# Impossibility Results for Distributed Computing Part I

Faith Ellen
Department of Computer Science
University of Toronto

# Distributed System

A collection of processes that can communicate by:

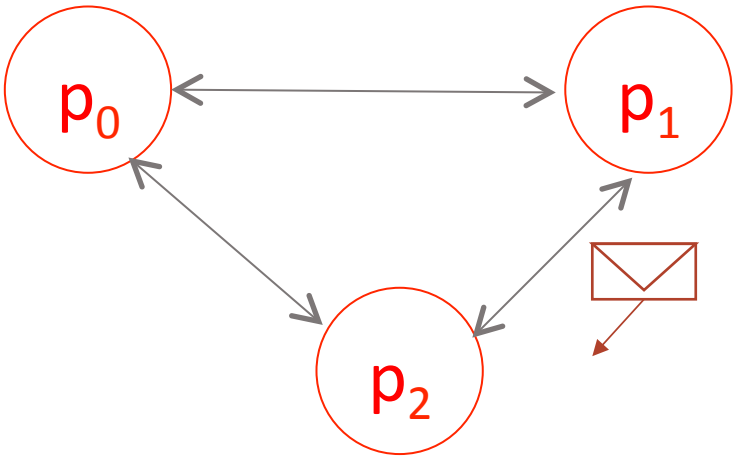
- sending and receiving messages between one another

or

- performing operations on shared objects.

# Synchronous Message-Passing Models

- An execution is divided into rounds.
- Each round, each process takes one step, in which it receives messages from other processes, does local computation, and sends messages to other processes.
- Messages sent in one round are received in the next round.



A configuration describes the system between consecutive rounds. It consists of:

- the state of every process at the end of the preceding round and
- the messages that are in transit.

An initial configuration describes the system at the beginning of an execution.

- Each process is in an initial state, which includes its input.
- There are no messages in transit.

# Asynchronous Message-Passing Models

Processes take steps in an arbitrary order, determined by an adversarial scheduler.

An execution is a sequence of steps.

In each step, either

- one message is delivered or
- one process receives messages that have been delivered to it from other processes, does local computation, and sends messages to other processes.

Messages can take arbitrarily long to be delivered.

A configuration describes the system between consecutive steps of an execution. It consists of:

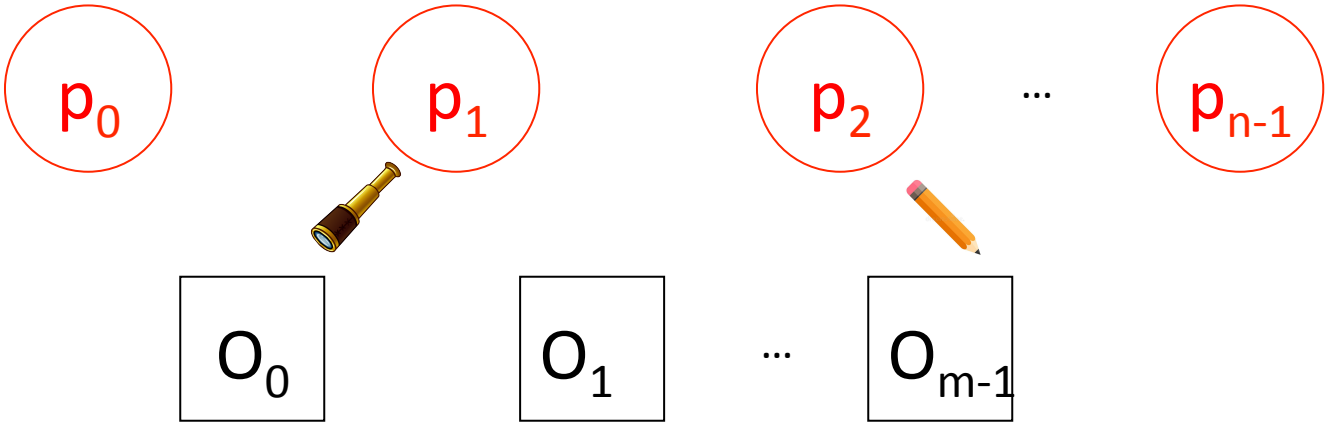
- the state of every process,
- the messages that each process has sent, but have not yet been delivered, and
- the messages that have been delivered to each process, which it has not yet received.

An initial configuration describes the system at the beginning of an execution.

- Each process is in an initial state, which includes its input.
- There are no messages in transit.

# Asynchronous Shared-Memory Models

- Processes take steps in an arbitrary order, determined by an adversarial scheduler.
- An execution is a sequence of steps.
- In each step, one process performs an operation on a shared object and then does local computation.



Each object has:

- a set of possible values,
- an initial value, and
- a set of operations that can be applied to it.

A process atomically performs an operation on an object. The process updates its state depending on its current state and the response from the object.

# Register

Set of possible values: {0,1,2,…}
Initial value: 0

READ(R): Returns the value of register R.
Doesn't change the value of R.

WRITE(R,v): Sets the value of R to v.
Returns ack.

# Compare&Swap Object

Set of possible values: {⊥,0,1,2,…}
Initial value: ⊥

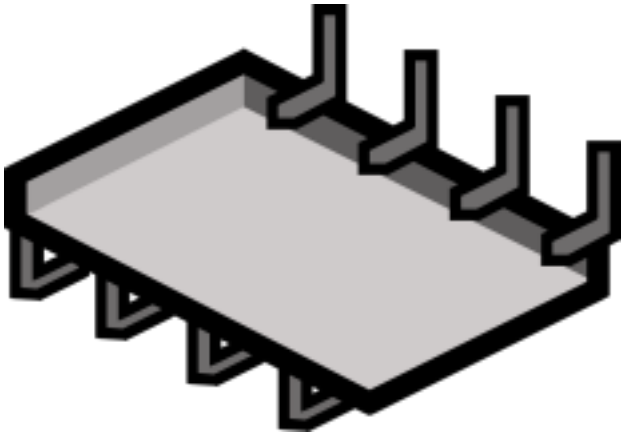
CAS(R,u,v): returns the value of R.
If the value of R is u, changes the value of R from u to v.

A configuration describes the system between consecutive steps of an execution. It consists of:

- the state of every process and
- the value of every shared object.

An initial configuration describes the system at the beginning of an execution.

- Each process is in an initial state, which includes its input.
- Each shared object has a predetermined initial value.



## Crash Faults

A faulty process stops taking steps before it finishes its task.

## Byzantine Faults

A faulty process deviates from its protocol.

# Distributed vs. Parallel Computing

- Each process may have its own task.
- Each process has its own inputs.
- More processes make the tasks harder.
- Fault tolerance.

- One task is shared by all processes.
- All inputs are available to all processes.
- More processes make the task easier.

# Central Questions in Theory of Distributed Computing

- What problems can be solved in a given distributed model?
- How efficiently can a particular problem be solved?
- What makes certain problems hard to solve?
- How do parameters of a model affect its computational power?
- What are the relationships between the computational power of different models?

# Most impossibility results in distributed computing rely on lack of knowledge about the system.

At any point in time, the state of a process, including the values of its input and local variables, describes the knowledge the process has about the system.

To solve many distributed problems, processes need to learn information about the states of other processes.

Unsolvability proofs show that this knowledge cannot be obtained.

Lower bound proofs show that this knowledge cannot be obtained with limited resources.

Initially, processes lack knowledge of:

- inputs of other processes
- possibly some parameters of the system

Additional lack of knowledge can arise from:

- asynchrony
- faults

# Leader Election

Exactly one process (the leader) must output 1.

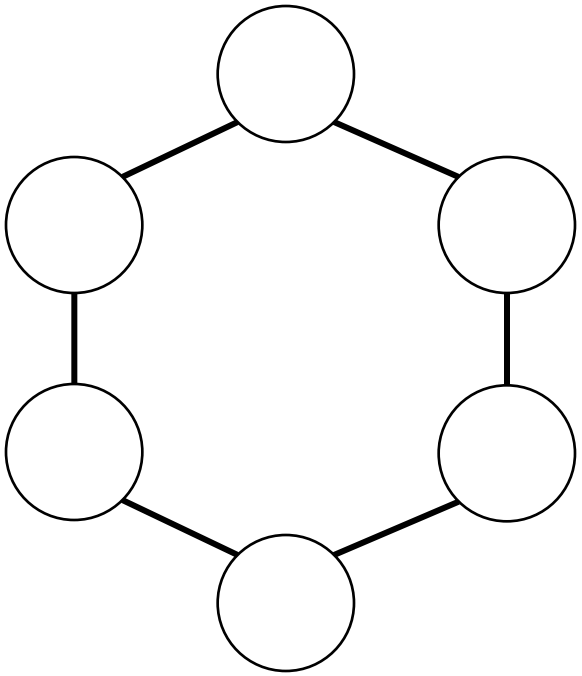
All other processes must output 0.

Model:

- synchronous message-passing
- anonymous processes
- each round, each process receives a message from each of its neighbours and sends a message to each of its neighbours

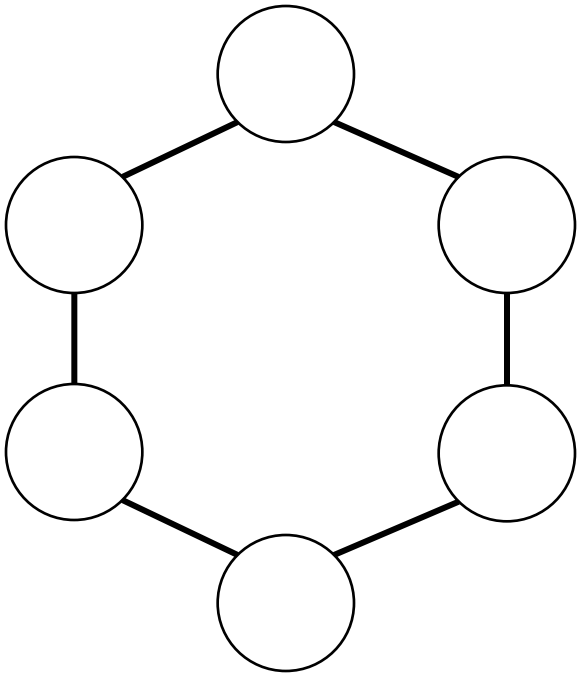
# Leader Election

THEOREM [Angluin 1980] There is no deterministic algorithm for leader election, even if all processes know the network is a cycle of size n.



# Leader Election

THEOREM [Angluin 1980] There is no deterministic algorithm for leader election, even if all processes know the network is a cycle of size n.



There is no way to break the initial symmetry.

By induction, at the end of every round, every process must be in the same state.

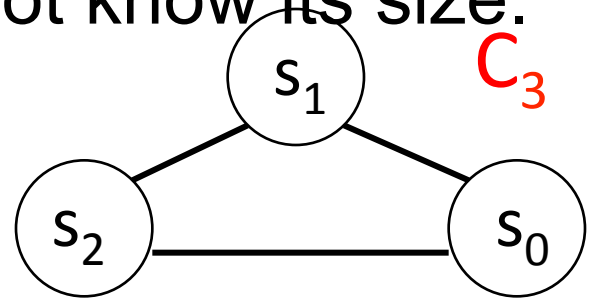
Thus if one process outputs 1, they all do.

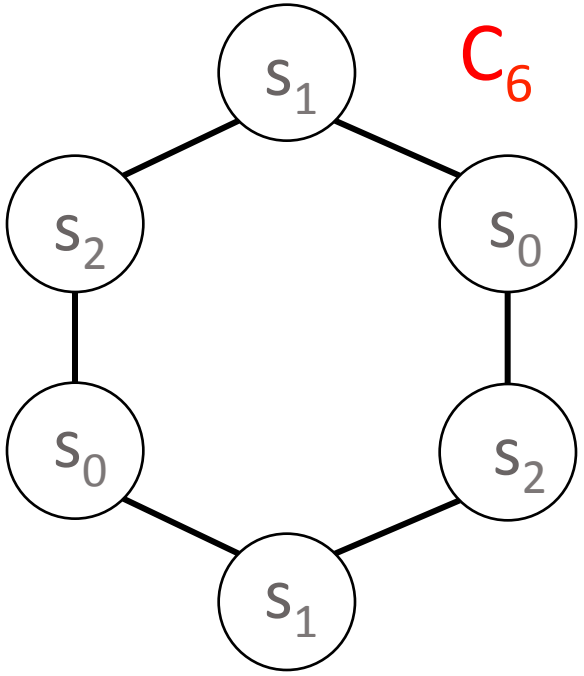
# Leader Election

THEOREM [Angluin 1980] There is no randomized algorithm for leader election, even if all processes know the network is a cycle, but do not know its size.

# Leader Election

THEOREM [Angluin 1980] There is no randomized algorithm for leader election, even if all processes know the network is a cycle, but do not know its size.




If one process outputs 1 in some execution in $C_3$, then there is an execution in $C_6$ in which two processes output 1.

# Leader Election

Every nonfaulty process must output 0 or 1.

At most one process can output 1.

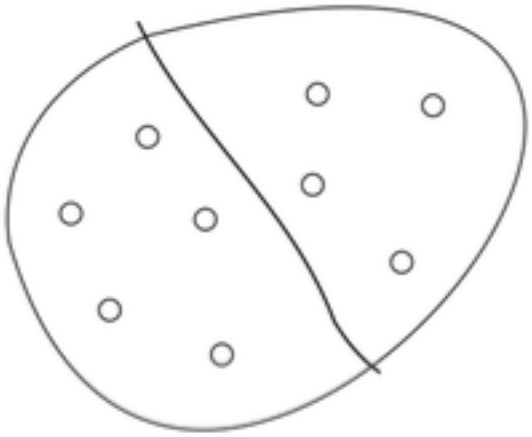
No process can output 0 until the eventual leader takes its first step.

Model:

- asynchronous message passing
- every process has a distinct ID
- each step, a process sends a message to one of its neighbours or receives a message that it was sent, but it has not yet received
- at most f processes can crash

# Leader Election

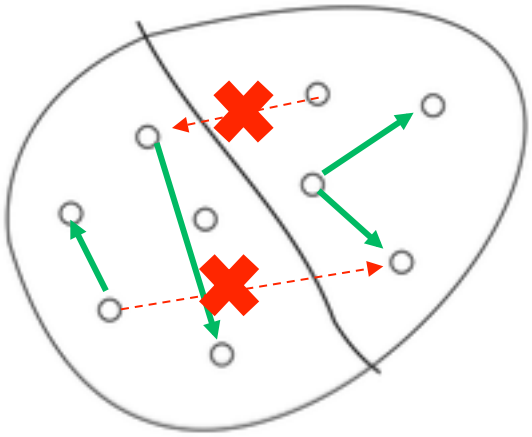
THEOREM [Gilbert & Lynch 2012] There is no algorithm for leader election in an asynchronous message-passing system with $n$ processes if $f \geq \lceil n/2 \rceil$ processes can crash.



Partition the network into 2 parts, each of size $\leq f$.

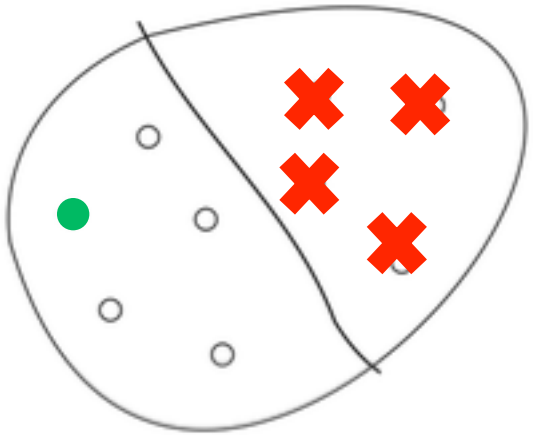
# Leader Election

All messages from a process to another process in the same part are delivered, but all messages to a process in the other part are delayed for a long time.



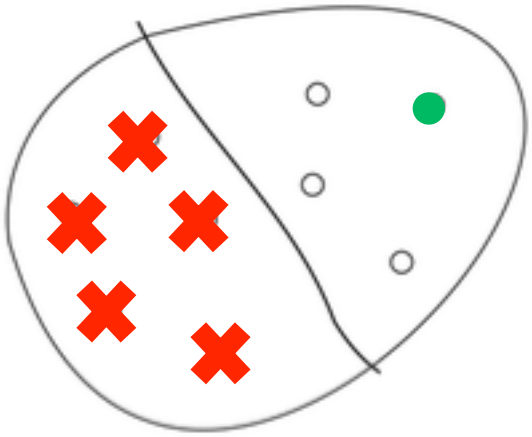
# Leader Election

As far as the processes in the left part know, the (≤ f) processes in the right part may have all crashed before taking any steps.



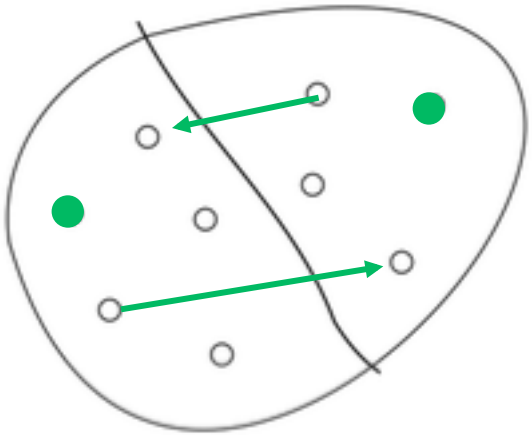
# Leader Election

As far as the processes in the right part know, the (≤ f) processes in the left part may have all crashed before taking any steps.



# Leader Election

After a process in each part has output 1, the scheduler delivers all undelivered messages.



# Indistinguishability

Configurations C and C' are indistinguishable

to a set of processes P, denoted $C \overset{P}{\sim} C'$, if

- each process in P has the same state in C and C'.

Executions α and α' starting from configurations C and C'

are indistinguishable to a set of processes P,

denoted $\alpha \overset{P}{\sim} \alpha'$, if

- $C \overset{P}{\sim} C'$ and
- each process in P performs the same steps in α and α' in the same order and gets the same responses from them.

# Indistinguishability

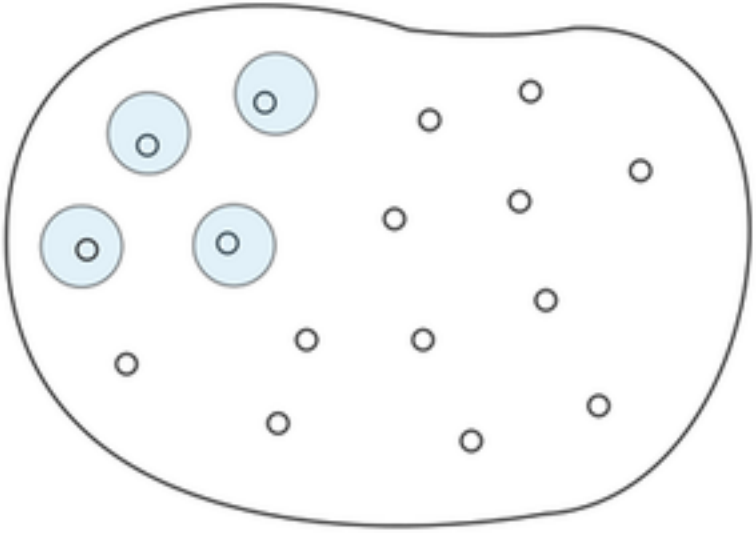
Suppose $\alpha$ is a finite execution starting from configuration $C$.

Then $C\alpha$ denotes the configuration at the end of $\alpha$.

Proposition Suppose $\alpha$ and $\alpha'$ are finite executions starting from configurations $C$ and $C'$, respectively. If $\alpha \overset{P}{\sim} \alpha'$, then $C\alpha \overset{P}{\sim} C'\alpha$.

# Leader Election

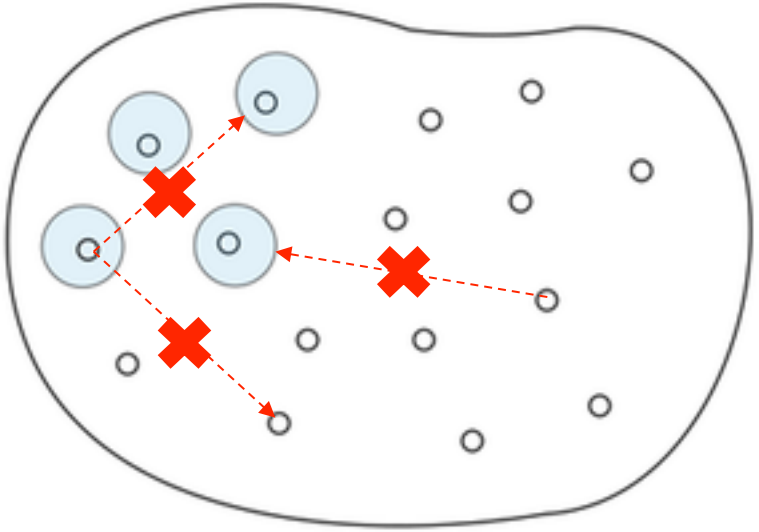
THEOREM[Alistarh, Gelashvili, & Vladu 2015]
Suppose up to $\lceil n/2 \rceil - 1$ processes can crash.
Any algorithm in an asynchronous message-passing system with $n$ processes has $\Omega(n^2)$ worst case expected message complexity.



The adversary picks a set of $\lfloor n/4 \rfloor$ processes and puts each in a bubble.

# Leader Election

Processes are allocated steps in round-robin order. Messages sent to or from a process in a bubble are trapped in its buffer. All other messages are delivered immediately.



A process is released from its bubble when its buffer contains $\geq n/4$ messages

# Leader Election

To each process not in a bubble, this execution is indistinguishable to an execution in which all processes that remain in bubbles have crashed.

To each process $p_i$ in a bubble, this execution is indistinguishable to an execution in which the following processes have crashed:
the other processes in bubbles,
the processes which have sent a message to $p_i$, and
the processes to which $p_i$ has sent a message.
Thus $p_i$ must eventually leave its bubble or return while it is still in a bubble.

# Leader Election

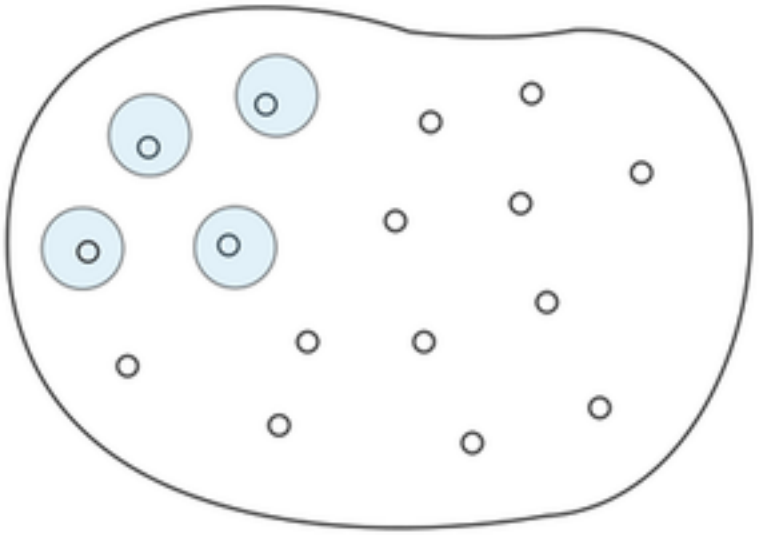
No process $p_i$ can return while it is still in a bubble because
all its steps might be before the steps of all other processes, in which case $p_i$ must return 1
or
all its steps might be after one or more other processes have returned, in which case $p_i$ must return 0.

Thus every process in a bubble must eventually leave its bubble.

# Leader Election

There are $\lfloor n/4 \rfloor$ processes initially in bubbles. A process is released from its bubble when its buffer contains $\geq n/4$ messages.

Thus $\Omega(n^2)$ messages are sent in this execution.



# Computing Diameter

Every process knows the size n of the network and the IDs of its neighbours.

Every process must output the diameter D of the network.

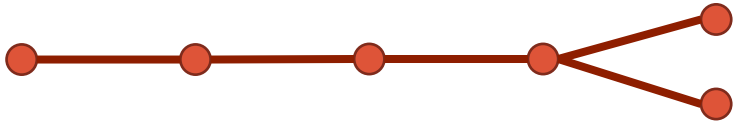
Model:

- synchronous message-passing
- each processes has a distinct ID
- each round, each process can send a message of unlimited size to each of its neighbours
- no faults

# Computing Diameter

requires $\Omega(D)$ rounds:

D=5

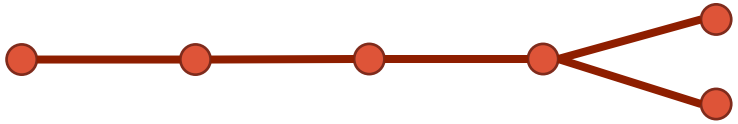


D=4

# Computing Diameter

requires $\Omega(D)$ rounds:

D=5



D=4

can be computed in $O(D)$ rounds

# Computing Diameter

Every process knows the size $n$ of the network and the IDs of its neighbours.

Every process must output the diameter $D$.

CONGEST Model:

- synchronous message-passing
- each processes has a distinct ID
- each round, each process can send a message of $B$ bits to each of its neighbours, where $B \in O(\log n)$
- no faults

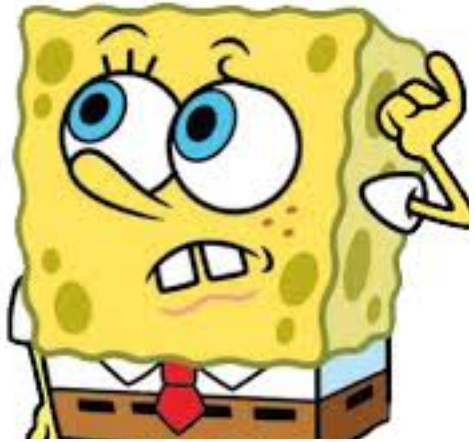
# Computing Diameter

THEOREM [Frischknecht, Holzer & Wattenhofer 2012] Any algorithm for determining the diameter $D$ of a graph with $n$ nodes using $B$-bit messages requires $\Omega(n/B)$ rounds.

# Computing Diameter

THEOREM [Frischknecht, Holzer& Wattenhofer 2012] Any algorithm for determining the diameter $D$ of a graph with $n$ nodes using $B$-bit messages requires $\Omega(n/B)$ rounds, even if all nodes know that $D \in \{2,3\}$.

# Communication Complexity



$(x_1,\ldots, x_k) \in \{0,1\}^k$ $(y_1,\ldots, y_k) \in \{0,1\}^k$

Set Disjointness Problem:

Determine if, for all $i \in \{1,\ldots,k\}$, either $x_i = 0$ or $y_i = 0$.

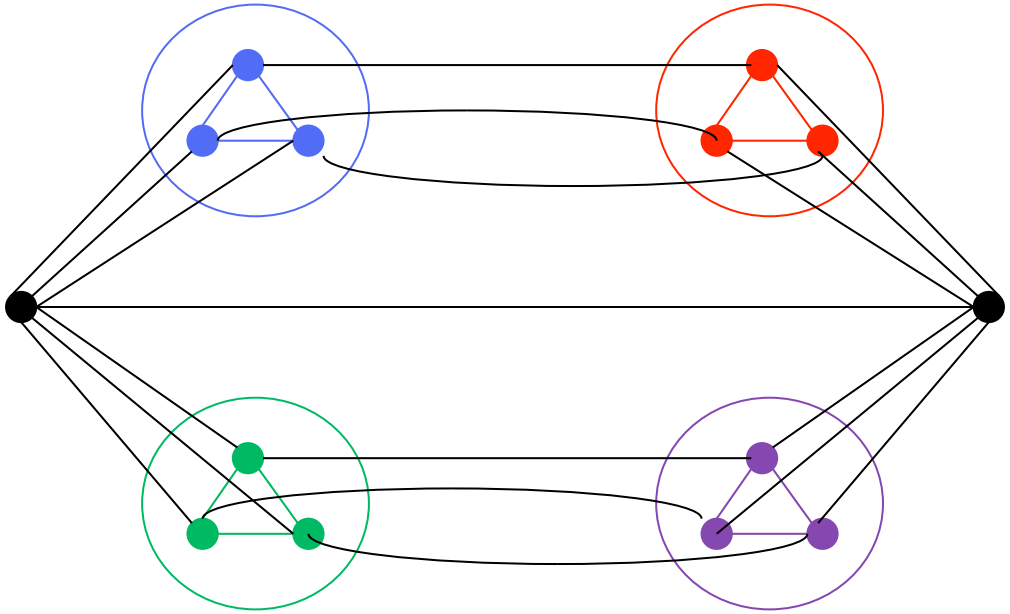
# Communication Complexity

$(x_1,\ldots, x_k) \in \{0,1\}^k$ $(y_1,\ldots, y_k) \in \{0,1\}^k$

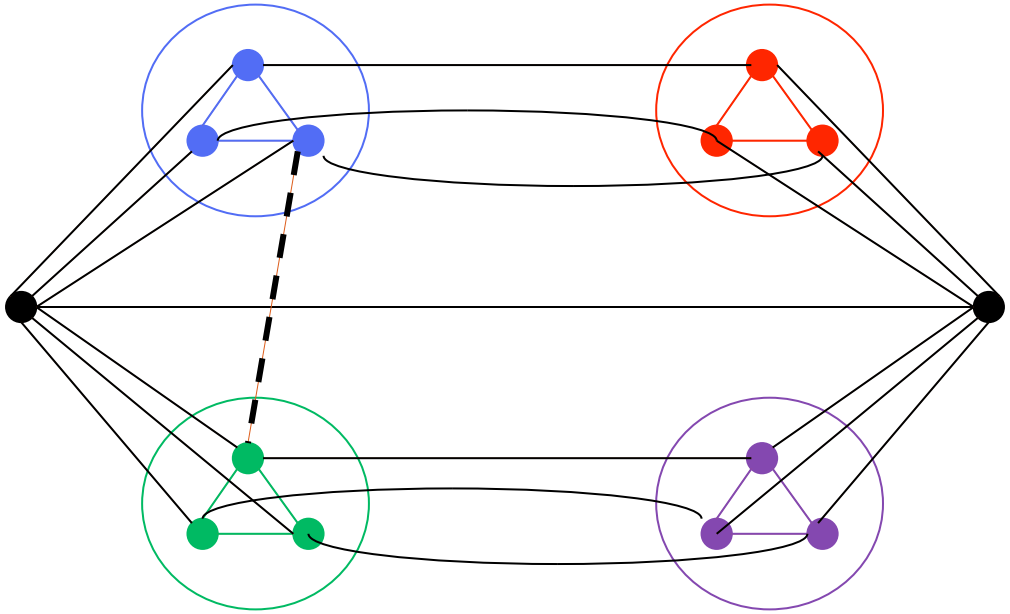
Set Disjointness Problem:
Determine if, for all $i \in \{1,\ldots,k\}$, either $x_i = 0$ or $y_i = 0$.

THEOREM [Kalyanasundaram and Schnitger, Razborov] $\Omega(k)$ bits of communication are required by any randomized protocol for set disjointness.

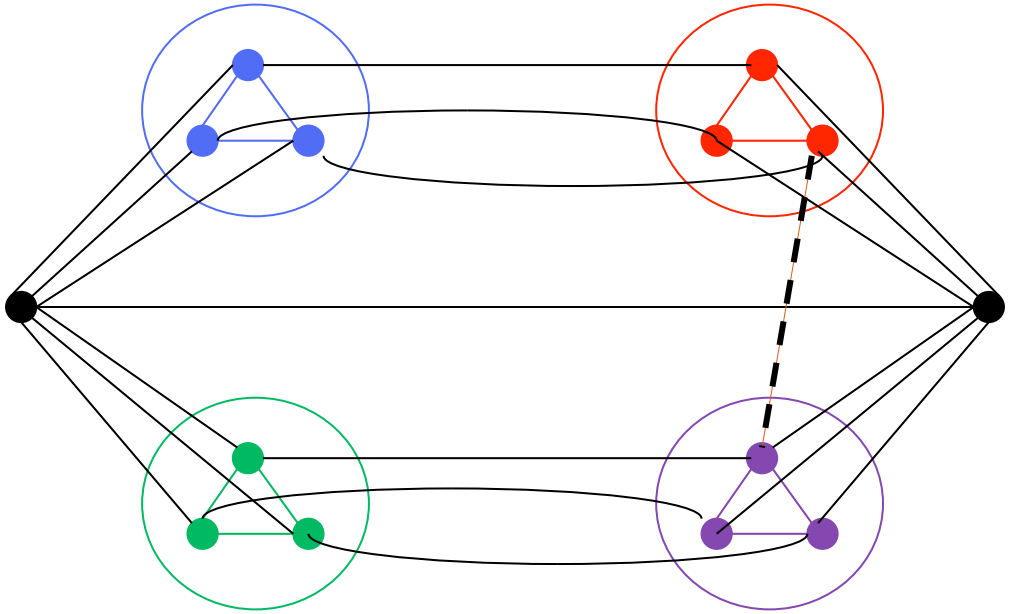
# Computing Diameter



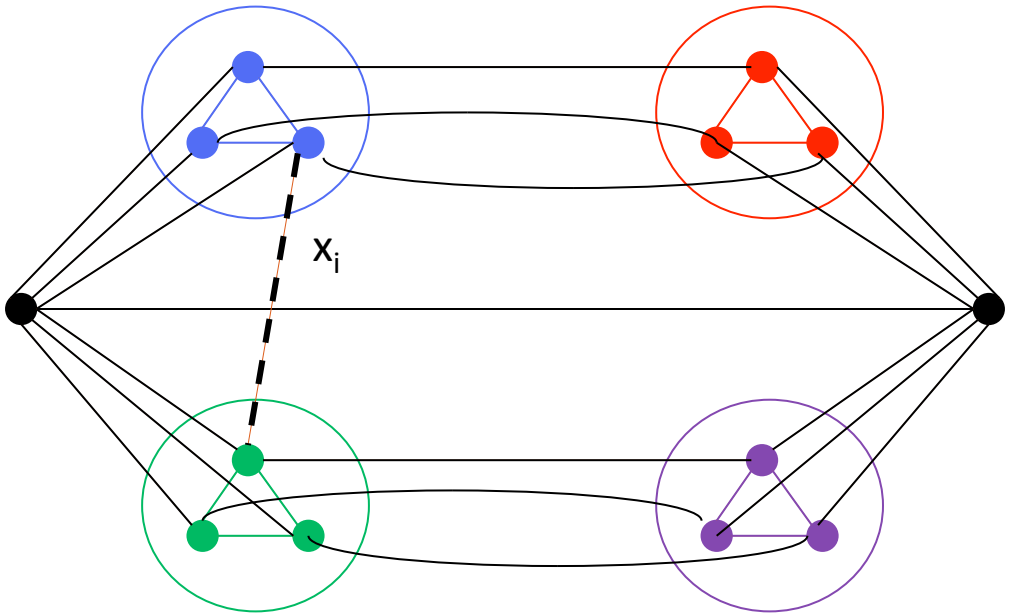
# Computing Diameter



# Computing Diameter

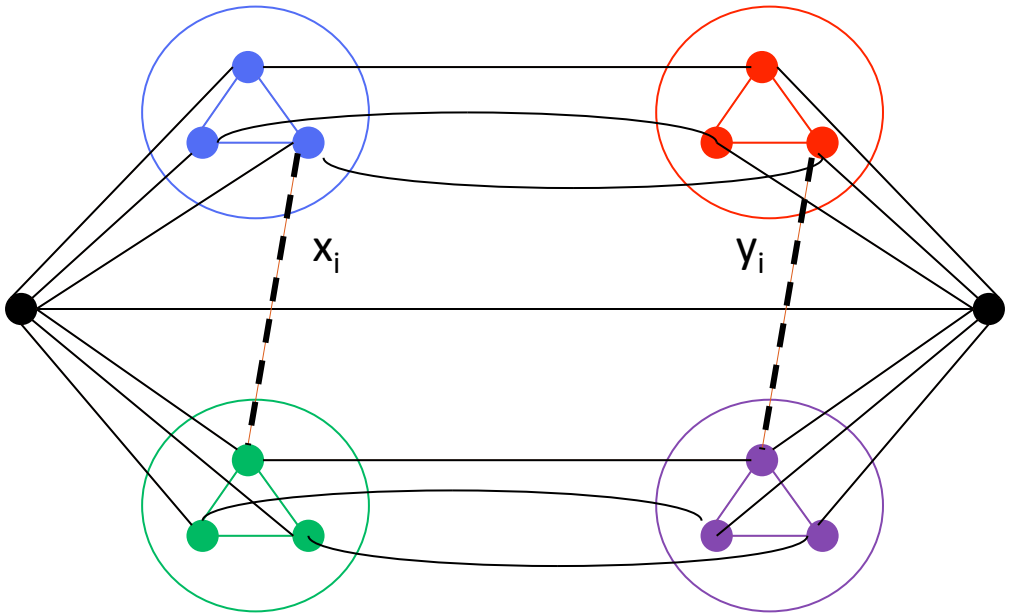


# Computing Diameter




Associate a different variable $x_i$ with each of the $k = ((n-2)/4)^2$ possible edges between blue and green nodes.
The edge is in the graph if and only if $x_i = 0$.

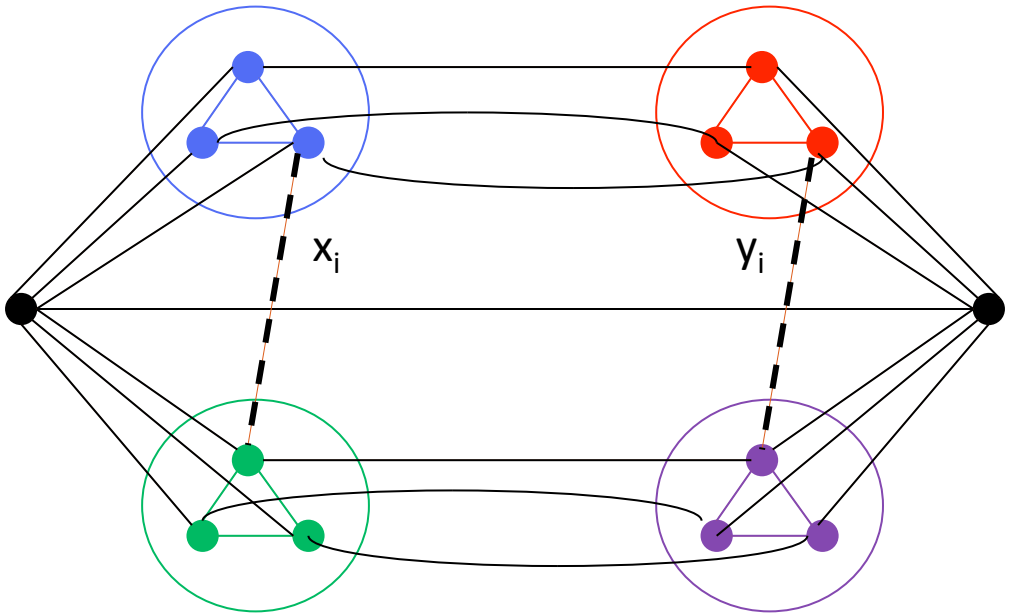
# Computing Diameter




Associate the variable $y_i$ with the possible edge between the corresponding red and purple nodes.

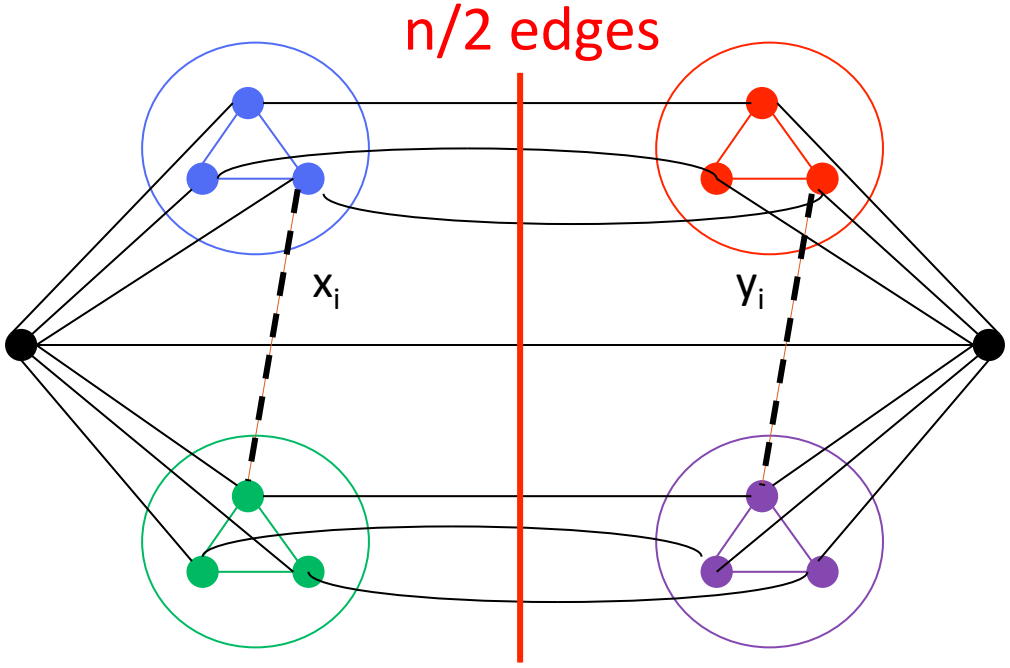
The edge is in the graph if and only if $y_i = 0$.

# Computing Diameter



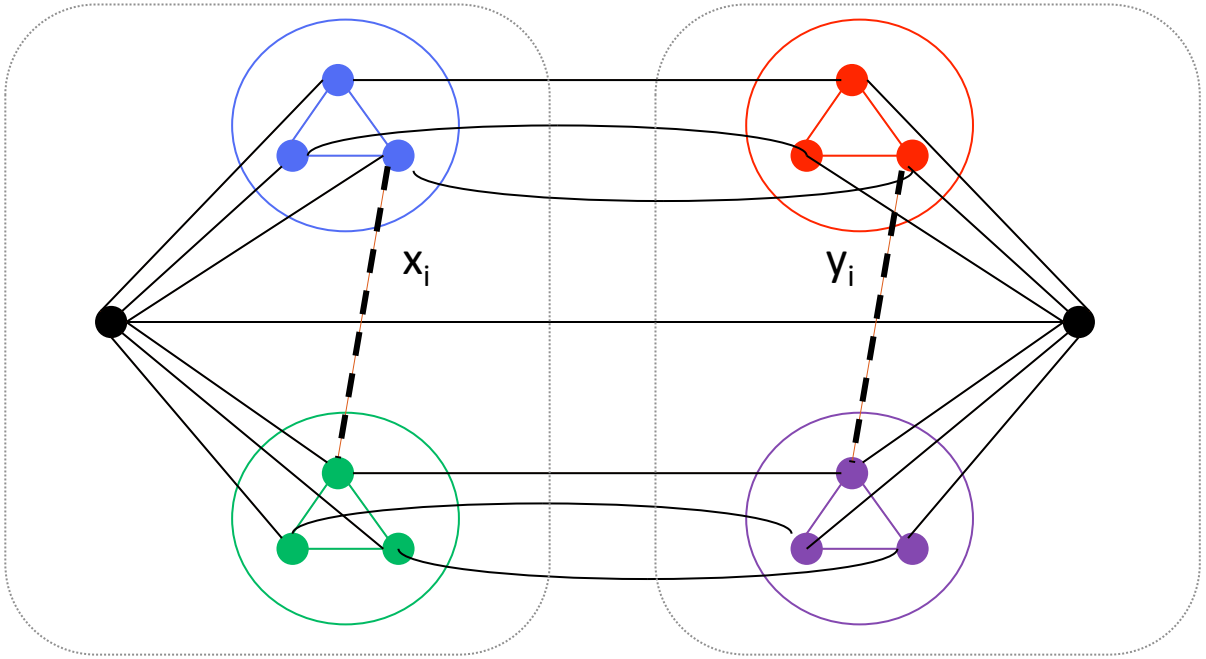

The resulting graph has diameter 2 if and only if for all $i \in \{1,\ldots,k\}$, either $x_i = 0$ or $y_i = 0$.
Otherwise the graph has diameter 3.

# Computing Diameter




Suppose there is an R round algorithm for determining whether the diameter of any graph in this class has diameter 2 or 3.
It communicates at most nRB bits across these edges.

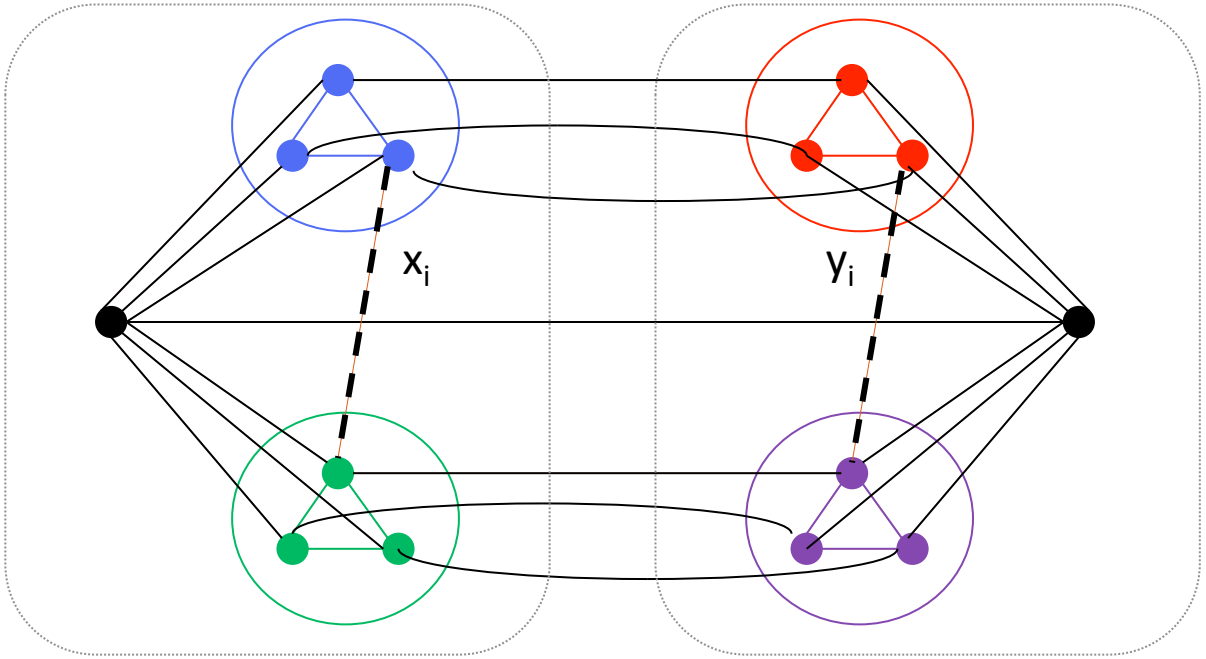
# Solving Set Disjointness for $k = ((n-2)/4)^2$




Alice has input $x_1,\ldots,x_k$

Bob has input $y_1,\ldots,y_k$

# Solving Set Disjointness for $k = ((n-2)/4)^2$




They simulate the diameter algorithm, communicating at most nRB bits. Hence $nRB \in \Omega(k) = \Omega(n^2)$.
Thus, the diameter algorithm takes $R \in \Omega(n/B)$ rounds.

Computing the Diameter D of an n node graph with B-bit messages

| Approximation Factor | Number of Rounds |
| --- | --- |
| 1 | $\Omega(D + n/B)$ |
| | [Frischknecht, Holzer& Wattenhofer 2012] |
| $(3/2 - \varepsilon)$ | $\Omega(D + n/B\text{polylog}(n))$ |
| | [Abboud, Censor-Hillel & Khoury 2016] |
| 3/2 | $O(D + n^{1/2})$ |
| | [Holzer, Peleg, Roditty & Wattenhofer 2014] |
| 2 | $O(D)$ |
| | Breath-first search |

# Impossibility Results for Distributed Computing Part II

Faith Ellen
Department of Computer Science
University of Toronto

# Covering Arguments

A process $p_i$ covers a register R in a configuration C if $p_i$ writes to R whenever $p_i$ is next allocated a step by the scheduler.

Previous writes to R by other processes are hidden by performing this write.

# Covering Arguments

A process $p_i$ covers a register R in a configuration C if $p_i$ writes to R whenever $p_i$ is next allocated a step by the scheduler.

Previous writes to R by other processes are hidden by performing this write.

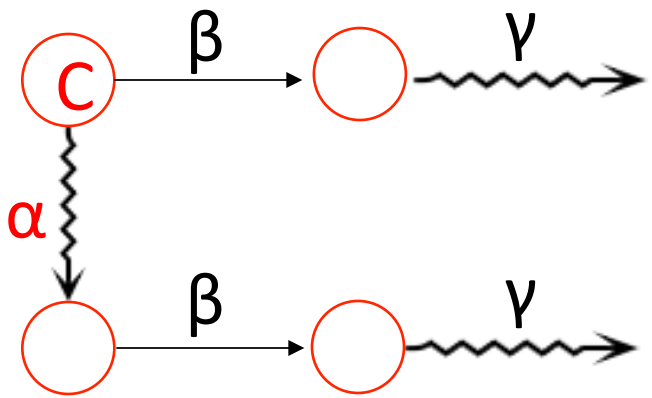
A block write is a sequence of consecutive steps by different processes in which they each write to a different register.

Previous writes to this set of registers by other processes are hidden when this block write is performed.

# Covering Arguments

The goal of a covering argument is to construct a configuration C in which all registers are covered.

Then any execution α from C by the remaining processes can be hidden by a block write β.



# Covering Arguments

The goal of a covering argument is to construct a configuration C in which all registers are covered.

Then any execution from C by the remaining processes can be hidden by a block write.

This can be used to show that any asynchronous algorithm using too few registers (each of unbounded size) can behave incorrectly.

The construction is usually inductive, with the number of covered objects increasing as the argument progresses.

THEOREM [Burns & Lynch 1993] Any mutual exclusion algorithm for $n \geq 2$ processes uses at least $n$ registers.

Model

- asynchronous shared-memory system
- every process has a distinct ID
- processes communicate through shared (read/write) registers of unbounded size
- no faults

# Mutual Exclusion Problem

This problem models the situation in which processes need temporary exclusive access to a shared resource.

- To get the resource, a process performs an entry protocol.
- When a process has finished using the resource, it performs an exit protocol.
- A process is idle if it does not have the resource and is not performing its entry or exit protocol.

# Mutual Exclusion Problem

A mutual exclusion algorithm consists of entry and exit protocols for each process that satisfy:

- mutual exclusion: in any configuration, at most one process has the resource, and
- deadlock freedom: starting from any configuration C in which some process is performing its entry protocol and no process has the resource, there is a finite execution by processes that are not idle in C which causes some process to get the resource.

LEMMA 1 Suppose

$\alpha$ is a finite execution by process $p_i$ starting from configuration C,

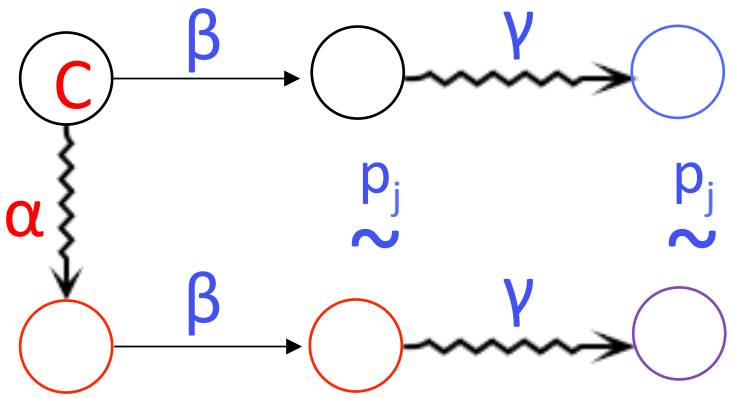
$p_i$ is idle in C,

$p_i$ has the resource in C$\alpha$, and

R is the set of registers covered in C.

Then, during $\alpha$, $p_i$ writes to some register not in R.

Proof: By contradiction.




$\beta$ is a block write to R.
Some process $p_j \neq p_i$ has the resource in C$\beta\gamma$.

A configuration is quiescent if every process is idle.

LEMMA 2 From any quiescent configuration Q and for $1 \leq k \leq n$, there are executions $\sigma$ and $\delta$ such that

- $Q\delta$ is quiescent,
- all registers have the same values in $Q\sigma$ and $Q\delta$
- $p_0,\ldots,p_{k-1}$ cover $k$ different registers in $Q\sigma$, and
- $Q\sigma$ and $Q\delta$ are indistinguishable to all other processes.

A configuration is quiescent if every process is idle.

LEMMA 2 From any quiescent configuration Q and for $1 \le k \le n$, there are executions $\sigma$ and $\delta$ such that

- $Q\delta$ is quiescent,
- all registers have the same values in $Q\sigma$ and $Q\delta$,
- $p_0,\ldots,p_{k-1}$ cover $k$ different registers in $Q\sigma$, and
- $Q\sigma$ and $Q\delta$ are indistinguishable to all other processes.

PROOF OF THEOREM: The initial configuration is quiescent. Apply Lemma 2 with $k = n$.

PROOF OF LEMMA 2: By induction on $k$.

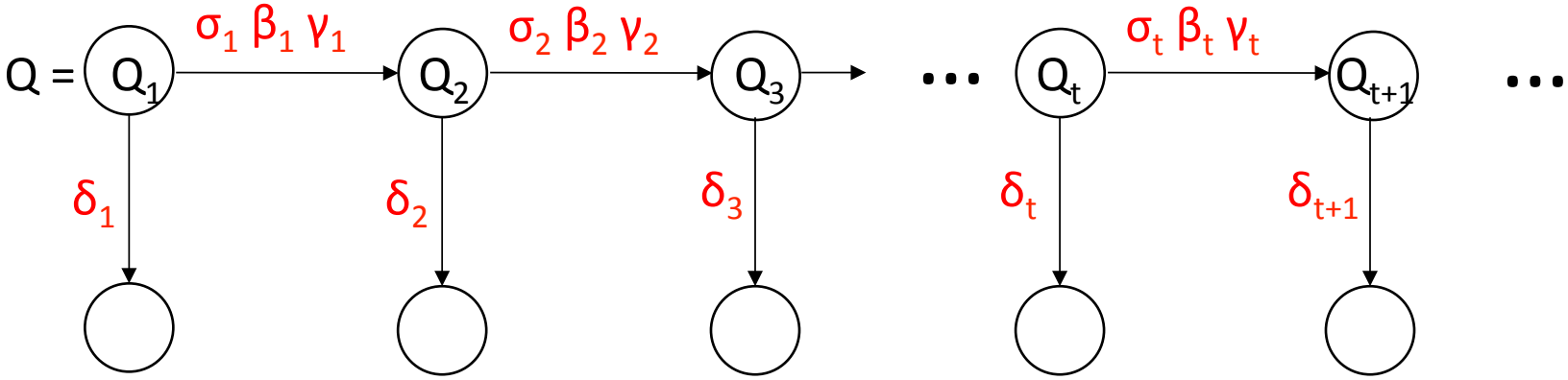
Base case: $k = 1$.
By deadlock freedom, there is an execution $\alpha$ by $p_0$ in which $p_0$ gets the resource.
By Lemma 1, during $\alpha$, $p_0$ writes to some register.
Let $\sigma$ be the longest prefix of $\alpha$ that contains no writes and let $\delta$ be the empty execution.
Then

- $Q\delta = Q$ is quiescent,
- all registers have the same values in $Q\sigma$ and $Q\delta$,
- $p_0$ covers 1 register in $Q\sigma$, and
- $Q\sigma$ and $Q\delta$ are indistinguishable to all other processes.

Induction step: Let $1 \leq k < r$ and assume the claim is true for $k$.
By the induction hypothesis, from any quiescent configuration $Q_t$, there are executions $\sigma_t$ and $\delta_t$ that satisfy the claim.

Since $Q_t\delta_t$ is quiescent and $Q_t\sigma_t$ and $Q_t\delta_t$ are indistinguishable to all other processes,
they are idle in $Q_t\sigma_t$.

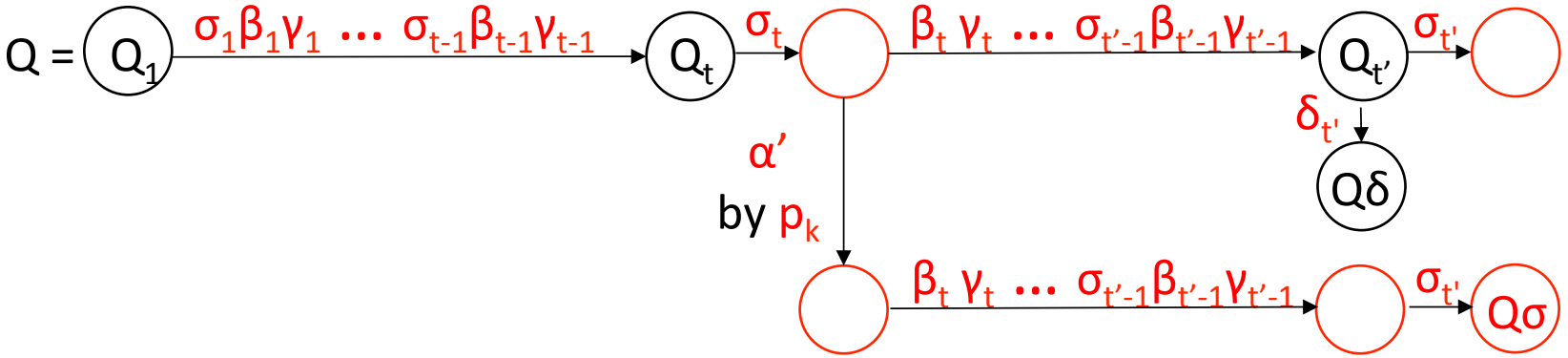
Let $\beta_t$ be a block write by $\{p_0,\ldots,p_{k-1}\}$ starting from $Q_t\sigma_t$.
Let $\gamma_t$ be a finite execution by $\{p_0,\ldots,p_{k-1}\}$ starting from $Q_t\sigma_t\beta_t$ such that $Q_{t+1} = Q_t\sigma_t\beta_t\gamma_t$ is quiescent.




By the pigeon hole principle, there exist $1 \leq t < t'$ such that $\beta_t$ and $\beta_{t'}$ are block writes to the same set of $k$ registers $R$.

Since $Q_t\delta_t$ is quiescent, there is a finite execution $\alpha$ by $p_k$ starting from $Q_t\delta_t$ such that $p_k$ has the resource in $Q_t\delta_t\alpha$.

Since $Q_t\sigma_t \overset{p_k}{\simeq} Q_t\delta_t$ and all registers have the same values in $Q_t\sigma_t$ and $Q_t\delta_t$, $\alpha$ can occur starting from $Q_t\sigma_t$ and $p_k$ has the resource in $Q_t\sigma_t\alpha$.



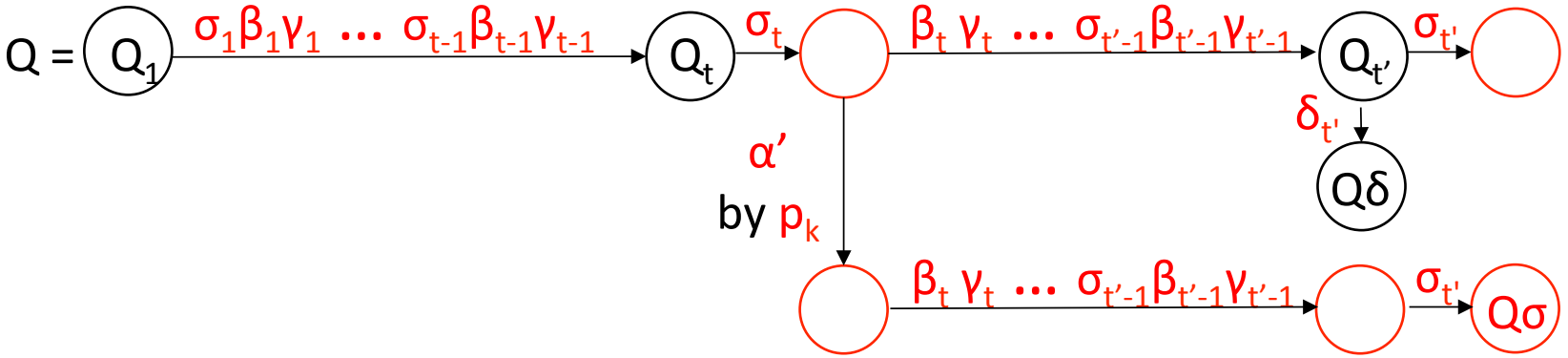

By Lemma 1, during $\alpha$, $p_k$ writes to some register not in R. Let $\alpha'$ be the shortest prefix of $\alpha$ such that, in $Q_t\sigma_t\alpha'$, $p_k$ covers a register not in R.

Since $\beta_t$ is a block write to R, each register has the same value in $Q_t\sigma_t\beta_t$ and $Q_t\sigma_t\alpha'\ \beta_t$.

Since $Q_t\sigma_t\beta_t$ and $Q_t\sigma_t\alpha'\ \beta_t$ are indistinguishable to all other processes, the execution $\gamma_t\sigma_{t+1}\beta_{t+1}\gamma_t \ldots \gamma_{t'-1}$ can also occur starting from $Q_t\sigma_t\alpha'\ \beta_t$.

Let $\sigma = \sigma_1\beta_1\gamma_1 \ldots \sigma_{t-1}\beta_{t-1}\gamma_{t-1}\sigma_t\alpha'\ \beta_t\ \gamma_t \ldots \sigma_{t'-1}\beta_{t'-1}\gamma_{t'-1}\sigma_{t'}$

and $\delta = \sigma_1\beta_1\gamma_1 \ldots \sigma_{t-1}\beta_{t-1}\gamma_{t-1}\sigma_t\ \ \beta_t\ \gamma_t \ldots \sigma_{t'-1}\beta_{t'-1}\gamma_{t'-1}\delta_{t'}$.




Let $\sigma = \sigma_1\beta_1\gamma_1 \ldots \sigma_{t-1}\beta_{t-1}\gamma_{t-1}\sigma_t\alpha' \beta_t \gamma_t \ldots \sigma_{t'-1}\beta_{t'-1}\gamma_{t'-1}\sigma_{t'}$

and $\delta = \sigma_1\beta_1\gamma_1 \ldots \sigma_{t-1}\beta_{t-1}\gamma_{t-1}\sigma_t \quad \beta_t \gamma_t \ldots \sigma_{t'-1}\beta_{t'-1}\gamma_{t'-1}\delta_{t'}$.

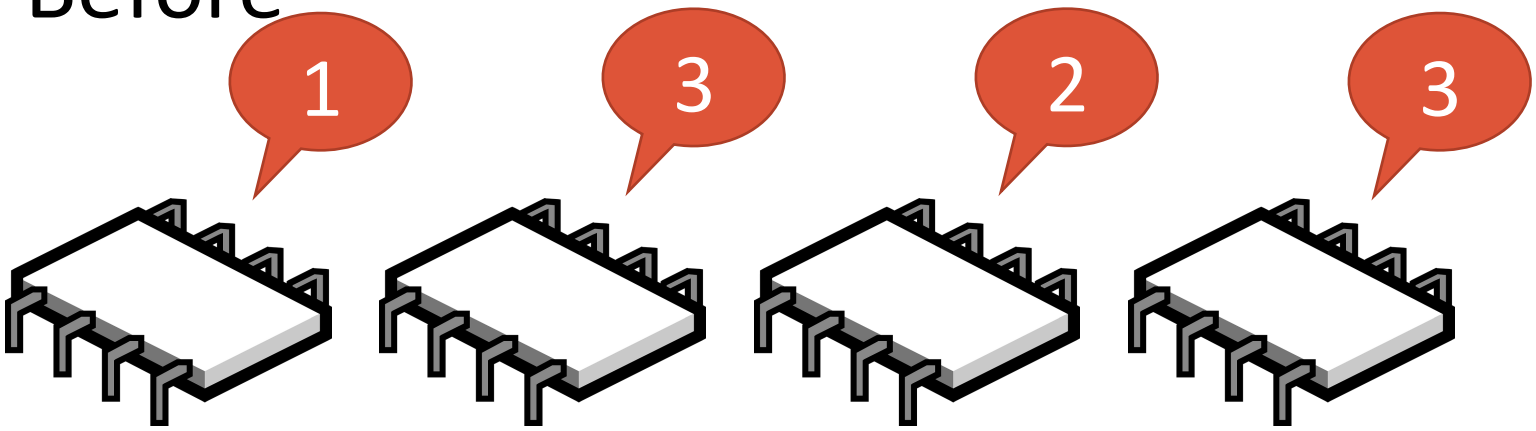
Then

- $Q\delta$ is quiescent,
- all registers have the same values in $Q\sigma$ and $Q\delta$,
- $p_0,\ldots,p_k$ cover $k+1$ different registers in $Q\sigma$, and
- $Q\sigma$ and $Q\delta$ are indistinguishable to all other processes.

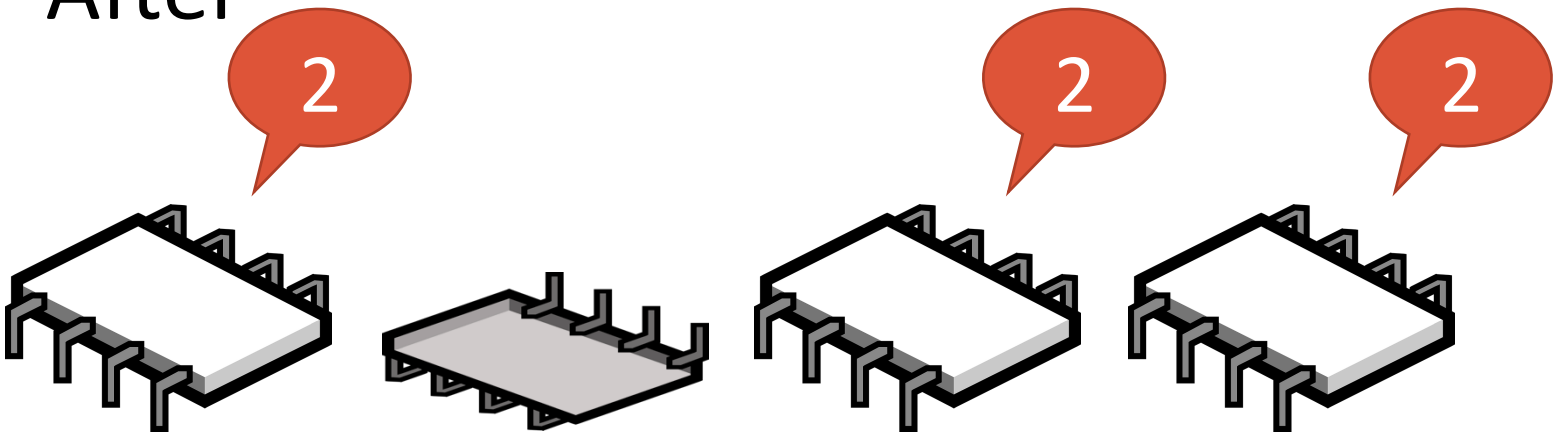
Thus the claim is true for $k+1$.

# Consensus

Before




After



# Consensus Problem

Each process $p_i$ has a private input value $v_i$.

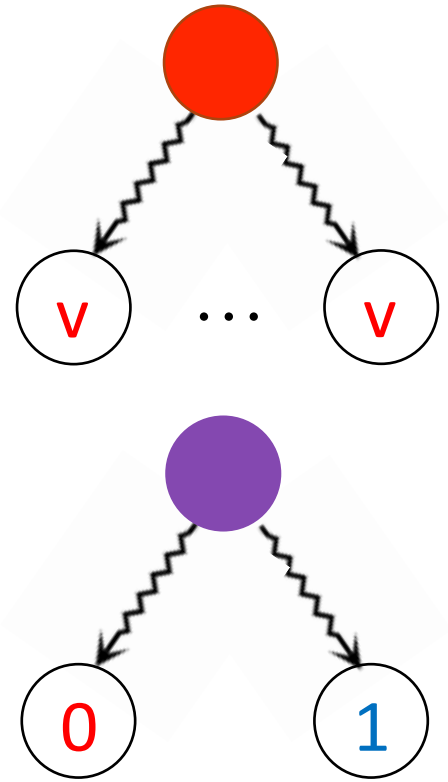
- Agreement: All output values are the same.
- Validity: The output value of each process is the input value of some process.
- Wait-Free Termination: Each non-faulty process outputs a value after taking a finite number of steps.

# Valency Arguments

A configuration C is:

v-univalent if all executions starting from C output v and

bivalent if there are two executions starting from C that output different values.



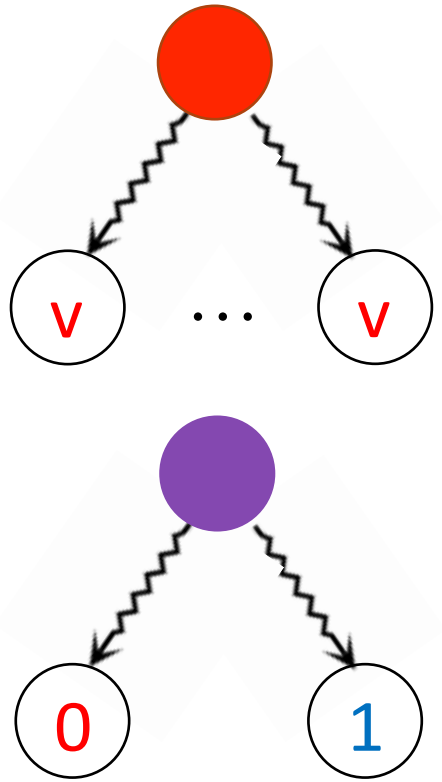
# Valency Arguments

A configuration C is:

v-univalent if all executions starting from C output v and

bivalent if there are two executions starting from C that output different values.

After value v has been output, every configuration is v-univalent.



THEOREM [Fischer, Lynch & Paterson 1983, Chor, Israeli & Li 1987, Loui & Abu Amara 1987, Abrahamson 1988, Herlihy 1991]
There is no deterministic algorithm to solve consensus among $n \geq 2$ processes in an asynchronous system.

Model

- asynchronous shared-memory
- processes communicate using (read/write) registers
- every process has a distinct ID
- at most $n-1$ processes can crash

THEOREM [Fischer, Lynch & Paterson 1983, Chor, Israeli & Li 1987, Loui & Abu Amara 1987, Abrahamson 1988, Herlihy 1991]
There is no deterministic algorithm to solve consensus among $n \geq 2$ processes in an asynchronous shared memory system where processes communicate using registers.

Idea: An adversary can create an infinitely long execution that doesn't output a value.

LEMMA 1 Every consensus algorithm has a bivalent initial configuration.

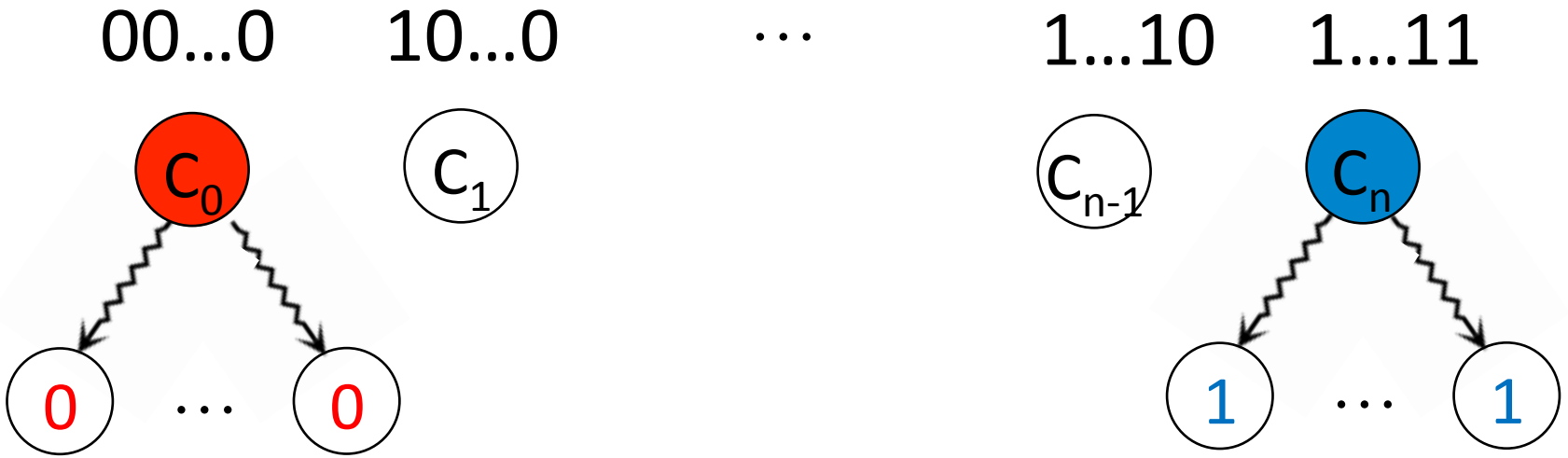
Proof: by contradiction. Suppose all initial configurations are univalent.

Let $C_i$ denote the initial configuration where, for $0 \leq j < n$, process $p_j$ has input value

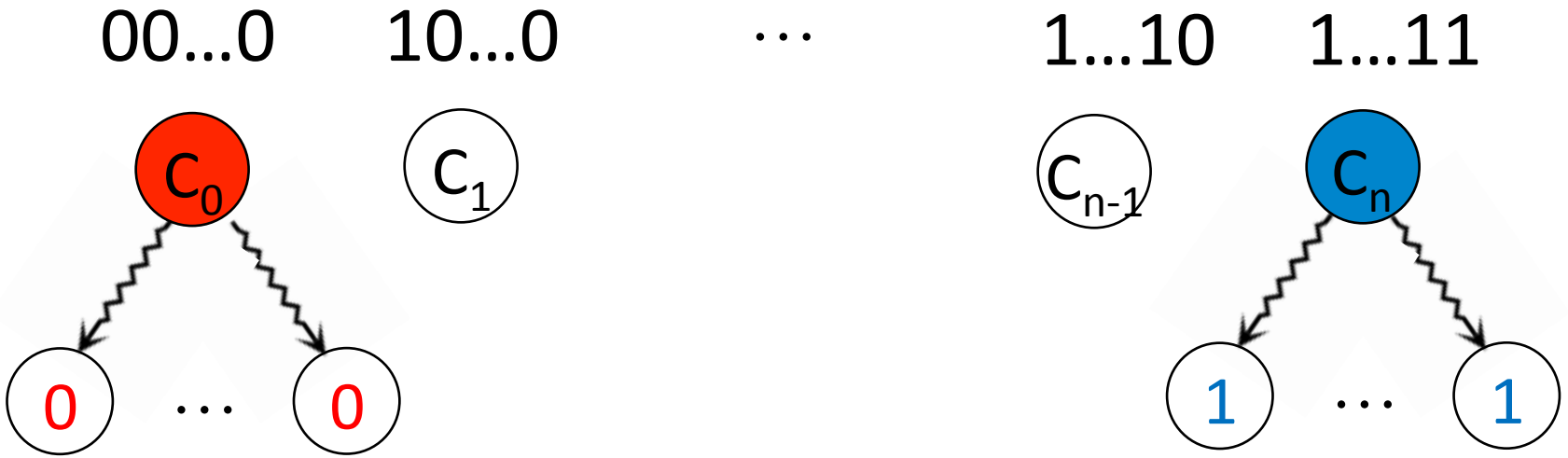
1 if $j < i$

0 if $j \geq i$.

| 00…0 | 10…0 | … | 1…10 | 1…11 |
|---|---|---|---|---|
| $C_0$ | $C_1$ | | $C_{n-1}$ | $C_n$ |

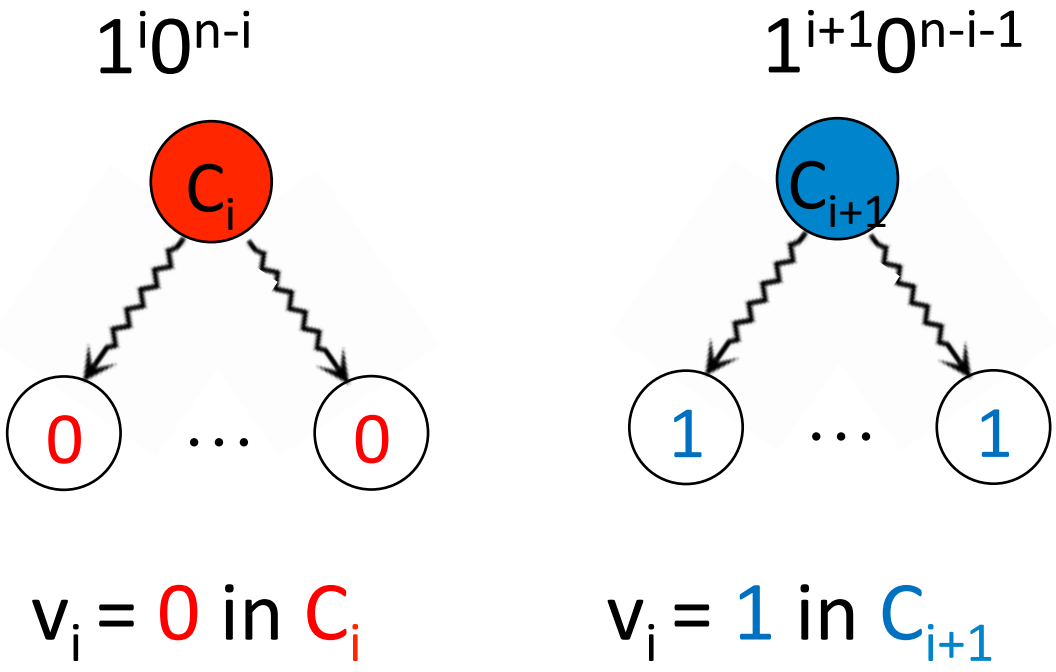



By validity, $C_0$ is 0-univalent and $C_n$ is 1-univalent.



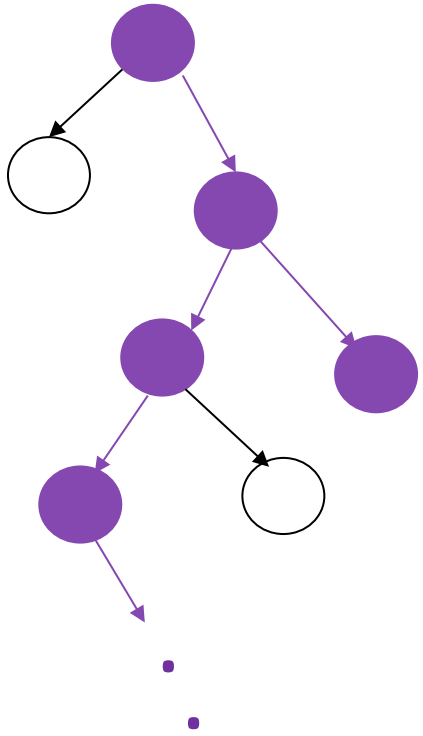

By validity, $C_0$ is 0-univalent and $C_n$ is 1-univalent.

Hence, there exists $i \in \{0,\ldots,n-1\}$ such that $C_i$ is 0-univalent and $C_{i+1}$ is 1-univalent.



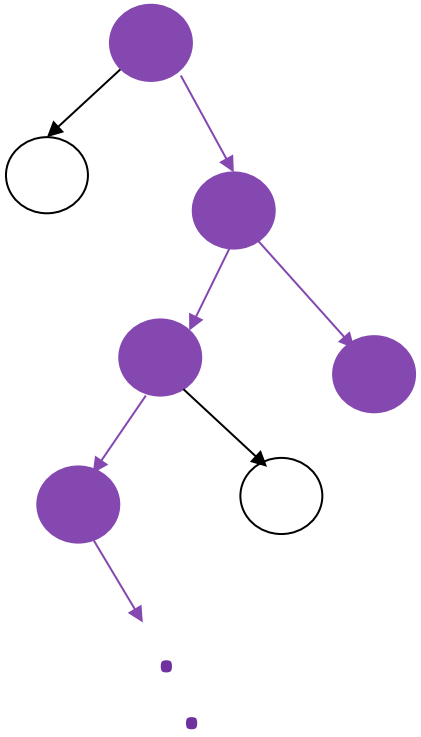

Consider any execution $\alpha$ starting from $C_i$ in which $p_i$ crashes before taking any steps. To all other processes, $C_i$ and $C_{i+1}$ are indistinguishable. Therefore, the non-faulty processes also output 0 when $\alpha$ starts from $C_{i+1}$.

LEMMA 2 From every bivalent configuration, there is a step that leads to a bivalent configuration.

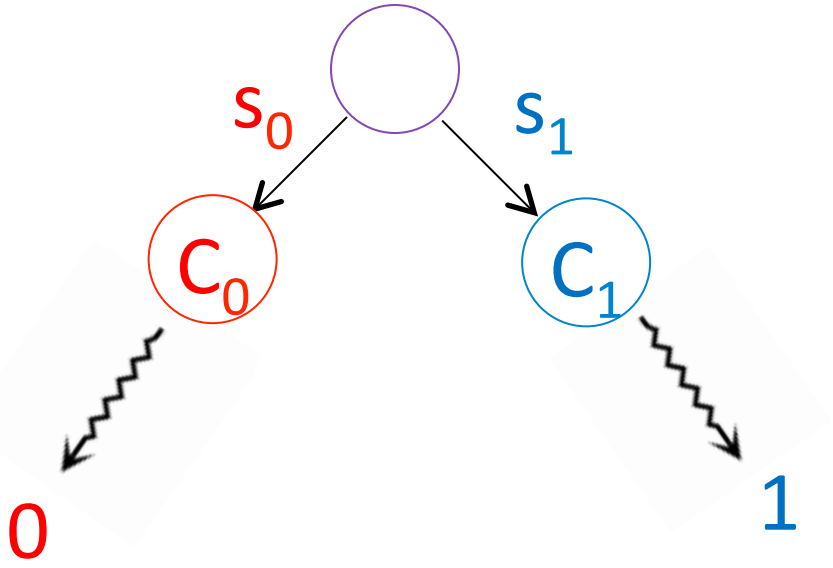


LEMMA 2 From every bivalent configuration, there is a step that leads to a bivalent configuration.

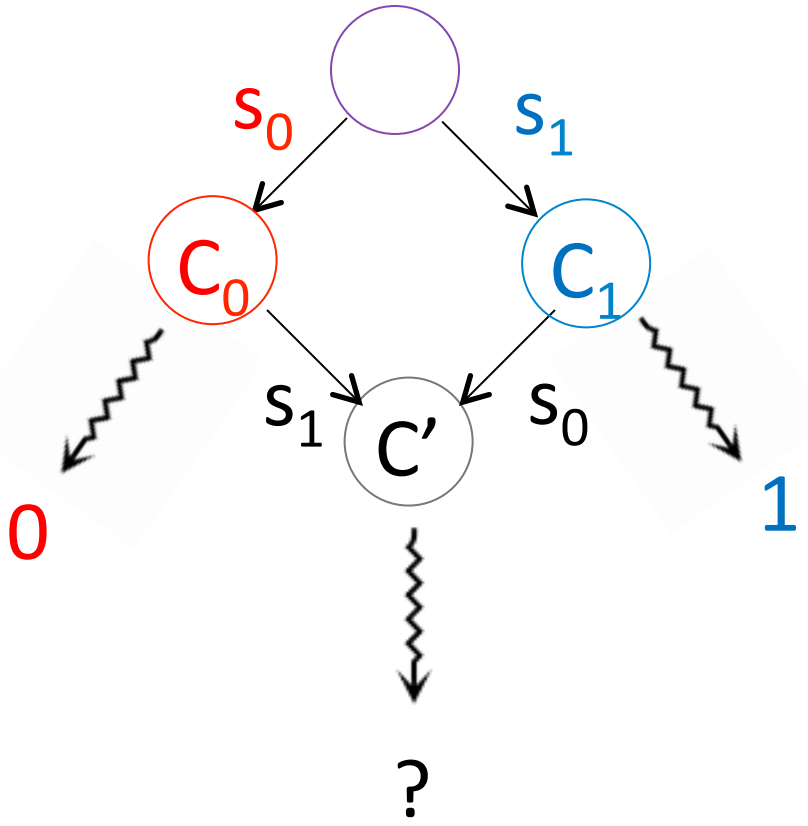
This implies there is an infinite execution, consisting of only bivalent configurations, violating wait-free termination.



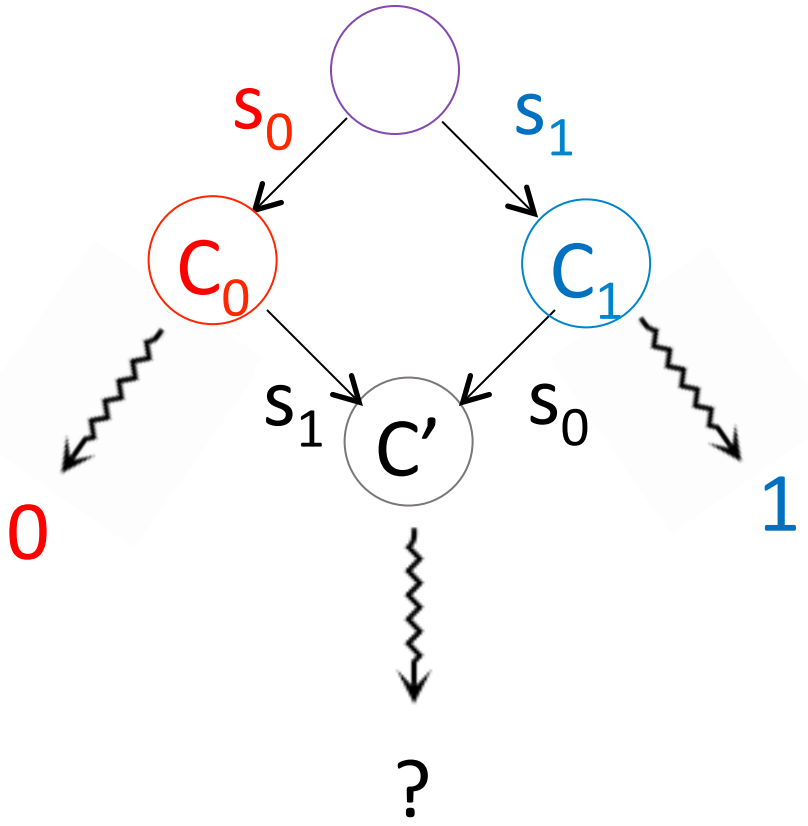
Consider any bivalent configuration C, where the next step $s_0$ by process $p_0$ results in a 0-univalent configuration $C_0$ and the next step $s_1$ by process $p_1$ results in a 1-univalent configuration $C_1$.



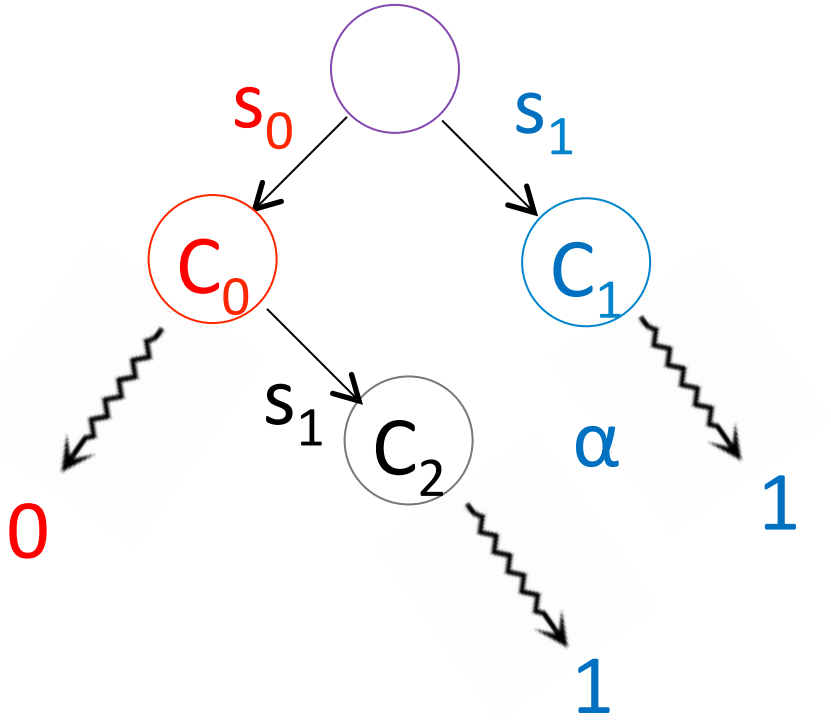
$s_0$ and $s_1$ access different registers



$s_0$ and $s_1$ read the same register



$s_1$ writes to the the same register that $s_0$ accesses




$C_1$ and $C_2$ are indistinguishable to $p_1$, so the solo execution α by $p_1$ from $C_1$ behaves the same when started from $C_2$.

Consensus can be solved in an asynchronous shared-memory system using a single Compare&Swap object.

Consensus can be solved in an asynchronous shared-memory system using a single Compare&Swap object.

Therefore, asynchronous shared memory systems with only registers are less powerful than asynchronous shared-memory systems with Compare&Swap objects and registers.

# Randomized Consensus

- Agreement: All output values are the same.
- Validity: The output value of each process is the input value of some process.
- Randomized Wait-Free Termination: Each non-faulty process outputs a value after taking an expected finite number of steps.

There is an algorithm that solves randomized consensus among $n$ processes in which the expected total number of steps taken by all processes is $O(n^2)$.

THEOREM [Attiya&Censor-Hillel 2008]
For any algorithm that solves randomized consensus among $n$ processes, the expected total number of steps taken by all processes is $\Omega(n^2)$.

There are randomized algorithms for consensus among n processes that use n registers.

THEOREM [Zhu 2016] Any randomized algorithm for consensus among n processes must use at least n-1 registers.

THEOREM [Ellen, Gelashvili & Zhu 2018] Any randomized algorithm for consensus among n processes must use at least n registers.

# k-set Agreement Problem

Each process $p_i$ has a private input value $v_i$.

- Agreement: At most $k$ different values are output.
- Validity: The output value of each process is the input value of some process.
- Wait-Free Termination: Each non-faulty process outputs a value after taking a finite number of steps.

There are randomized algorithms for $k$-set agreement among $n$ processes that use $n-k+1$ registers.

THEOREM [Ellen, Gelashvili & Zhu 2018] Any randomized algorithm for $k$-set agreement among $n$ processes must use at least $\lfloor (n-1)/k \rfloor + 1$ registers.