MADS

Emmanuelle Anceaume

Lesson 3: Consensus in Asynchronous Environments

http://people.irisa.fr/Emmanuelle.Anceaume/

Terminology

- We consider distributed systems where processes can communicate and synchronize by exchanging messages (message-passing model).
- The system is composed of n processes usually denoted $\Pi = \{p_1, \ldots, p_n\}$.
- The system is asynchronous because there exists no bound :
  - neither on the relative speeds of processes
  - nor on the communications speed.

Why such a model ?

- It is extremely simple
- If a problem can be solved in asynchronous systems, it can be solved in more constrained model (like synchronous systems or partially synchronous systems)
- A solution to a problem P in this model can always be used directly in a more demanding model M
  - It will then benefit from the good properties exhibited by model M
  - While at the same time being robust enough to tolerate violations of the properties exhibited by model M

Consensus
Informal specification

- In this problem processes are trying to reach a consensus.
- Each process initially proposes a value v taken from a given set of value V.
- At the end of the protocol, all processes agree on a single value, called the decided value, or decision.
- This value must have been proposed by one of the processes.

Consensus Specification

Each process has an initial value and at the end of the protocol, the following must hold :

- Termination : All correct processes must eventually decide a value.
- Integrity : At most one decision per process.
- Agreement : All processes that decide (correct or not) must decide the same value.
- Validity : The value decided by a process must have been initially proposed.

A simple consensus algorithm

```
 propose(v_i) // algorithm run by process p_i
{
  local_state = v_i
  send (i, v_i) to all processes
  wait until n-1 different messages of the
     form (j, v_j) have been received
  d_i ← δ((1, v_1), ..., (n, v_n) ∪ (i, v_i))
  return decide(d_i)
}
```

Theorem (FLP impossibility result)

There exists no deterministic algorithm that solves the binary consensus problem in the presence of even if a single faulty process [a]

a. M. Fischer, N. Lynch, and M. Paterson. « Impossibility of distributed consensus with one faulty process ». Journal of the ACM, 32(2) : 374-382, 1985

Binary consensus : processes have solely two possible input values « 0 » and « 1 »

Asynchronous Broadcast System

An asynchronous broadcast system consists of a set of processes $1, \ldots, n$ and a broadcast channel.

- Each process p_i has a one-bit input register x_{p_i}, and output register y_{p_i} with values in $\{0, 1, b\}$
- The state of process p_i comprises the value of x_{p_i}, the value of y_{p_i} (and its program counter, and its internal storage...)
- Initial state of p_i : $x_{p_i} = 0$ or $x_{p_i} = 1$ and $y_{p_i} = b$
- Decision states : $y_{p_i} = 0$ or $y_{p_i} = 1$
- Transition function
  - deterministic
  - cannot change the decision value (y_{p_i} is writable only once)

Processes communicate by exchanging messages

- Processes communicate by sending messages
- A message is a pair (p, m) where p is the recipient of m and m is some message value.
- The message system maintains a message buffer of messages that have been sent but not yet delivered
- It provides two operations
  - send(p, m) : places (p, m) in the message buffer
  - receive(p) :
    - delete some message (p, m) from the buffer and returns m to p
    - we say that (p, m) is delivered
    - or return null and leave the buffer unchanged

Processes communicate by exchanging messages

Thus the message system acts in a non deterministic way

- receive(p) can return null even though a message (p, m) belongs to the buffer
- however if queried infinitely many times, every message (p, m) is eventually delivered

Configuration

- A configuration (or global state) of the system consists of the internal state of each process and the content of the message buffer
  - $C = (s, \mathcal{B})$ with $s = (s_1, s_2, \ldots, s_n)$
- An initial configuration is a configuration in which each process starts at an initial state and the message buffer is empty

Step

The system moves from one configuration to the next one by a step. A step executed by process p consists of the following set of actions :

- Let $C = (s, \mathcal{B})$ be a configuration
- p performs receive(p) on the message buffer in $\mathcal{B}$ of C
- p delivers a value $m \in \{M, null\}$
- based on its local state in C and m, p enters a new state and sends a finite number of messages
- $C.e$ denotes the resulting configuration. We say that e can be applied to C

Thus the only way the system state may change is by some process receiving a message

Step (cont'd)

Since processes are deterministic

- the step is completely determined by C and $e = (p, m)$
- in the following the step e is also called an event

Since the receive operation is non-deterministic

- there are many different possible execution from an initial configuration
- to show that some algorithm solves the consensus problem one has to show that for any possible execution, the termination, agreement, integrity and validity must hold

Decision value

- A configuration C has decision value v if some process p is in a decision state (i.e. $y_p = 0$ or $y_p = 1$)

Run

- A run is a sequence of steps taken by the processes from an initial global state of the system
- Non faulty processes take infinitely many steps in a run. Otherwise the process is faulty.
- A run is admissible provided that at most one process is faulty and all messages have been delivered
- A run is decidable provided that some process eventually decides
- A consensus protocol is correct if every admissible run is a deciding run

Adversary

When designing fault-tolerant algorithms, we often assume the presence of an adversary

- It has some control on the behavior of the system
- It knows the content of all sent messages
- It knows the local state of each process
  - → it can select the next process to take a step
  - → It can select the message the process will receive
- However
  - It cannot prevent a message from being eventually received
  - It cannot make more than one processe crash

Correct consensus protocol P

Theorem

No correct consensus protocol exists

- The idea behind the theorem is to show that there exists some admissible run which is not deciding : no process ever decides
- That's enough to show that there is just one initial configuration in which a given protocol will not work because starting in that configuration can never be ruled out.

All the following slides have been made in collaboration with Frédéric Tronel (Centrale-Supélec).

Proof of the theorem

The proof proceeds in two steps :

- the first step shows that there are initial configurations in which the decision is not pre-determined
- the second step shows that one can always find configurations in which processes cannot decide

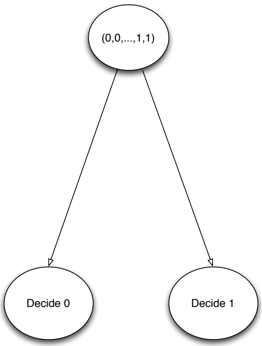
Say differently : for any consensus protocol, an adversary tries to steer the execution away from a deciding one

Valence of configurations

First step of the proof :

- It always exists some initial configuration in which the decision is impossible to predict

- A decision results from the protocol execution
- Completely depends on the asynchrony of the system
  - messages receipt out of order
  - arbitrary delays and potential failure



Valence of configurations

Let C be any configuration. Let V be the set of decision values of configurations reachable from C

1. If $V = \{0\}$ then C is said to be **univalent** or 0-**valent**
2. If $V = \{1\}$ then C is said to be **univalent** or 1-**valent**
3. If $V = \{0, 1\}$ then C is said to be **bivalent**.

- A 0-valent configuration necessarily leads to decision 0
- A 1-valent configuration necessarily leads to decision 1
- A bivalent configuration is a configuration from which we cannot say whether the decision will be 0 or 1. This is an « undecided » configuration

Lemma

Any consensus protocol that tolerates at least one faulty process has at least one bivalent initial configuration.

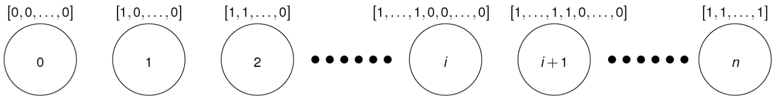
Proof of Lemma 1

Proof : By contradiction. Suppose that all the initial configurations are univalent (i.e. are completely determined by the set of initial values) By the validity property,

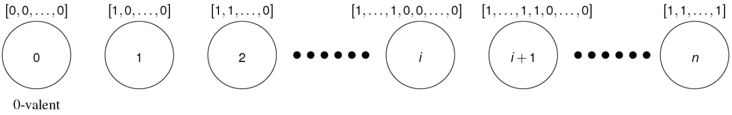
- initial configurations such that 0 is decided
- initial configurations such that 1 is decided

We can order initial configurations in a chain of configurations, where two configurations are next to each other if they differ by only one value

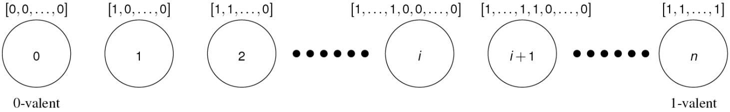
$\rightarrow$ the diference between two adjacent configurations is the starting value of a one process



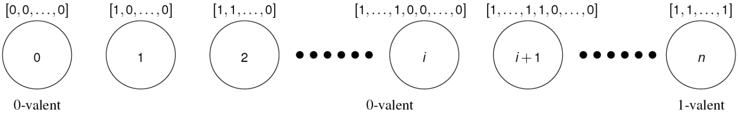
Proof of Lemma 1



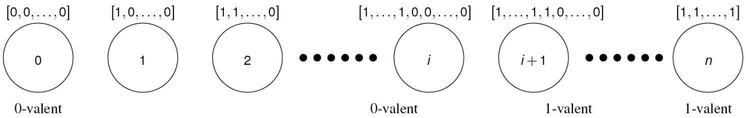
Proof of Lemma 1



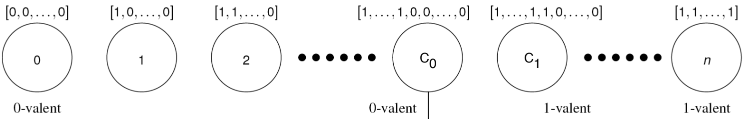
Proof of Lemma 1



Proof of Lemma 1



Proof of Lemma 1

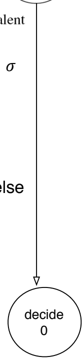


In σ process p_i does not take any steps
(i.e does not receive nor send messages).

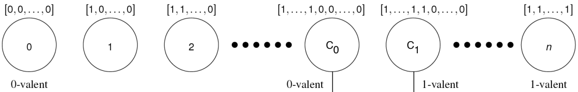
So its initial value cannot be observed by someone else

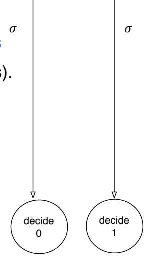
All processes must eventually decide
(1-failure tolerant protocol)

Since C_0 is 0-valent the decision state is 0



Proof of Lemma 1





In σ process p_i does not take any steps
(i.e does not receive nor send messages).

So its initial value cannot be observed

All processes must eventually decide
(1-failure tolerant protocol)

Since C_0 is 0-valent the decision state is 0

Run σ can be made from C1 too
since no process has ever heard about p_i

Thus all the processes (except p_i) should reach
the "0" deciding state

This is a contradiction since
by assumption C1 is a "1"-valent configuration

Proof of Lemma 1

- So this results contradicts the fact that the outcome of the consensus algorithm is uniquely predetermined by the initial configurations
- C_0 can lead to a "0" decision state or to a "1"-decision state, depending on the pattern of failures and events

Initial bivalent configuration

Any consensus protocol that tolerates at least one faulty process has at least one bivalent initial configuration

Second step of the proof

The intuitive argument :

- Start from a bivalent configuration C
- Let some event $e = (p, m)$ which is applicable to C
- Delay arbitrarily long event e
- There will be one configuration in which p makes step e that ends up in a bivalent configuration

If you can do that infinitely many times then the protocol never terminates

Lemma 2

A little bit more formally . . .

Bivalent extension Lemma

Let C be a bivalent configuration of the protocol, and let $e = (p, m)$ be an event that is applicable to C.

Let $\mathcal{C}$ be the set of configurations reachable from C without doing e and without failing any process.

Let $\mathcal{D}$ be the set of configurations of the form $C'.e$ where $C' \in \mathcal{C}$.

Then $\mathcal{D}$ contains a bivalent configuration.

- Note that step e is always applicable in $\mathcal{C}$ since
  - e is applicable to C
  - $\mathcal{C}$ is the set of configurations reachable from C
  - and messages can be delayed arbitrarily long

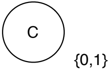
Proof of the bivalent extension lemma

The proof is by contradiction

1. We assume that $\mathcal{D}$ contains only univalent configurations
2. We prove that $\mathcal{D}$ contains both 0-valent and 1-valent configurations D_0 and D_1
3. We prove that $\mathcal{C}$ contains two configurations C_0 and C_1 that resp. lead to D_0 and D_1 by applying step e
4. We derive a contradiction

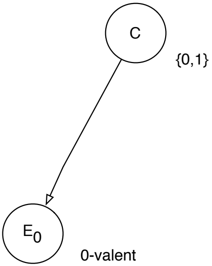
Proof of the bivalent extension lemma

We start from a bivalent configuration C (C exists by the first lemma)



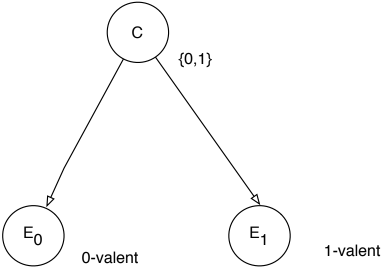
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

There must exist a 0-valent configuration E_0 reachable from C (recall that C is bivalent)



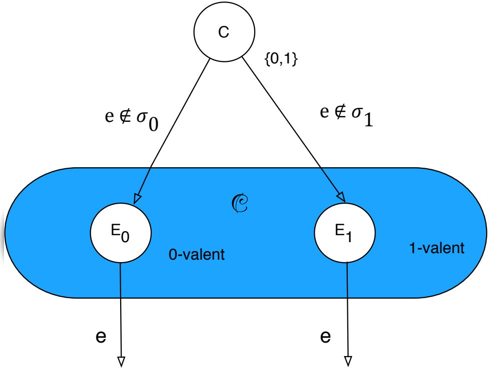
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

There must exist a 1-valent configuration E_1 reachable from C (recall that C is bivalent)



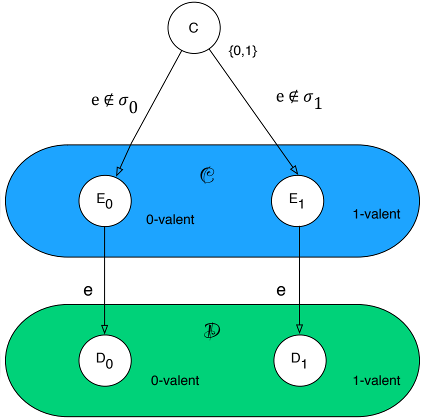
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

Case 1 : If E_i belongs to $\mathcal{C}$ (that is step e is not applied along σ_i) then e can be applied to E_i



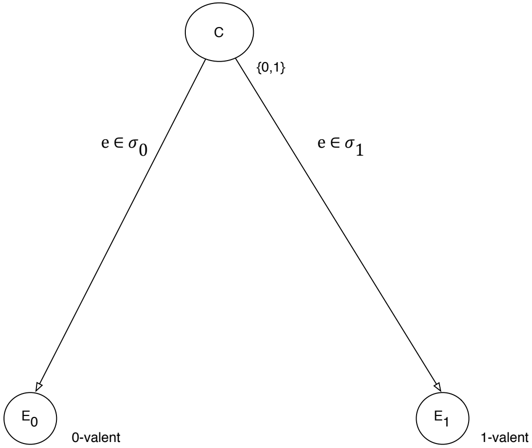
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

Let D_i be the configuration reached from E_i by application of step e. D_i is i-valent since D_i belongs to $\mathcal{D}$ and by assumption $\mathcal{D}$ contains only univalent configurations.



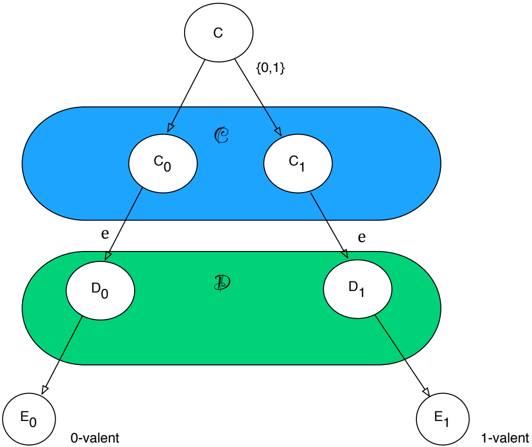
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

case 2 : E_i does not belong to $\mathcal{C}$ (that is step e has been applied along σ_i).



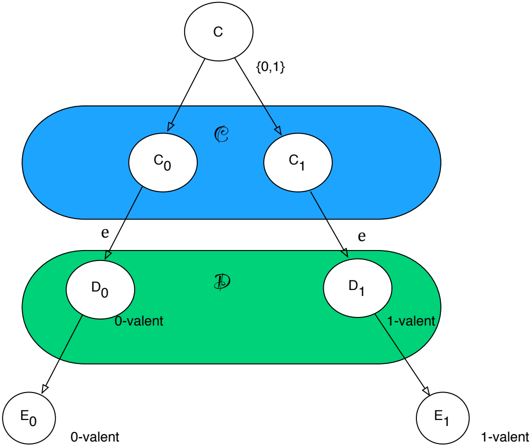
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

Thus there is a configuration $C_i \in \mathcal{C}$ such that step e is applied to C_i and $D_i = C_i.e$, with $D_i \mathcal{D}$.



$\mathcal{D}$ contains both 0-valent and 1-valent configurations

By assumption $\mathcal{D}$ contains only univalent configurations. Thus D_i is univalent and since D_i lead to E_i which is i-valent, D_i is i-valent.



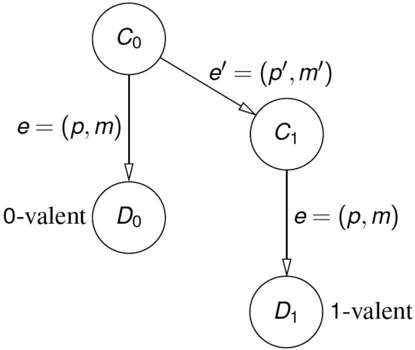
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

So far we have shown that $\mathcal{D}$ contains both 0-valent and 1-valent configurations.

- Definition :
  - Configurations C_0 and C_1 are neighbor if one results from the other by application of a single step.

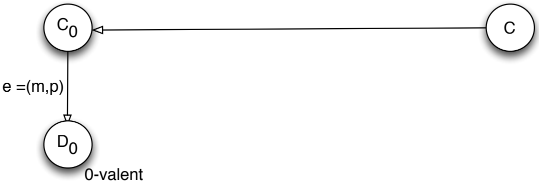
We want to prove that $\mathcal{C}$ contains two neighbor configurations C_0 and C_1 that lead to D_0 and D_1 in $\mathcal{D}$

What do we want to prove ?



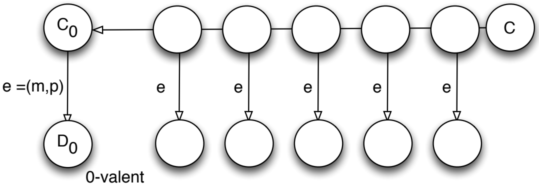
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Let C be a bivalent configuration, and C_0 reachable from C that leads to D_0 a 0-valent configuration of $\mathcal{D}$ by applying step e

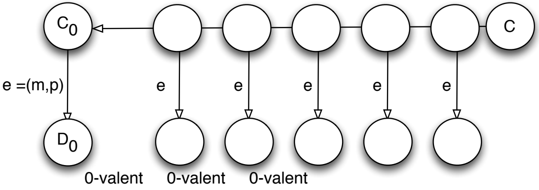


Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Since step e is applicable from C then one can apply this step all along the path from C to C_0

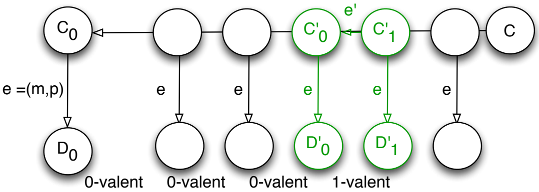


All these configurations belong to $\mathcal{D}$. Hence they are all univalent.
Some of them can be 0-valent as is D_0



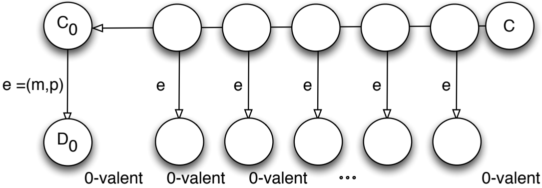
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

If one of them is 1-valent, we are done. We have found the hook we were looking for.



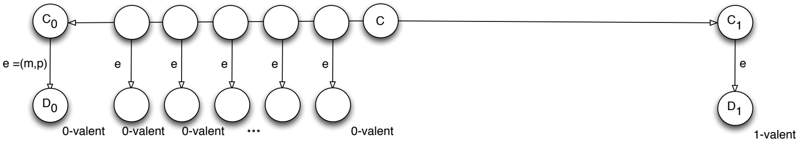
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Otherwise all of them of 0-valent.



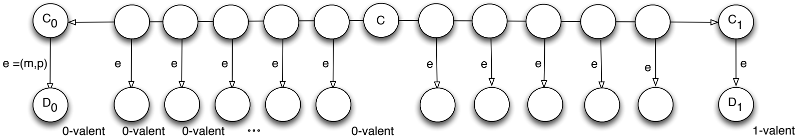
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Then consider C_1 a configuration in $\mathcal{C}$ reachable from C that leads to D_1 a 1-valent configuration in $\mathcal{D}$ by applying step e



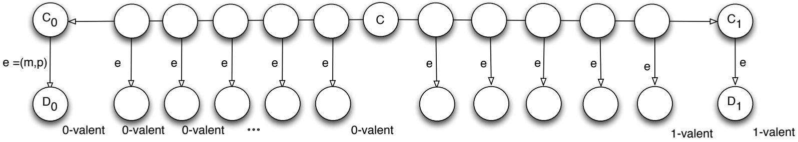
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Since step e is applicable from C then one can apply this step all along the path from C to C_1



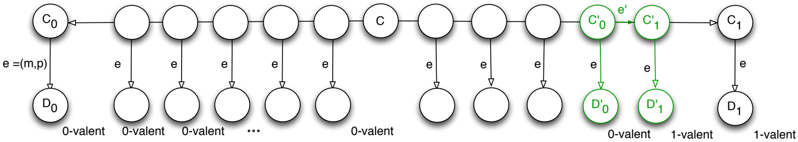
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

All these configurations belong to $\mathcal{D}$. Hence they are all univalent. Some of them can be 1-valent as is D_1



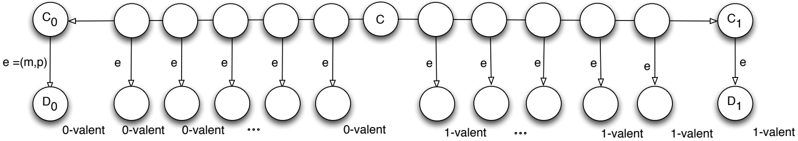
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

If one of them is 0-valent, we are done. We have found the hook we were looking for.



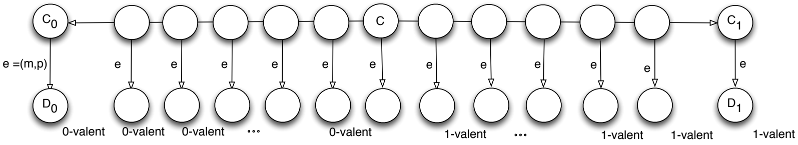
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Otherwise all of them of 1-valent.



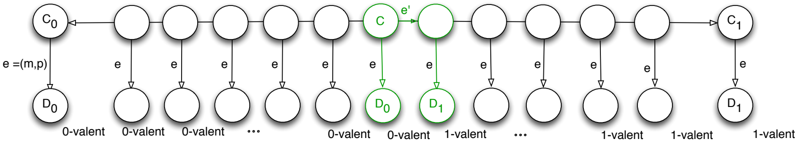
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

The hook we are looking for is located at configuration C. Let us apply step e to C



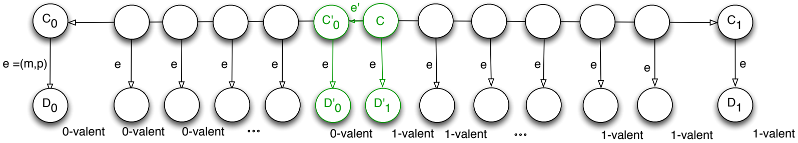
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Either this configuration of $\mathcal{D}$ is 0-valent, and thus we can identify the hook we were looking for

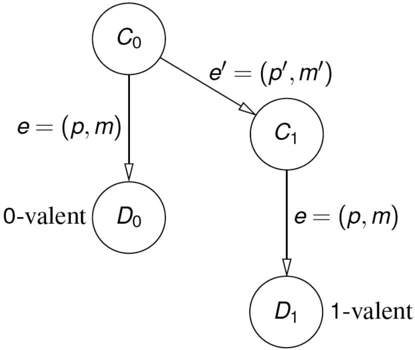


Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

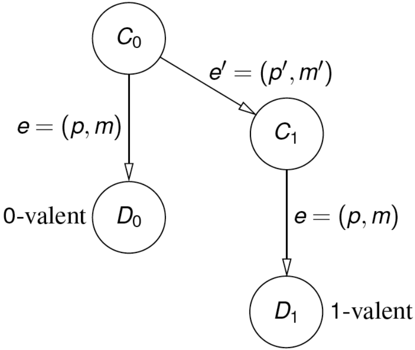
Or this configuration of $\mathcal{D}$ is 1-valent, and thus we can identify the hook we were looking for



Where have we been so far ?



Where have we been so far ?



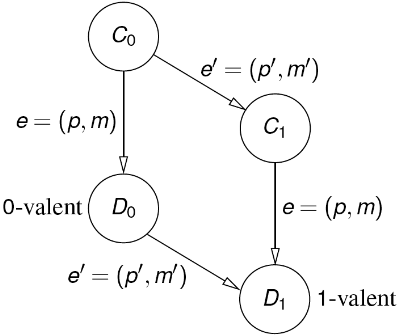
We are almost done. We need to consider two cases :

1. either $p \neq p'$
2. or $p = p'$

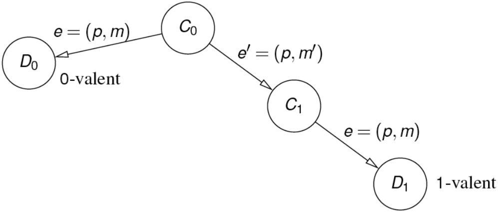
$p \neq p'$

- Since p is different from p' then steps e and e' do not interact
- Steps e' can be applied to configuration D_0
- Thus $D_0.e' = D_1$ which closes the diamond

We get a contradiction since a 0-valent configuration cannot lead to a 1-valent configuration.



$p = p'$

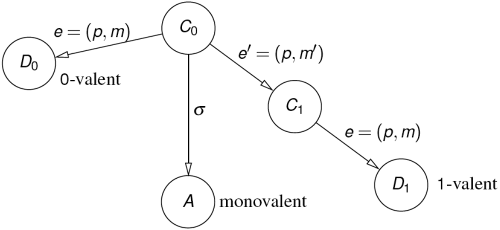


$p = p'$

Let σ be an execution that can be applied to C_0 such that

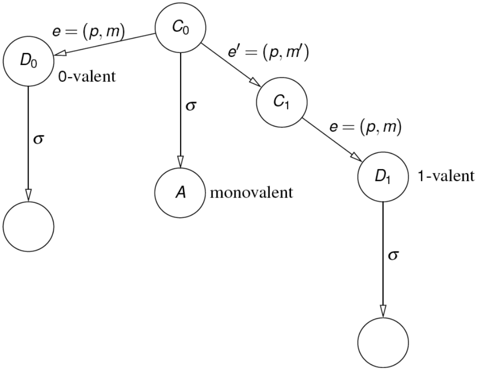
1. All the processes decide
2. Except p that does not make any step in σ (the protocol tolerates one crash thus it must allow $n-1$ processes to decide)

- Let $A = C_0.\sigma$ be such a decision configuration
- By the validity property of the consensus protocol, configuration A must be univalent

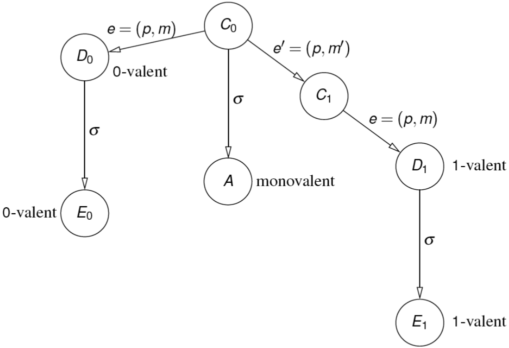


$p = p'$

Since p takes no step in σ, σ can be applied to D_0 and to D_1

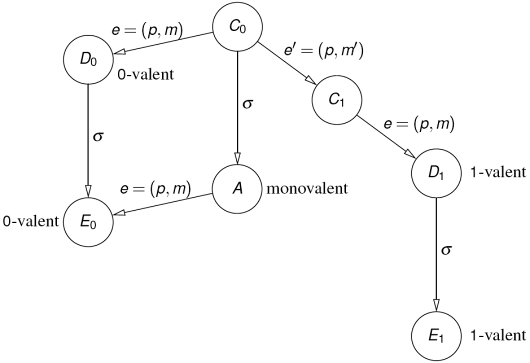


Leading to a 0-valent configuration E_0 and 1-valent configuration E_1

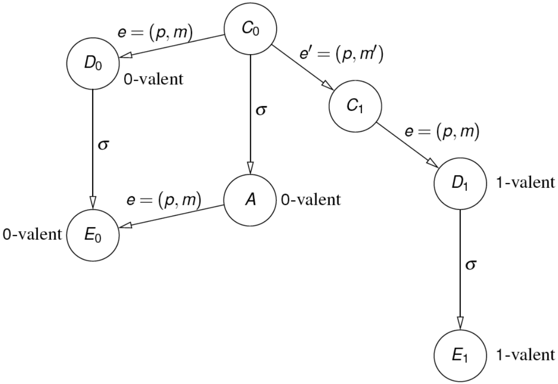


$p = p'$

Now the adversary allows p to make its step e from configuration A. This leads to configuration $E_0 = A.e$ by applying the same argument as before.

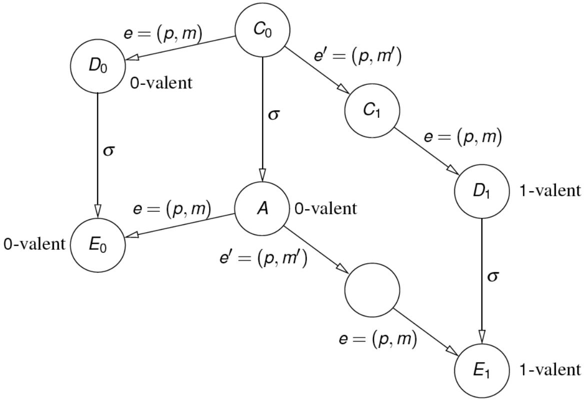


Thus configuration A must be 0-valent

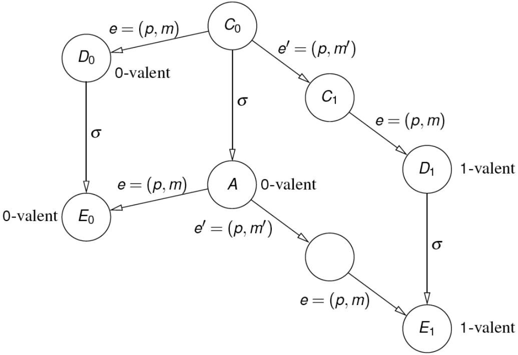


$p = p'$

Both e' and e can be applied to configuration A and leads to $E_1 = A.e'.e$.



Thus A must be 1-valent. But A is 0-valent. A contradiction



Bridging it all together

- The final step amounts to showing that any deciding run also allows the construction of an infinite non-deciding one
- By applying the bivalent extension lemma, we can always extend a finite execution made up of bivalent configurations with another execution also made up of bivalent configurations with the step of a given process.
- We can repeat this step with each process infinitely often
- But no process will ever decide.

FLP impossibility result

- This theorem is so far the most fundamental one for the field of fault-tolerant distributed computing
- This work has received the Edsger W. Dijkstra Prize in Distributed Computing prize in 2001.

A randomized consensus algorithm

- Soon after the FLP impossibility results appeared, people try to find a way to circumvent it.
- Ben-Or gave the first the first randomized algorithm that solves consensus with probability 1
- Asynchronous message-passing system with $f \leq n/2$ crash failures (n number of processes and f max. number of processes that may crash

Model of the system

- Set of n processes
- At most $f < n/2$ processes may crash (may stop to take steps)
- Asynchronous environment
- Communication channel is reliable
- Each process has access to a coin : when a process tosses its coin, it obtains 0 or 1 with probability $1/2$.

Step

A step of execution is as follows :

- Receipt of a message
- Tosses a coin (optional)
- Changing its state
- Sending a message to all processes

Adversary

When designing fault-tolerant algorithms, we often assume the presence of an adversary

- It has some control on the behavior of the system
- It knows the content of all sent messages
- It knows the local state of each process
  - $\rightarrow$ it can select the next process to take a step
  - $\rightarrow$ It can select the message the process will receive
- However
  - It cannot prevent a message from being eventually received
  - It cannot make more than f processes crash

The randomized consensus problem

Every process has some initial value $v_p \in \{0, 1\}$, and must decide on a value such that the following properties hold :

- Agreement : No two processes decide differently
- Validity : If any process decides v, then v is the initial value of some process
- Termination : With probability 1, every correct process eventually decides

Note that Agreement and Validity are safety properties and Termination is a liveness property.

Ben Or's randomized consensus algorithm

- First algorithm [1] to achieve consensus with probabilistic termination in an asynchronous model
- The algorithm is correct if no more than f crash occur with $f < n/2$
- Expected time to decide : $O(2^{2n})$ rounds

1. M. Ben-Or. « Another advantage of free choice : Completely asynchronous agreement protocols (extended abstract) ». In Proc. of the 2nd annual ACM Symposium on Principles of Distributed Computing Systems (PODCâĂŹ83), pages 27âĂŞ30, 1983.

Ben-Or's randomized consensus algorithm

- Operates in rounds, each round has two phases :
  - Report phase : each process transmits its value, and waits to hear from other processes
  - Decision phase : if majority found, take its value ; otherwise flip a coin to change the local value
- The idea :
  - If enough processes detect the majority, then decide
  - If I know that someone detected majority, then switch to the majority value
  - Otherwise, flip a coin ; eventually, a majority of correct processes will flip in the same way

Ben-Or's randomized consensus algorithm

Every process p_i executes the following algorithm :

```
procedure consensus(v_i)
{
 x ← v_i // p_i's current estimate of the decision value
 k = 0
 while true do
   k ← k + 1 // k is the current round number
   send (R, k, x) to all processes

   wait for (R, k, *) msgs from n − f processes //"*" in {0,1}
   if received more than n/2 (R, k, v) with the same v then
     send(P, k, v) to all processes
   else
     send(P, k, ?) to all processes

   wait for (P, k, *) msgs from n − f processes //"*" in {0,1,?}
   if received at least f + 1 (P, k, *) with the same v ≠? then
     decide v
   if received at least one (P, k, v) with v ≠? then
      x ← v
   else
      x ← 0 or 1 randomly // toss coin
}
```

Ben-Or's randomized consensus algorithm

```
procedure consensus(v_i)
{
 x ← v_i // p_i's current estimate of the decision value
 k = 0
 while true do
   k ← k+1 // k is the current round number
   send (R,k,x) to all processes

   wait for (R,k,*) msgs from n−f processes //"*" in {0,1}
   if received more than n/2 (R,k,v) with the same v then
     send(P,k,v) to all processes
   else
     send(P,k,?) to all processes

   wait for (P,k,*) msgs from n−f processes //"*" in {0,1,?}
   if received at least f+1 (P,k,*) with the same v ≠? then
     decide v
   if received at least one (P,k,v) with v ≠? then
      x ← v
   else
      x ← 0 or 1 randomly // toss coin
```

At the end of the first phase a process
- proposes v if received a strict majority of reports v
- proposes ? otherwise
if v=1 is proposed then v=0 cannot be proposed

Ben-Or's randomized consensus algorithm

```
procedure consensus(v_i)
{
 x ← v_i // p_i's current estimate of the decision value
 k = 0
 while true do
   k ← k + 1 // k is the current round number
   send (R, k, x) to all processes

   wait for (R, k, *) msgs from n − f processes //"*" in {0,1}
   if received more than n/2 (R, k, v) with the same v then
     send(P, k, v) to all processes
   else
     send(P, k, ?) to all processes

   wait for (P, k, *) msgs from n − f processes //"*" in {0,1,?}
   if received at least f + 1 (P, k, *) with the same v ≠? then
     decide v
   if received at least one (P, k, v) with v ≠? then
     x ← v
   else
     x ← 0 or 1 randomly // toss coin
```

At the end of the second phase a process
- decides v if received f+1 proposals v (≠ ?)
- adopts v for x if received at least one v (≠ ?)
- chooses a random value for x otherwise

Safety properties hold

Let p_i and p_j be any two processes.

Lemma 1

It is impossible for p_i to propose 0 and for p_j to propose 1 in the same round $k \geq 1$

Proof : By contradiction.

- Suppose that p_i proposes 0 and p_j proposes 1 in round k.
- Thus p_i receives $> n/2$ reports $= 0$ and p_j receives $> n/2$ reports $= 1$ in round k
- Thus it exists a process p_k that reports 0 to p_i and 1 to p_j in round k
- This is impossible

Safety properties hold

Lemma 2

If some process p_i decides v in round $k \geq 1$, then all the processes p_j that start round $k+1$ do so with $x_{p_j} = v$.

Proof :

- Suppose that some p_i decides v in round k
- p_i must have received $f+1$ proposals for v in round k
- Let p_j be any process that starts round $k+1$.
- p_j received $n-f$ proposals in Line 11 of round k
  - $\rightarrow$ p_j receives at least one v in round k (since $n-f > f+1$)
- By the first result, p_j did not receive $\overline{v}$ in round k
  - $\rightarrow$ p_j sets $x = v$ in Line 15 in round k
  - $\rightarrow$ p_j starts round $k+1$ with $x_{p_j} = v$

We say that v is $(k+1)$-locked

Safety properties hold

Lemma3

If a value v is k-locked, then every process that reaches Line 12 in round k decides v

Proof :

- Suppose v is k-locked
- Then all reports received in Line 7 of round k are equal to v
- Since $n - f > n/2$, every process that proposes a value in round k proposes v in Line 9
- Since $n - f > f + 1$, every process that reaches Line 12 decides v

Corollary 1

If some process decides v in round k, then every processes that executes Line 11 in round $k+1$ decides v in round $k+1$

Proof :
From Lemma 2 and 3

Safety properties hold

Corollary 2 - Agreement

If some processes p_i and p_j decide v and v' in round k and k' then $v = v'$

Proof : Suppose that p_i and p_j decide v and v' in round k and k'. There are 2 cases :

1. $k = k'$. Then both v and v' were proposed in round k. By Lemma 1, $v = v'$
2. $k < k'$. Since p' decides in round k', p' executed Line 11 in round $k+1, \ldots k'$. Since p decides v in round k, by repeated applications of Corollary 1, p' decides v in rounds $k+1, \ldots, k'$. So p' decides both v' and v in round k', by case 1, $v = v'$

Validity

If any process p decides v, then v is the initial value of some process

Proof : by contradiction.

- Suppose that some process p decides a value v that has never been proposed.
- Then all the processes have the initial value $\overline{v}$
- So $\overline{v}$ is 1-locked (i.e., locked in round 1)
- From Lemma 3, p decides $\overline{v}$ in round 1
- So p decides both v and $\overline{v}$. This is a contradiction by Corollary 2.

Liveness property

- By Lemma 3, if some value v is k-locked, then v is decided in round k
- At round 1, the probability that some value v is 1-locked $= (1/2)^n$
- At round k, the probability that some value v is k-locked is at least $(1/2)^n$
  - indeed some process p_i can set $x_i = v$ not necessarily by flipping a coin

Liveness property

- Hence, for any round k

$$Pr[\text{ no value is k-locked}] < 1 - (1/2)^n$$

- Since coin flips are independent,

$$Pr[\text{ no value is k-locked for the k first rounds }] < (1-(1/2)^n)^r$$

- Thus the proba that v is k-locked during the first k rounds is

$$Pr[\text{ v is k-locked during the k first rounds }] \geq 1-(1-(1/2)^n)^r$$

Any questions ?