scan[j] ≠ ⊥}
    if |view| = x then
        return view
    else
        return REC_WRITE-SNAPSHOT(x − 1, v)
```

Table of Contents

1 Consensus from Stacks and Registers

2 Immediate Snapshots

3 The Iterated Immediate Snapshot Model
- Computation Model
- Equivalence Theorem

4 Set Agreement in $\mathcal{IIS}$

5 k-Set Agreement from Registers

The Iterated Immediate Snapshot Model

- Processes execute a sequence of asynchronous rounds.

The Iterated Immediate Snapshot Model

- Processes execute a sequence of asynchronous rounds.
- During each round, a process that has not crashed invokes WRITE-SNAPSHOT(s) to write its current state in the immediate snapshot object $IS[r]$ associated to the round, and to collect the states of other processes.

- Processes execute a sequence of asynchronous rounds.
- During each round, a process that has not crashed invokes WRITE-SNAPSHOT(s) to write its current state in the immediate snapshot object $IS[r]$ associated to the round, and to collect the states of other processes.
- It then updates its state to include the knowledge it has gained on the state of other processes and proceeds to the next round.

The Iterated Immediate Snapshot Model

- Processes execute a sequence of asynchronous rounds.
- During each round, a process that has not crashed invokes $\textsc{write-snapshot}(s)$ to write its current state in the immediate snapshot object $IS[r]$ associated to the round, and to collect the states of other processes.
- It then updates its state to include the knowledge it has gained on the state of other processes and proceeds to the next round.
- After a predetermined number of rounds R, a process that does not crash decides a value by applying a deterministic function $\textsc{decide}$ of its final state.

The Iterated Immediate Snapshot Model

```
initialization
    s ← {⟨0, input of the process⟩}
    r ← 1
while r ≤ R do
    view ← IS[r].WRITE-SNAPSHOT(s)
    s ← s ∪ {⟨r, view⟩}
    r ← r + 1
DECIDE(s)
```

The Read/Write Wait-free Model vs. $\mathcal{IIS}$

- As shown before, $\mathcal{IIS}$ can be simulated in the read/write wait-free model.

The Read/Write Wait-free Model vs. $\mathcal{IIS}$

- As shown before, $\mathcal{IIS}$ can be simulated in the read/write wait-free model.
- Any one-shot colorless task that can be solved in the read/write wait-free model can be solved in $\mathcal{IIS}$.

The Read/Write Wait-free Model vs. $\mathcal{IIS}$

- As shown before, $\mathcal{IIS}$ can be simulated in the read/write wait-free model.
- Any one-shot colorless task that can be solved in the read/write wait-free model can be solved in $\mathcal{IIS}$.
- One-shot tasks: processes decide and it stops (e.g. consensus), as opposed to long-lived objects like stacks or queues that keep a separate state in shared memory.

The Read/Write Wait-free Model vs. $\mathcal{IIS}$

- As shown before, $\mathcal{IIS}$ can be simulated in the read/write wait-free model.
- Any one-shot colorless task that can be solved in the read/write wait-free model can be solved in $\mathcal{IIS}$.
- One-shot tasks: processes decide and it stops (e.g. consensus), as opposed to long-lived objects like stacks or queues that keep a separate state in shared memory.
- Colorless tasks: in any execution, if a process decides, its decision value can be adopted by any other process as its own (e.g. consensus, k-set agreement but not renaming).

The Read/Write Wait-free Model vs. $\mathcal{IIS}$

- As shown before, $\mathcal{IIS}$ can be simulated in the read/write wait-free model.
- Any one-shot colorless task that can be solved in the read/write wait-free model can be solved in $\mathcal{IIS}$.
- One-shot tasks: processes decide and it stops (e.g. consensus), as opposed to long-lived objects like stacks or queues that keep a separate state in shared memory.
- Colorless tasks: in any execution, if a process decides, its decision value can be adopted by any other process as its own (e.g. consensus, k-set agreement but not renaming).

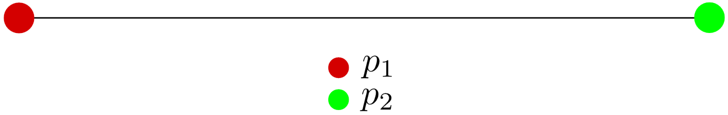
A one-shot colorless task can be solved in the read/write wait-free model iff it can be solved in $\mathcal{IIS}$.

Table of Contents

1. Consensus from Stacks and Registers
2. Immediate Snapshots
3. The Iterated Immediate Snapshot Model
4. Set Agreement in $\mathcal{IIS}$
5. k-Set Agreement from Registers

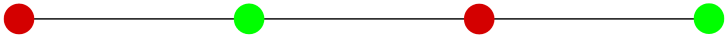
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.



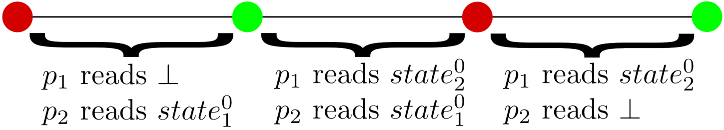
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.



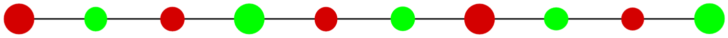
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.



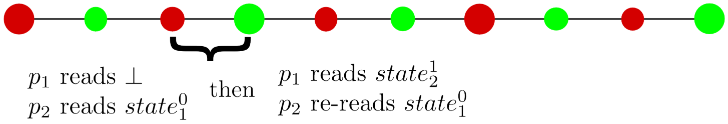
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.



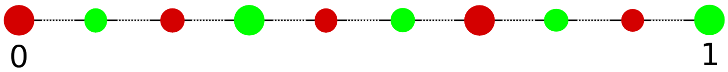
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.



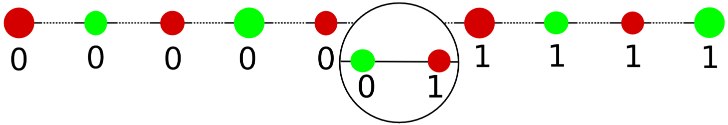
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.
- The processes have to decide in a finite number of rounds R, the subdivision is consequently finite.



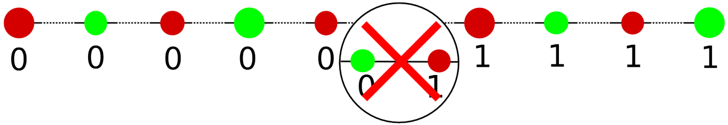
Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.
- The processes have to decide in a finite number of rounds R, the subdivision is consequently finite.
- The states can be tagged with the corresponding decided values.



Solving Consensus is impossible in $\mathcal{IIS}$ with Two Processes

- The possible executions of an algorithm in $\mathcal{IIS}$ between two processes can be seen as a subdivision of the initial configuration.
- The processes have to decide in a finite number of rounds R, the subdivision is consequently finite.
- The states can be tagged with the corresponding decided values.
- Impossibility result comes from Sperner's Lemma.



A k-set agreement object offers an operation PROPOSE(v) that returns a value. It fulfills the following properties:

Termination Any invocation of PROPOSE by a correct process terminates.

A k-set agreement object offers an operation $\textsc{propose}(v)$ that returns a value. It fulfills the following properties:

- **Termination** Any invocation of PROPOSE by a correct process terminates.
- **Agreement** At most k different values are decided in the system.

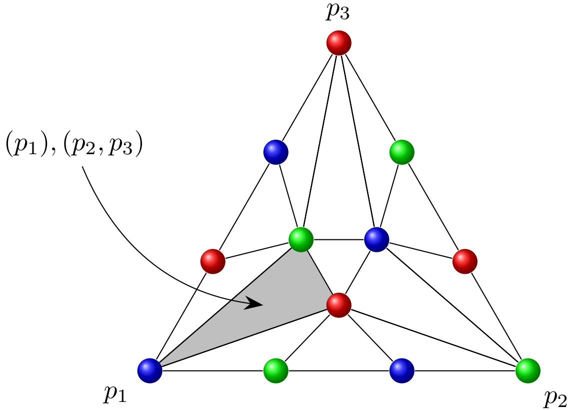
The k-Set Agreement Problem

A k-set agreement object offers an operation $\text{PROPOSE}(v)$ that returns a value. It fulfills the following properties:

- **Termination** Any invocation of PROPOSE by a correct process terminates.
- **Agreement** At most k different values are decided in the system.
- **Validity** All decided values are proposed values.

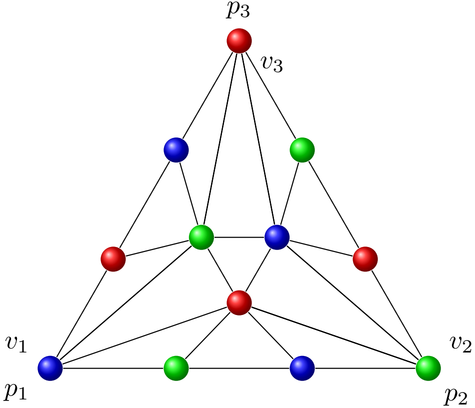
2-Set Agreement from Registers Among 3 Processes

The possible executions of an algorithm in $\mathcal{IIS}$ between three processes can be seen as a subdivision of the initial configuration.



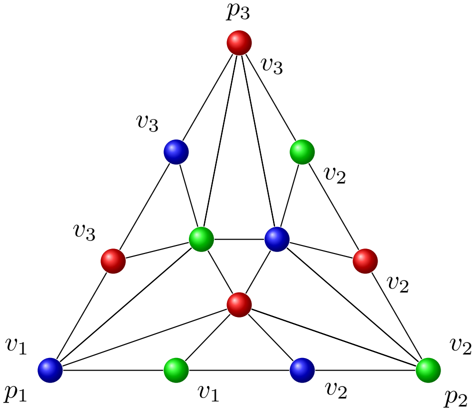
2-Set Agreement from Registers Among 3 Processes

If a process runs alone, it has to decide its input.



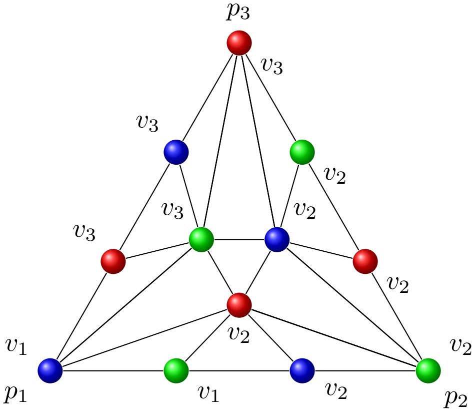
2-Set Agreement from Registers Among 3 Processes

If two processes run without seeing the third one, they have to decide on one of their two values.



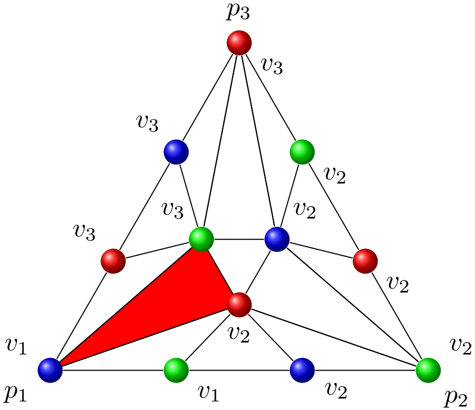
2-Set Agreement from Registers Among 3 Processes

By Sperner's Lemma, any completion of this type of coloring...



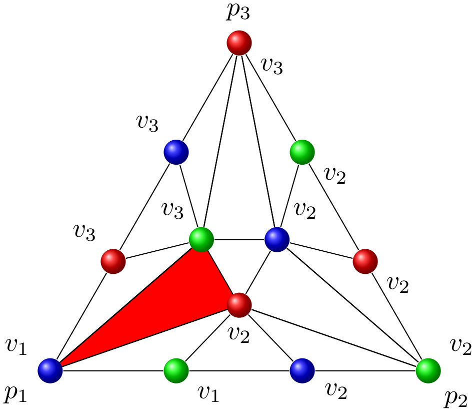
2-Set Agreement from Registers Among 3 Processes

By Sperner's Lemma, any completion of this type of coloring...
has at least one configuration where processes decide on 3 different values.



2-Set Agreement from Registers Among 3 Processes

2-set agreement is consequently impossible in one round of $\mathcal{IIS}$ between 3 processes, but the same argument applies for any finite number R of rounds.



k-Set Agreement from Registers Among $k + 1$ Processes

- Using the same principles with $k + 1$ processes:

k-Set Agreement from Registers Among $k + 1$ Processes

- Using the same principles with $k + 1$ processes:
  - starting from an initial configuration where $k + 1$ different values are proposed;

k-Set Agreement from Registers Among $k+1$ Processes

- Using the same principles with $k+1$ processes:
  - starting from an initial configuration where $k+1$ different values are proposed;
  - the set of possible configurations after R iterations of $\mathcal{IIS}$ is a triangulation of the initial (k-dimensional) configuration;

k-Set Agreement from Registers Among $k + 1$ Processes

- Using the same principles with $k + 1$ processes:
  - starting from an initial configuration where $k + 1$ different values are proposed;
  - the set of possible configurations after R iterations of $\mathcal{IIS}$ is a triangulation of the initial (k-dimensional) configuration;
  - the possible final states have to be associated with decision values;

k-Set Agreement from Registers Among $k+1$ Processes

- Using the same principles with $k+1$ processes:
  - starting from an initial configuration where $k+1$ different values are proposed;
  - the set of possible configurations after R iterations of $\mathcal{IIS}$ is a triangulation of the initial (k-dimensional) configuration;
  - the possible final states have to be associated with decision values;
  - the processes can only decide on values they have seen, these constrains impose that their decision in their final state forms a Sperner's coloring of the possible final states;

k-Set Agreement from Registers Among $k+1$ Processes

- Using the same principles with $k+1$ processes:
  - starting from an initial configuration where $k+1$ different values are proposed;
  - the set of possible configurations after R iterations of $\mathcal{IIS}$ is a triangulation of the initial (k-dimensional) configuration;
  - the possible final states have to be associated with decision values;
  - the processes can only decide on values they have seen, these constrains impose that their decision in their final state forms a Sperner's coloring of the possible final states;
  - Sperner's Lemma states that their is at least one final configuration that has the $k+1$ colors.

k-Set Agreement from Registers Among $k+1$ Processes

- Using the same principles with $k+1$ processes:
  - starting from an initial configuration where $k+1$ different values are proposed;
  - the set of possible configurations after R iterations of $\mathcal{IIS}$ is a triangulation of the initial (k-dimensional) configuration;
  - the possible final states have to be associated with decision values;
  - the processes can only decide on values they have seen, these constrains impose that their decision in their final state forms a Sperner's coloring of the possible final states;
  - Sperner's Lemma states that their is at least one final configuration that has the $k+1$ colors.
- Consequently, k-set agreement cannot be wait-free implemented in $\mathcal{IIS}$ in a system of $k+1$ processes.

k-Set Agreement from Registers Among $k+1$ Processes

- Using the same principles with $k+1$ processes:
  - starting from an initial configuration where $k+1$ different values are proposed;
  - the set of possible configurations after R iterations of $\mathcal{IIS}$ is a triangulation of the initial (k-dimensional) configuration;
  - the possible final states have to be associated with decision values;
  - the processes can only decide on values they have seen, these constrains impose that their decision in their final state forms a Sperner's coloring of the possible final states;
  - Sperner's Lemma states that their is at least one final configuration that has the $k+1$ colors.
- Consequently, k-set agreement cannot be wait-free implemented in $\mathcal{IIS}$ in a system of $k+1$ processes.
- It follows that k-set agreement cannot be wait-free implemented from registers in a system of $k+1$ processes.

Table of Contents

1. Consensus from Stacks and Registers
2. Immediate Snapshots
3. The Iterated Immediate Snapshot Model
4. Set Agreement in $\mathcal{IIS}$
5. k-Set Agreement from Registers

BG-Simulation

Any one-shot colorless task that we can solve with n processes and t crashes, we can solve it with $t+1$ processes and t crashes.

BG-Simulation

Any one-shot colorless task that we can solve with n processes and t crashes, we can solve it with $t+1$ processes and t crashes.

BG-Simulation

The study of decision tasks computability can be reduced to the $n-1$-resilient case.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.
- We can then build a k-resilient algorithm for $k + 1$ processes/simulators.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.
- We can then build a k-resilient algorithm for $k + 1$ processes/simulators.
  - Use the BG-simulation with n simulators to simulate $k + 1$ processes.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.
- We can then build a k-resilient algorithm for $k + 1$ processes/simulators.
    - Use the BG-simulation with n simulators to simulate $k + 1$ processes.
    - Simulate the protocol.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.
- We can then build a k-resilient algorithm for $k + 1$ processes/simulators.
    - Use the BG-simulation with n simulators to simulate $k + 1$ processes.
    - Simulate the protocol.
    - Decide any value decided by a simulated process.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k+1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.
- We can then build a k-resilient algorithm for $k+1$ processes/simulators.
  - Use the BG-simulation with n simulators to simulate $k+1$ processes.
  - Simulate the protocol.
  - Decide any value decided by a simulated process.
- This solves k-set agreement between our $k+1$ simulators.

k-Set Agreement is Impossible with k crashes or more.

- k-set agreement is impossible to solve among $k + 1$ processes with k crashes.
- For any $n > k$, suppose we have a k-resilient algorithm for k-set agreement.
- We can then build a k-resilient algorithm for $k + 1$ processes/simulators.
    - Use the BG-simulation with n simulators to simulate $k + 1$ processes.
    - Simulate the protocol.
    - Decide any value decided by a simulated process.
- This solves k-set agreement between our $k + 1$ simulators.
- Contradiction, so there is no such algorithm.

- Implementing a consensus object from stacks and registers is possible for 2 processes but not for 3.

- Implementing a consensus object from stacks and registers is possible for 2 processes but not for 3.
- The Iterated Immediate Snapshot model and the read/write wait-free model can compute the same tasks.

- Implementing a consensus object from stacks and registers is possible for 2 processes but not for 3.
- The Iterated Immediate Snapshot model and the read/write wait-free model can compute the same tasks.
- The possible configurations after R rounds of $\mathcal{IIS}$ have a regular geometric structure.

- Implementing a consensus object from stacks and registers is possible for 2 processes but not for 3.
- The Iterated Immediate Snapshot model and the read/write wait-free model can compute the same tasks.
- The possible configurations after R rounds of $\mathcal{IIS}$ have a regular geometric structure.
- Sperner's Lemma allows to prove that k-set agreement is impossible in a system of $k+1$ processes.

Wrap-Up

- Implementing a consensus object from stacks and registers is possible for 2 processes but not for 3.
- The Iterated Immediate Snapshot model and the read/write wait-free model can compute the same tasks.
- The possible configurations after R rounds of $\mathcal{IIS}$ have a regular geometric structure.
- Sperner's Lemma allows to prove that k-set agreement is impossible in a system of $k+1$ processes.
- BG-simulation allows to simulate larger systems while preserving the number of crashes.

Wrap-Up

- Implementing a consensus object from stacks and registers is possible for 2 processes but not for 3.
- The Iterated Immediate Snapshot model and the read/write wait-free model can compute the same tasks.
- The possible configurations after R rounds of $\mathcal{IIS}$ have a regular geometric structure.
- Sperner's Lemma allows to prove that k-set agreement is impossible in a system of $k+1$ processes.
- BG-simulation allows to simulate larger systems while preserving the number of crashes.
- Combining the two results, we show that k-set agreement is impossible from registers in a system prone to k crashes or more.

- Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993.
- Borowsky E. and Gafni E., A simple algorithmically reasoned characterization of wait-free computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, pp. 189-198, 1997.
- Herlihy M. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923, 1999.
- Herlihy M.P., Kozlov D. N. and Rajsbaum S., *Distributed computing through combinatorial topology.* Morgan Kaufmann, 2014 (ISBN 978-0-12-404578-1).