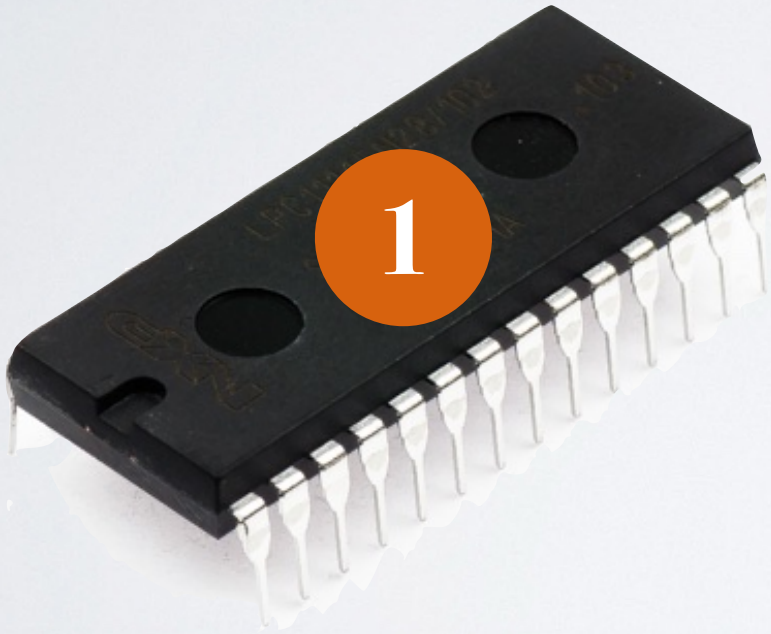
THE BASIC IDEA: THE WRITE-ONCE REGISTER



Leaders compete to create a permanent mapping between slot numbers and proposals

Participants forever

read register

if register contains a mapping then

decide that mapping and halt

else

attempt to write own mapping

How to implement a write once register?

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

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

- A leader never proposes a map that may conflict with what is stored in Paxos register
- A leader, before attempting to create a new map between a slot number for which it knows not a decision and a proposal, "reads" the Paxos register to check whether such map may already exist
- Once a leader learns that a new mapping has become permanent, it informs the replicas

Ballots: Permissions to write

- Each leader has an infinite supply of ballots



- The set of ballots of different leaders are disjoint

Ballots: Permissions to write

- Each leader has an infinite supply of ballots



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

$$\langle seq_no, LId \rangle$$

Acceptors

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

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

A mapping is forever...

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

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

Acceptor

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

Invariants

A1. An acceptor can only adopt strictly increasing ballot numbers

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

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

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

Acceptor

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

Invariants

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

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



Commander

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



Commander invariants

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



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



Commander invariants

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



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

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

Commander

Must enforce

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



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



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

Commander

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

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

Commander

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

Notify the leader and exit



Scout

- Before spawning commanders for ballot b, leader invokes a scout
- Scouts read the Paxos memory to help leaders propose mappings that satisfy C2.
- A scout's mission has two possible outcomes:
  - success: the leader learns that the proposed ballot has been adopted by a majority of acceptors and receives all pvalues accepted by that majority
  - failure: the leader learns that b may no longer be acceptable to a majority of acceptors

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

Scout

Scout

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

Scout

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

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

Scout

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

Notify the leader and exit



Leader

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

How a leader enforces

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

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

How a leader enforces

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

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

How a leader enforces

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

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

process $Leader(acceptors, replicas)$

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

$spawn(Scout(self(), acceptors, ballot_num));$

for ever

switch $receive();$

case <propose, s, p > :

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

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

if $active$ then

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

end if

end if

end case

case <adopted, $ballot_num, pvals$ >

$proposals = proposals \oplus pmax(pvals)$

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

$active := \text{true}$

end case

case <preempted, r', λ' >

if $(r', \lambda') > ballot_num$ then

$active := \text{false};$

$ballot_num := (r' + 1, self());$

$spawn(Scout(self(), acceptors, ballot_num));$

end if

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

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

end case

end switch

end for

end process



Leader

Implementing State Machine Replication

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

The role of the leader

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

What if a new λ is elected?

- Since the new λ serves also as a replica (i.e., a learner) in all instances of consensus, it should know most of the commands that have already been chosen. For example, it might know commands for slots 1–10, 13, and 15.



What if a new λ is elected?

- Since the new λ serves also as a replica (i.e., a learner) in all instances of consensus, it should know most of the commands that have already been chosen. For example, it might know commands for slots 1-10, 13, and 15.
  - It executes phase 1 for slots 11, 12, and 14 and of all slots 16 and larger.

What if a new λ is elected?

- Since the new λ serves also as a replica (i.e., a learner) in all instances of consensus, it should know most of the commands that have already been chosen. For example, it might know commands for slots 1–10, 13, and 15.
  - It executes phase 1 for slots 11, 12, and 14 and of all slots 16 and larger.
  - λ may find that some value was already accepted for slots 14 and 16 and that slots 11, 12 and all slots after 16 have accepted no command.

What if a new λ is elected?

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

Stop-gap measures

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

To infinity, and beyond

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



Paxos and FLP

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

A Lower Bound

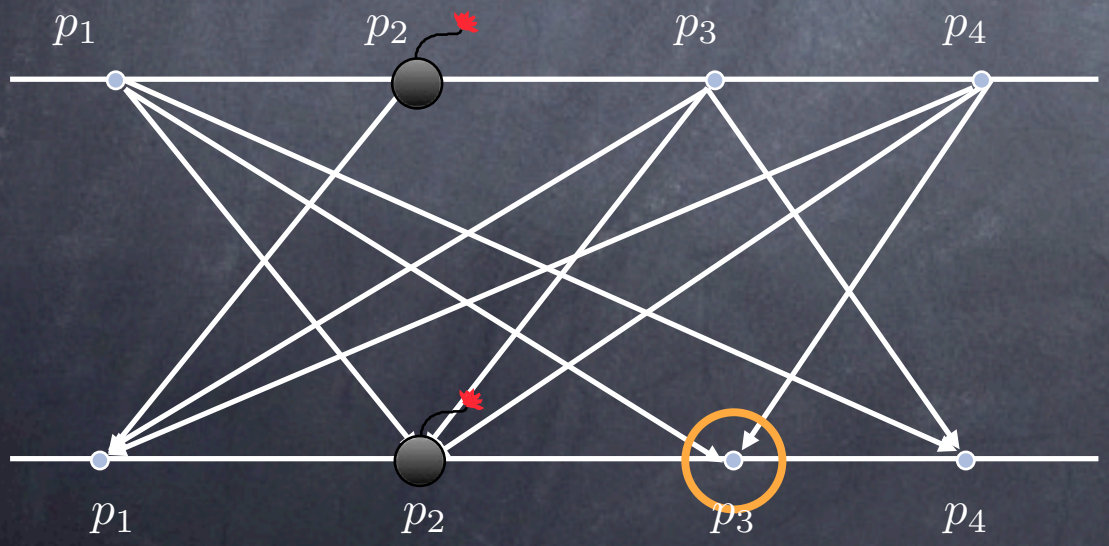
Theorem

There is no algorithm that solves the consensus problem in fewer than $f+1$ rounds in the presence of f crash failures, if $n \geq f+2$

We consider a special case $(f=1)$ to study the proof technique

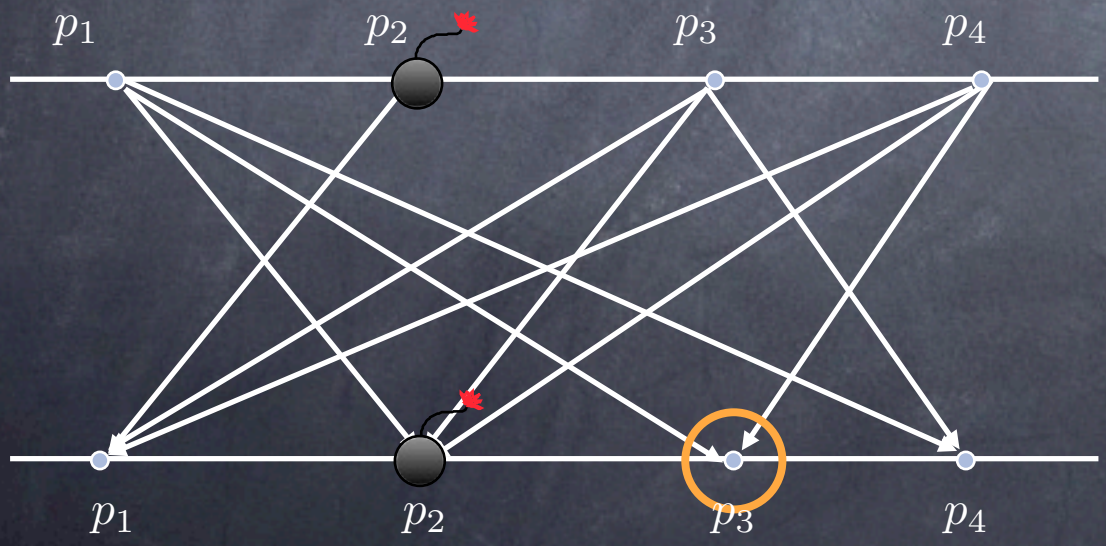
Views

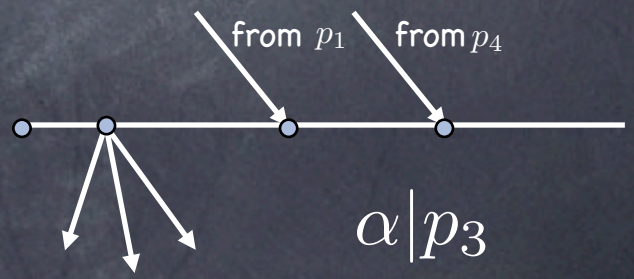
Let α be an execution. The view of process p_i in α, denoted by $\alpha|p_i$, is the subsequence of computation and message receive events that occur in p_i together with the state of p_i in the initial configuration of α



Views

Let α be an execution. The view of process p_i in α, denoted by $\alpha|p_i$, is the subsequence of computation and message receive events that occur in p_i together with the state of p_i in the initial configuration of α





Similarity

Definition Let α_1 and α_2 be two executions of consensus and let p_i be a correct process in both α_1 and α_2.

α_1 is similar to α_2 with respect to p_i, denoted $\alpha_1 \sim_{p_i} \alpha_2$ if

$$\alpha_1|p_i = \alpha_2|p_i$$

Similarity

Definition Let α_1 and α_2 be two executions of consensus and let p_i be a correct process in both α_1 and α_2.
α_1 is similar to α_2 with respect to p_i, denoted $\alpha_1 \sim_{p_i} \alpha_2$ if

$$\alpha_1|p_i = \alpha_2|p_i$$

Note If $\alpha_1 \sim_{p_i} \alpha_2$ then p_i decides the same value in both executions

Similarity

Definition Let α_1 and α_2 be two executions of consensus and let p_i be a correct process in both α_1 and α_2.
α_1 is similar to α_2 with respect to p_i, denoted $\alpha_1 \sim_{p_i} \alpha_2$ if

$$\alpha_1|p_i = \alpha_2|p_i$$

Note If $\alpha_1 \sim_{p_i} \alpha_2$ then p_i decides the same value in both executions

Lemma If $\alpha_1 \sim_{p_i} \alpha_2$ and p_i is correct, then dec(α_1) = dec(α_2)

Similarity

Definition Let α_1 and α_2 be two executions of consensus and let p_i be a correct process in both α_1 and α_2.
α_1 is similar to α_2 with respect to p_i, denoted $\alpha_1 \sim_{p_i} \alpha_2$ if

$$\alpha_1|p_i = \alpha_2|p_i$$

Note If $\alpha_1 \sim_{p_i} \alpha_2$ then p_i decides the same value in both executions

Lemma If $\alpha_1 \sim_{p_i} \alpha_2$ and p_i is correct, then dec(α_1) = dec(α_2)

The transitive closure of $\alpha_1 \sim_{p_i} \alpha_2$ is denoted $\alpha_1 \approx \alpha_2$.

We say that $\alpha_1 \approx \alpha_2$ if there exist executions $\beta_1, \beta_2, \ldots, \beta_{k+1}$ such that

$$\alpha_1 = \beta_1 \sim_{p_{i_1}} \beta_2 \sim_{p_{i_2}} \ldots, \sim_{p_{i_k}} \beta_{k+1} = \alpha_2$$

Similarity

Definition Let α_1 and α_2 be two executions of consensus and let p_i be a correct process in both α_1 and α_2.
α_1 is similar to α_2 with respect to p_i, denoted $\alpha_1 \sim_{p_i} \alpha_2$ if

$$\alpha_1|p_i = \alpha_2|p_i$$

Note If $\alpha_1 \sim_{p_i} \alpha_2$ then p_i decides the same value in both executions

Lemma If $\alpha_1 \sim_{p_i} \alpha_2$ and p_i is correct, then dec(α_1) = dec(α_2)

The transitive closure of $\alpha_1 \sim_{p_i} \alpha_2$ is denoted $\alpha_1 \approx \alpha_2$.

We say that $\alpha_1 \approx \alpha_2$ if there exist executions $\beta_1, \beta_2, \ldots, \beta_{k+1}$ such that

$$\alpha_1 = \beta_1 \sim_{p_{i_1}} \beta_2 \sim_{p_{i_2}} \ldots, \sim_{p_{i_k}} \beta_{k+1} = \alpha_2$$

Lemma If $\alpha_1 \approx \alpha_2$ then dec(α_1) = dec(α_2)

Single-Failure Case

There is no algorithm that solves consensus in fewer than two rounds in the presence of one crash failure, if $n \geq 3$

The Idea

By contradiction

- Consider a one-round execution in which each process proposes 0. What is the decision value?
- Consider another one-round execution in which each process proposes 1. What is the decision value?
- Show that there is a chain of similar executions that relate the two executions.

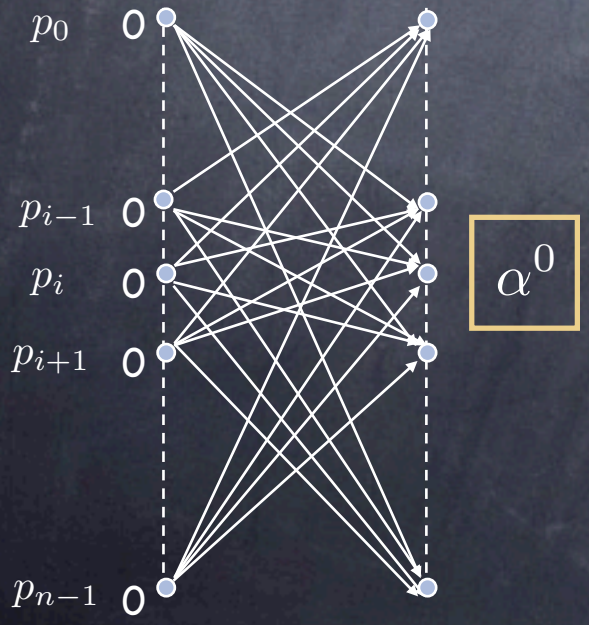
So what?

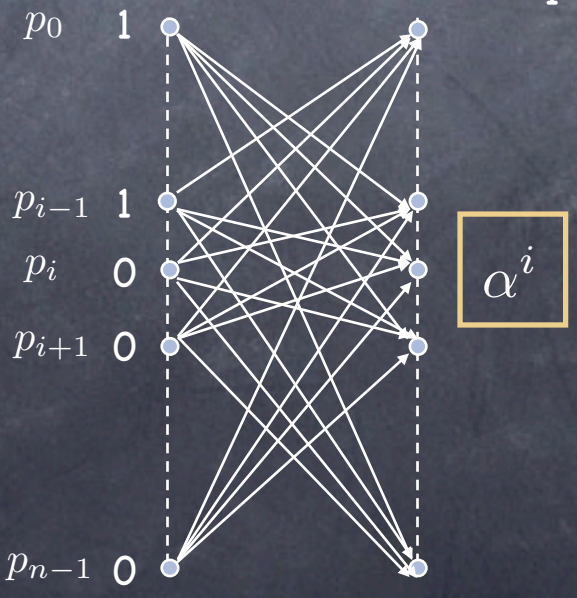
α^is

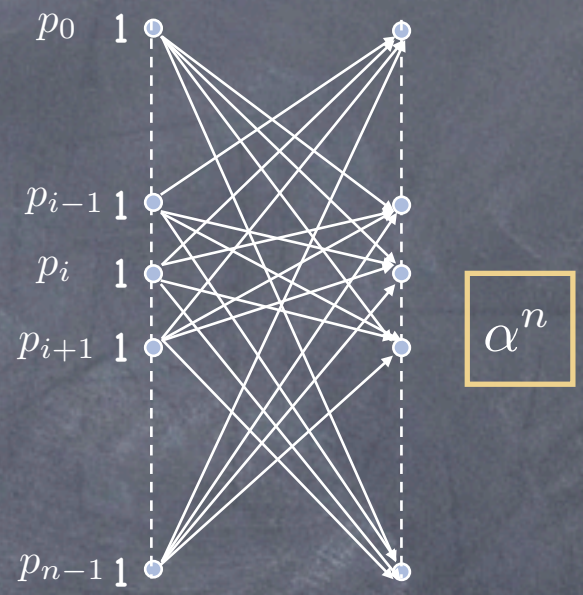
Definition

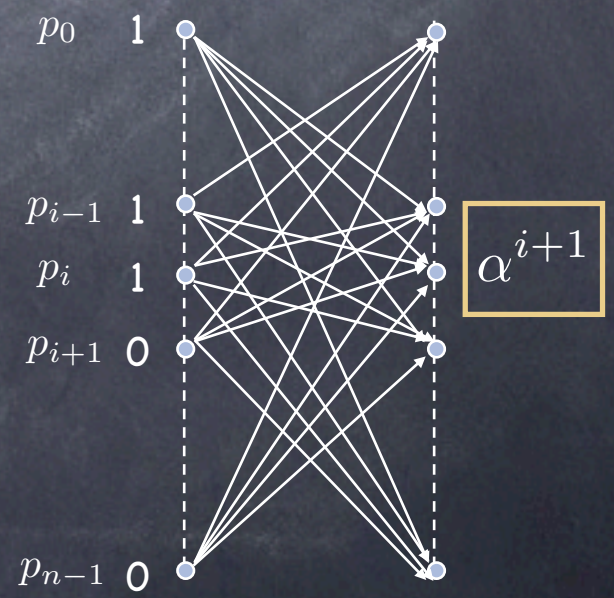
α^i is the execution of the algorithm in which

- no failures occur
- only processes $p_0, \ldots, p_{i-1}$ propose 1









Claim: Adjacent α^is are similar!

If so, then $\alpha^0 \sim \alpha^1 \sim \ldots \sim \alpha^{n-1} \sim \alpha^n$

and $\alpha^0 \approx \alpha^n$

Because they are similar, α^0 and α^n should decide the same value – but by Validity of Consensus, they cannot!

Contradiction!

How to go from α^i to α^{i+1}

Starting from α^i, we build a set of executions α^i_j where $0 \leq j \leq n-1$ as follows:

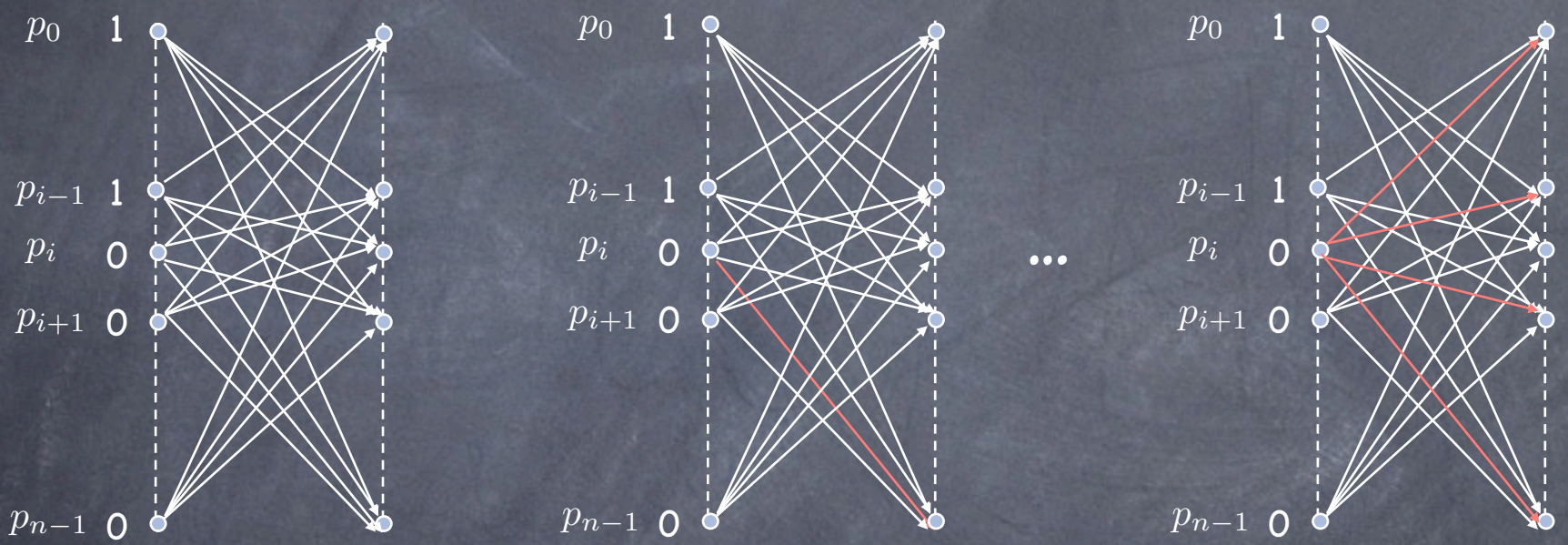
α^i_j is obtained from α^i after removing the messages that p_i sends to the j-th highest numbered processors (excluding itself)

So, α^i_0 removes no message: it is the same as α^i

α^i_1 removes the one message that p_i sends to the one process with the highest id

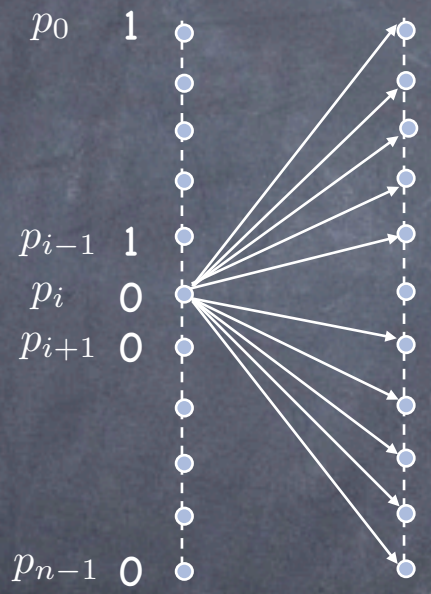
α^i_2 removes the two messages that p_i sends to the two processes with the highest id... and so on

The executions



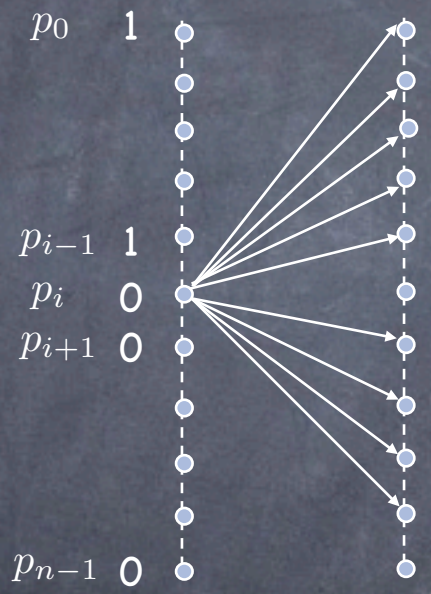
α_0^i α_1^i α_{n-1}^i

Indistinguishability



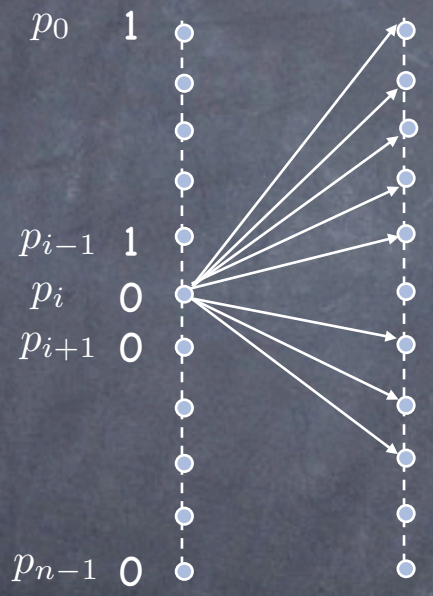
$$\alpha^i$$
$$\|\|$$
$$\alpha_0^i$$

Indistinguishability



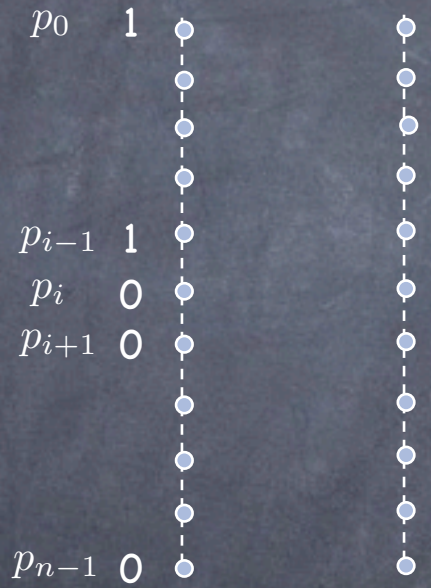
$$\alpha^i \wr \alpha_1^i$$

Indistinguishability



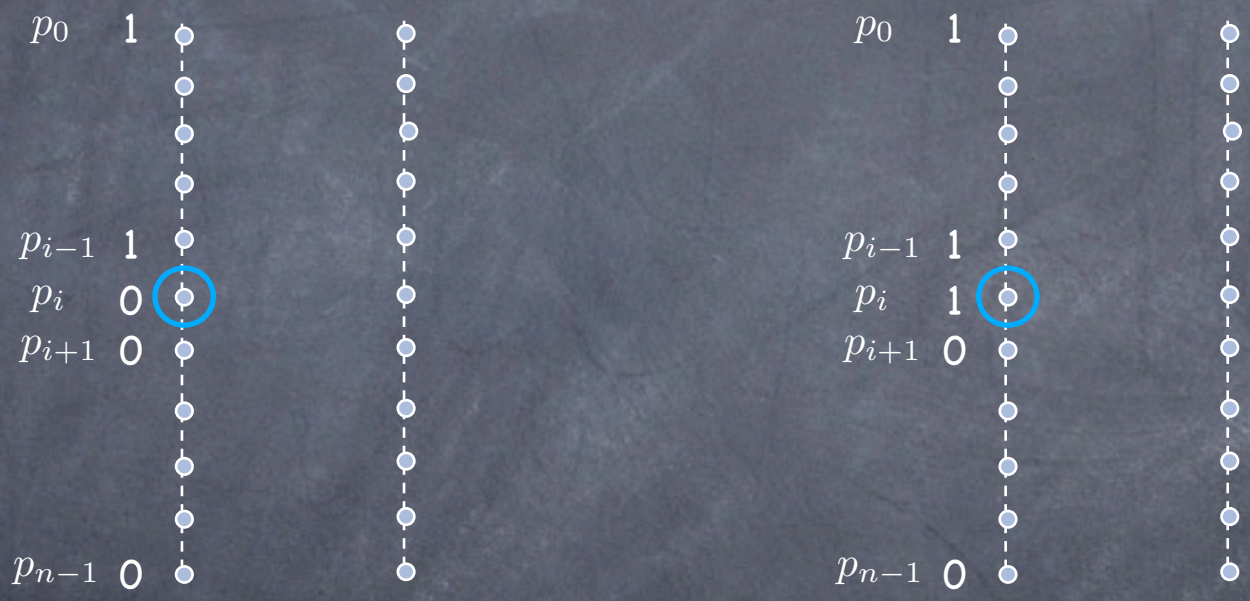
$$\alpha^i \approx \alpha_2^i$$

Indistinguishability



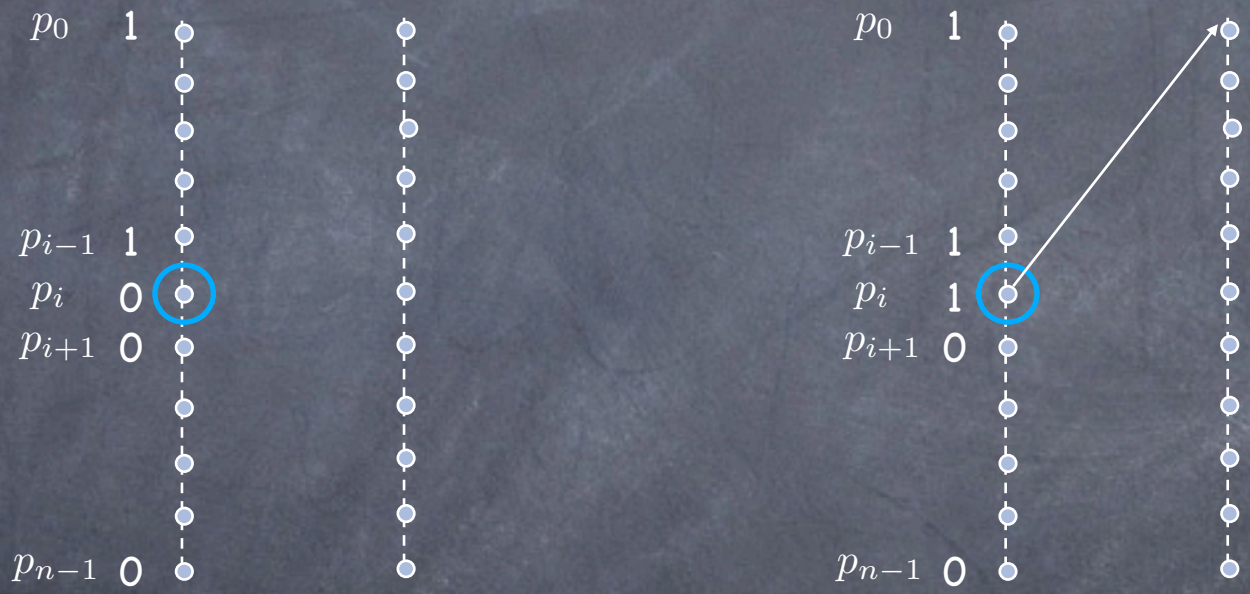
$$\alpha^i \approx \alpha^i_{n-1}$$

Indistinguishability



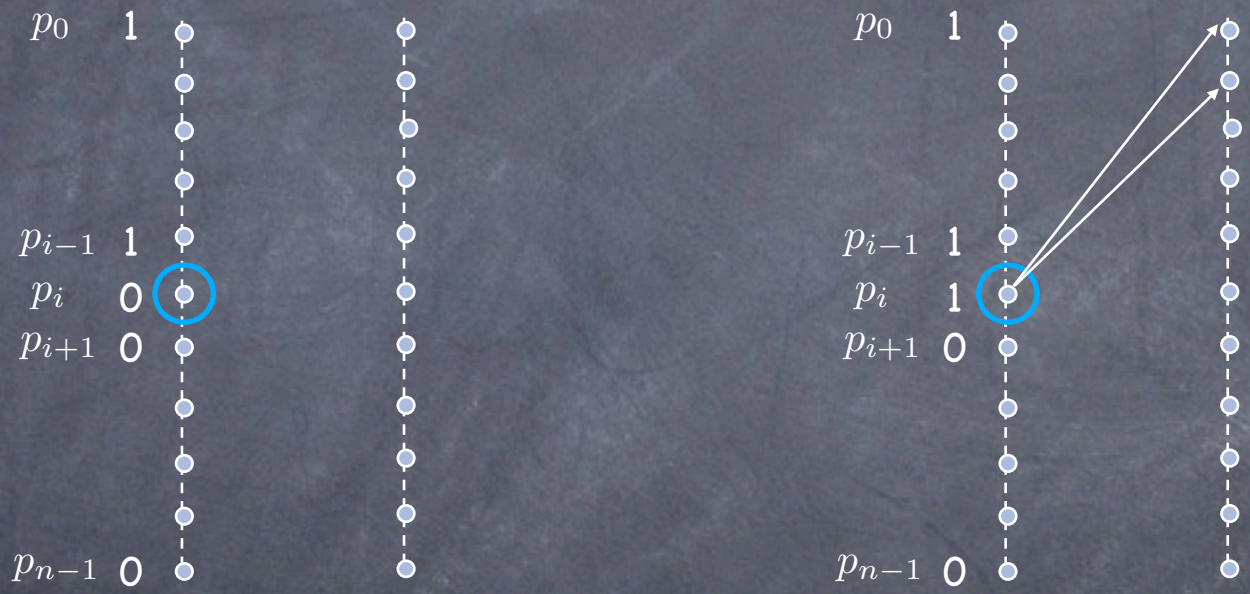
$$\alpha^i \approx \alpha^i_{n-1} \qquad \approx \qquad \beta^i_{n-1}$$

Indistinguishability



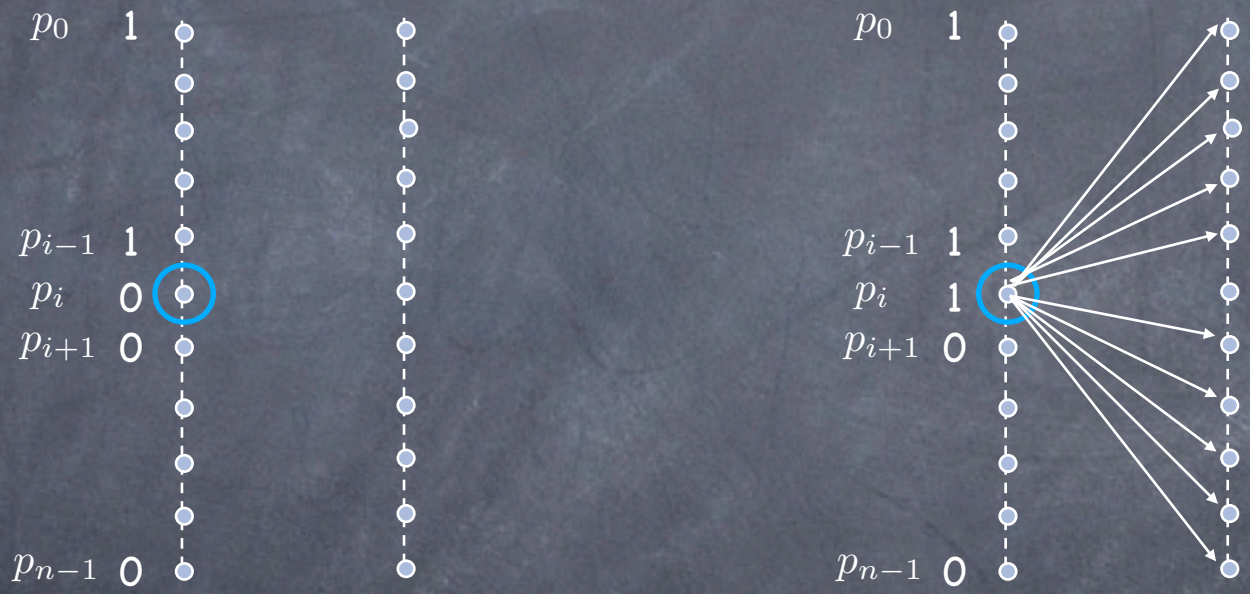
$$\alpha^i \approx \alpha^i_{n-1} \qquad \approx \qquad \beta^i_{n-2}$$

Indistinguishability



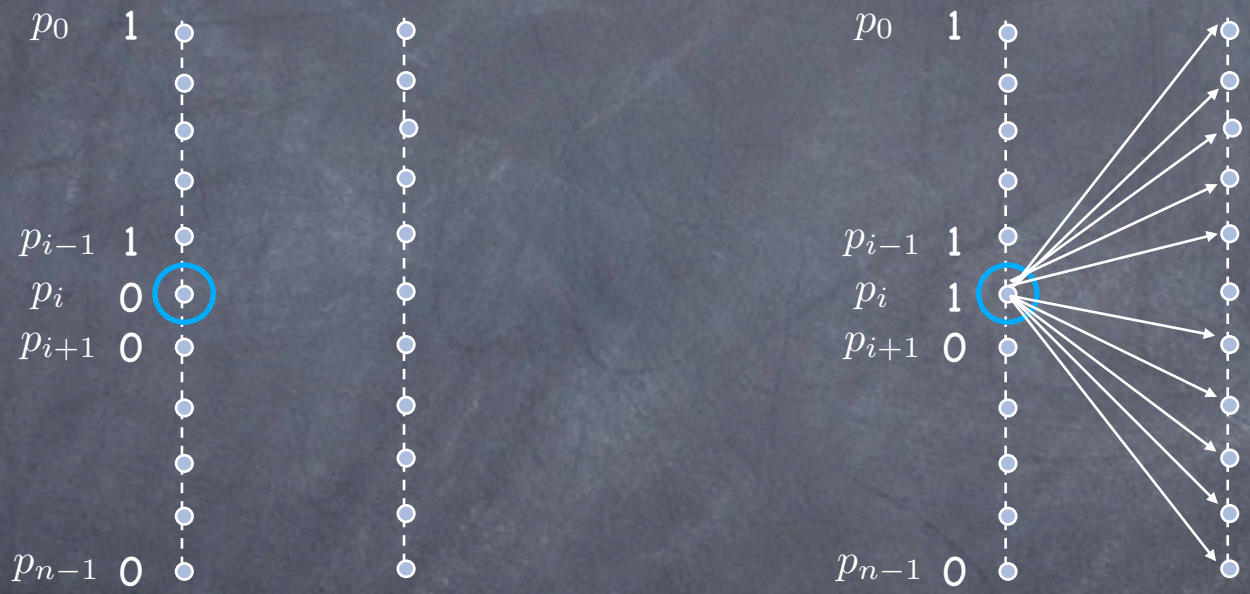
$$\alpha^i \approx \alpha^i_{n-1} \qquad \approx \qquad \beta^i_{n-3}$$

Indistinguishability



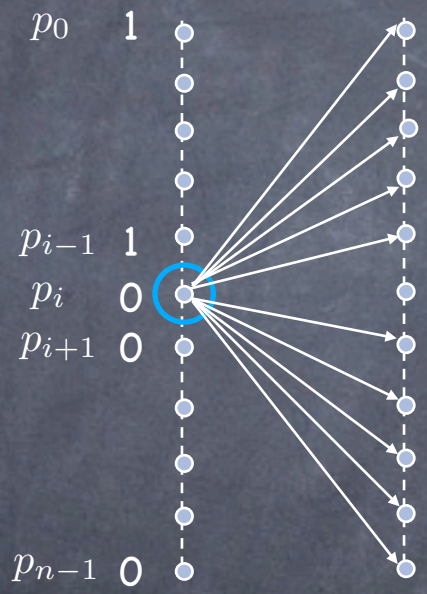
$$\alpha^i \wr \alpha^i_{n-1} \approx \beta^i_0$$

Indistinguishability



$$\alpha^i \wr \alpha^i_{n-1} \approx \alpha^{i+1} \wr \beta^i_0$$

Indistinguishability



α^i $\approx$ α^{i+1}

