MADS

Emmanuelle Anceaume

Lesson 2: Bitcoin and its Distributed Ledger Technology (cont'd)

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

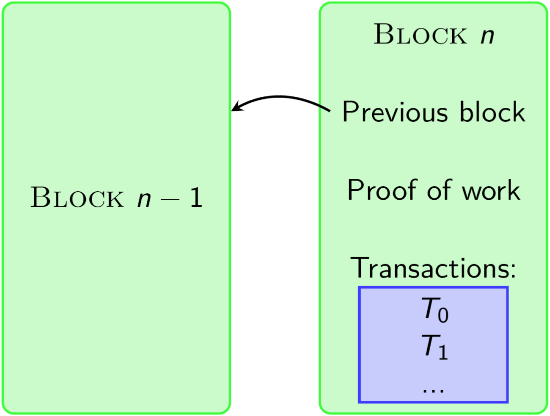
Blockchain : a sequence of blocks

- The blockchain is the data structure that implements the distributed ledger
- Each block contains transactions
  - The size of the block is limited to 1MB
  - In average (and today) there are $1,700$ transactions per block (no more than $4,000$)
- The number of blocks (today) is almost $500,000$ blocks
- The first block (block 0) of the blockchain is the genesis block

Blockchain : a sequence of blocks

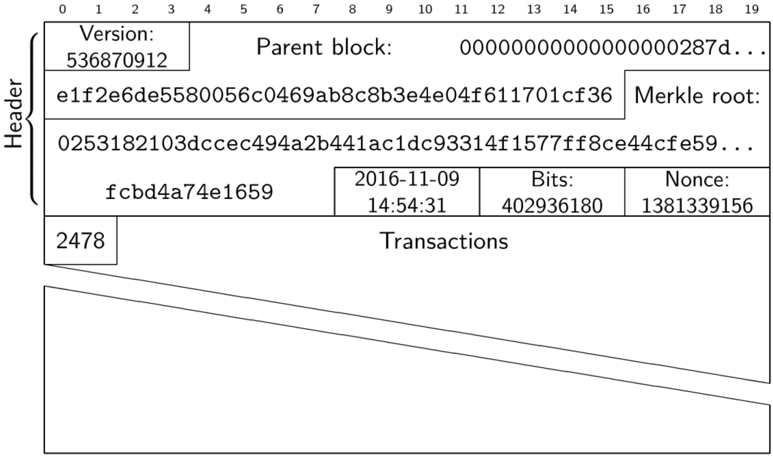
- A block = the header and the body
  - the header = allows the unique identification of a block
  - the body = contains all the transactions of the block

Blockchain : a sequence of blocks

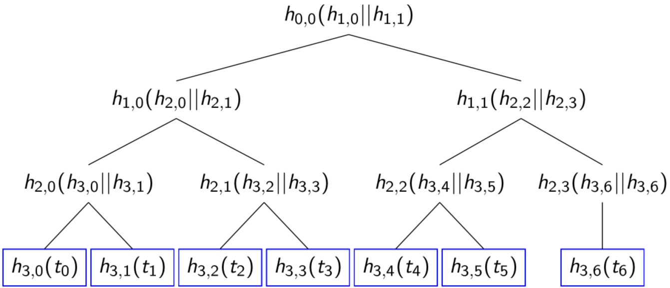


Blockchain : local view implementation

- Any locally valid transaction is embedded within a block
- Integrity proof : Merkle tree of the transactions in the block
- Resilience to sybil attacks : Hashcash Proof-of-Work
- Chaining with proof of integrity (fingerprint of the previous block)


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Header
Version: 536870912
Parent block: 0000000000000000287d...
e1f2e6de5580056c0469ab8c8b3e4e04f611701cf36
Merkle root:
0253182103dccec494a2b441ac1dc93314f1577ff8ce44cfe59...
fcbd4a74e1659
2016-11-09 14:54:31
Bits: 402936180
Nonce: 1381339156
2478
Transactions

Merkle root


$h_{0,0}(h_{1,0}||h_{1,1})$
$h_{1,0}(h_{2,0}||h_{2,1})$
$h_{1,1}(h_{2,2}||h_{2,3})$
$h_{2,0}(h_{3,0}||h_{3,1})$
$h_{2,1}(h_{3,2}||h_{3,3})$
$h_{2,2}(h_{3,4}||h_{3,5})$
$h_{2,3}(h_{3,6}||h_{3,6})$
$h_{3,0}(t_0)$
$h_{3,1}(t_1)$
$h_{3,2}(t_2)$
$h_{3,3}(t_3)$
$h_{3,4}(t_4)$
$h_{3,5}(t_5)$
$h_{3,6}(t_6)$

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td></tr>
<tr><td colspan="4">Version: 536870912</td><td colspan="16">Parent block: 00000000000000000287d...</td></tr>
<tr><td colspan="16">e1f2e6de5580056c0469ab8c8b3e4e04f611701cf36</td><td colspan="4">Merkle root:</td></tr>
<tr><td colspan="20">0253182103dccec494a2b441ac1dc93314f1577ff8ce44cfe59...</td></tr>
<tr><td colspan="8">fcbd4a74e1659</td><td colspan="4">2016-11-09 14:54:31</td><td colspan="4">Bits: 402936180</td><td colspan="4">Nonce</td></tr>
</table>

Goal: $\mathrm{SHA256} \circ \mathrm{SHA256}(\mathrm{header}) \leqslant \mathrm{target}$

Every 2016 blocks:

$$\mathrm{target} \leftarrow \mathrm{target} * \mathsf{max}\left(\frac{1}{4}, \mathsf{min}\left(4, \frac{\mathrm{real}}{\mathrm{expected}}\right)\right)$$

Hashcash Proof-of-Work

string=HelloWorld!, nonce=0, difficulty=000

Hashcash Proof-of-Work

string=HelloWorld!, nonce=0, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
```

Hashcash Proof-of-Work

string=HelloWorld !, nonce=1, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
```

Hashcash Proof-of-Work

string=HelloWorld!, nonce=2, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
HelloWorld!2 : 5b6fd9c27fcb54ca23404d9428f081b7c9280ba6370e33a6a20b16f40ce76320
```

Hashcash Proof-of-Work

string=HelloWorld!, nonce=3, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
HelloWorld!2 : 5b6fd9c27fcb54ca23404d9428f081b7c9280ba6370e33a6a20b16f40ce76320
HelloWorld!3 : 9c5d769416aa0ca894abf22bd17bd30fbb6959291423ae1903a9f86a1fe7ce78
....
```

Hashcash Proof-of-Work

string=HelloWorld!, nonce=94, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
HelloWorld!2 : 5b6fd9c27fcb54ca23404d9428f081b7c9280ba6370e33a6a20b16f40ce76320
HelloWorld!3 : 9c5d769416aa0ca894abf22bd17bd30fbb6959291423ae1903a9f86a1fe7ce78
....
HelloWorld!94 : 7090a0e5d88cff635e42ea33fcd6091a058e9cdd58ab8cd5c21c1c70421e35c6
```

Hashcash Proof-of-Work

string=HelloWorld!, nonce=95, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
HelloWorld!2 : 5b6fd9c27fcb54ca23404d9428f081b7c9280ba6370e33a6a20b16f40ce76320
HelloWorld!3 : 9c5d769416aa0ca894abf22bd17bd30fbb6959291423ae1903a9f86a1fe7ce78
....
HelloWorld!94 : 7090a0e5d88cff635e42ea33fcd6091a058e9cdd58ab8cd5c21c1c70421e35c6
HelloWorld!95 : b74f3b2cf1061895f880a99d1d0249a8cedf223d3ed061150548aa6212c88d43
```

Hashcash Proof-of-Work

string=HelloWorld!, nonce=96, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
HelloWorld!2 : 5b6fd9c27fcb54ca23404d9428f081b7c9280ba6370e33a6a20b16f40ce76320
HelloWorld!3 : 9c5d769416aa0ca894abf22bd17bd30fbb6959291423ae1903a9f86a1fe7ce78
....
HelloWorld!94 : 7090a0e5d88cff635e42ea33fcd6091a058e9cdd58ab8cd5c21c1c70421e35c6
HelloWorld!95 : b74f3b2cf1061895f880a99d1d0249a8cedf223d3ed061150548aa6212c88d43
HelloWorld!96 : 447ca2fa886965af084808d22116edde4383cbaa16fd1fbcf3db61421b9990b9
```

Hashcash Proof-of-Work

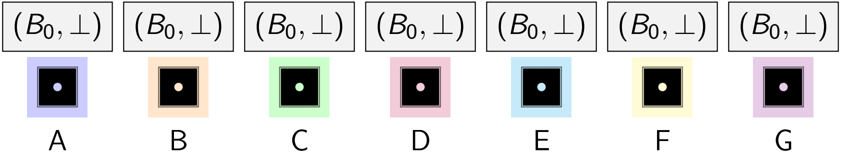
string=HelloWorld!, nonce=97, difficulty=000,

```
HelloWorld!0 : 3f6fc92516327a1cc4d3dca5ab2b27aeedf2d459a77fa06fd3c6b19fb609106a
HelloWorld!1 : b5690c48c2d0a09481186aaa99e4e090901ff2ac4d572e6706dfd30eefc22a27
HelloWorld!2 : 5b6fd9c27fcb54ca23404d9428f081b7c9280ba6370e33a6a20b16f40ce76320
HelloWorld!3 : 9c5d769416aa0ca894abf22bd17bd30fbb6959291423ae1903a9f86a1fe7ce78
....
HelloWorld!94 : 7090a0e5d88cff635e42ea33fcd6091a058e9cdd58ab8cd5c21c1c70421e35c6
HelloWorld!95 : b74f3b2cf1061895f880a99d1d0249a8cedf223d3ed061150548aa6212c88d43
HelloWorld!96 : 447ca2fa886965af084808d22116edde4383cbaa16fd1fbcf3db61421b9990b9
HelloWorld!97 : 000ba61ca46d1d317684925a0ef070e30193ff5fa6124aff76f513d96f49349d
```

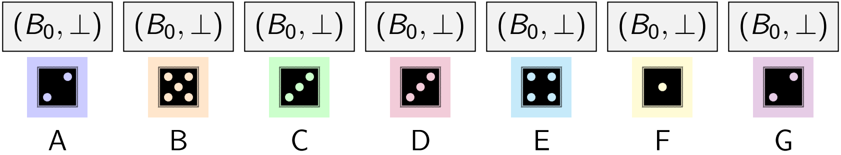
Block generation

- Bitcoin minting = creating blocks
- Computationally hard...
- ...but far from being impossible
- Result : easy to check
- Goal : find a valid PoW for the current blockchain
- Average generation time : 10 minutes

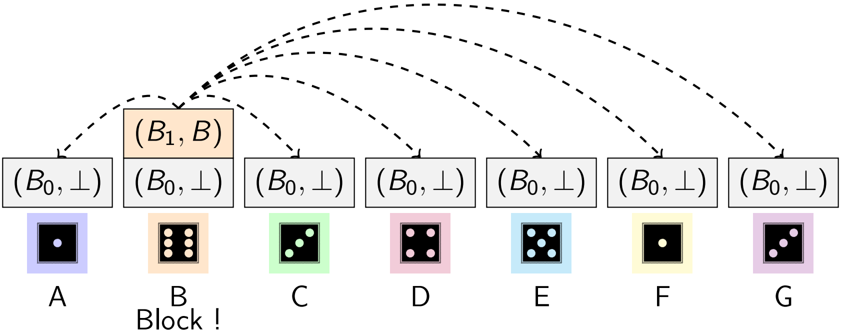
Blockchain construction



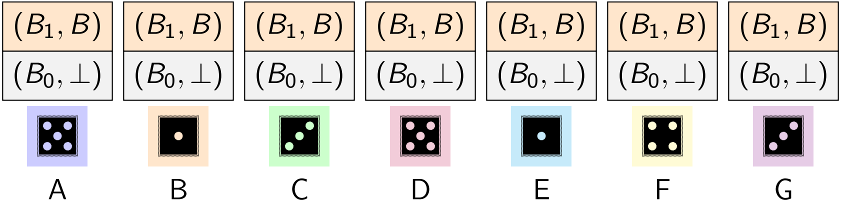
Thanks to Romaric


$(B_0, \perp)$ $(B_0, \perp)$ $(B_0, \perp)$ $(B_0, \perp)$ $(B_0, \perp)$ $(B_0, \perp)$ $(B_0, \perp)$
A
B
C
D
E
F
G

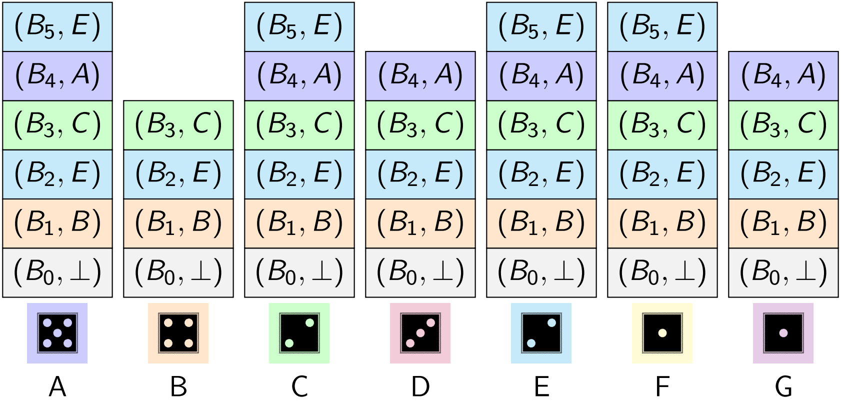
Blockchain construction



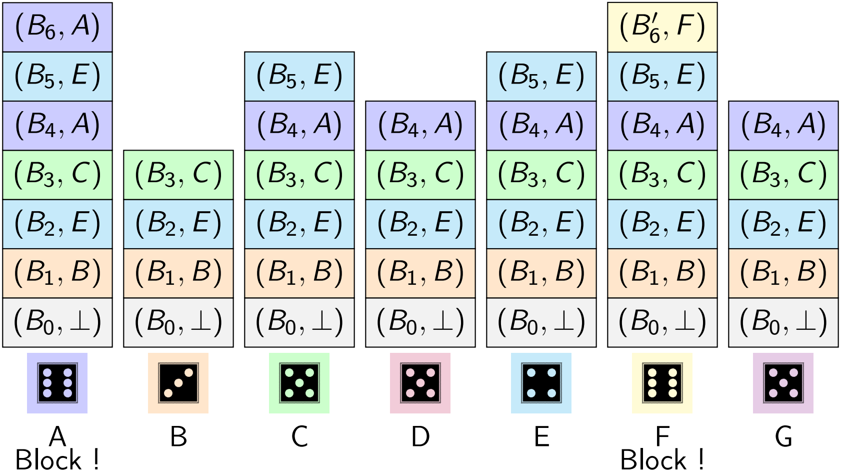
Blockchain construction



Blockchain construction



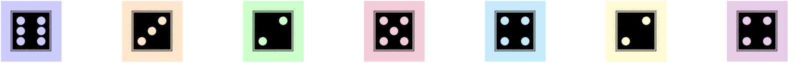
Transient inconsistencies



A	B	C	D	E	F	G
(B_7, G)		(B'_7, C)	(B'_7, C)			(B_7, G)
(B_6, A)	(B'_6, F)	(B'_6, F)	(B'_6, F)	(B_6, A)	(B'_6, F)	(B_6, A)
(B_5, E)	(B_5, E)	(B_5, E)	(B_5, E)	(B_5, E)	(B_5, E)	(B_5, E)
(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)
(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)
(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)
(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)
$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$
A	B	C Block !	D	E	F	G Block !

The longest chain is locally kept by every node

A	B	C	D	E	F	G
(B_8, A)		(B_8, A)	(B_8, A)		(B_8, A)	
(B_7, G)	(B_7, G)	(B_7, G)	(B_7, G)		(B_7, G)	(B_7, G)
(B_6, A)	(B_6, A)	(B_6, A)	(B_6, A)	(B_6, A)	(B_6, A)	(B_6, A)
(B_5, B)	(B_5, B)	(B_5, B)	(B_5, B)	(B_5, B)	(B_5, B)	(B_5, B)
(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)	(B_4, A)
(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)	(B_3, C)
(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)	(B_2, E)
(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)	(B_1, B)
$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$	$(B_0, \perp)$



A B C D E F G

Block !

Does Pow solves the consensus problem ?

It is frequently argued that because the set of miners succeed in reaching an agreement on the next block to append to the blockchain, Bitcoin solves the consensus problem in an asynchronous distributed system

In the remaining of the class we will see that in an asynchronous system there is no protocol that solves the consensus problem even if a single node may crash

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.

Asynchronous Systems
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. Distributed

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
  $d_i \leftarrow \delta((1, v_1), \ldots, (n, v_n) \cup (i, v_i))$
  return decide($d_i$)
}
```

FLP impossibility result [FLP85]

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

A step takes one configuration to another and consists of an atomic set of actions by a single process p

- 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

Step

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 (so an event can be though as the receipt of m by p)

Schedule

- A schedule from a configuration C is a finite or infinite sequence σ of events that can be applied in turn from C.
- This sequence of steps is called a run.
- If σ is finite then the resulting configuration is denoted by $C.\sigma$. We say that it is reachable from C. A configuration reachable from an initial configuration is said accessible.
- In the following we only consider accessible configurations

Decision value

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

Correct consensus protocol P

A consensus protocol is partially correct is

- It does not exist an accessible configuration which has more than one decision value
- For each value $v \in \{0, 1\}$, some accessible configuration has decision value v

A process is non faulty in a run provided it takes an infinite number of steps. It is faulty otherwise

A run is admissible provided that at most one process is faulty and all messages have been delivered

A run is a deciding run provided that some process reaches a decision in that run

A consensus protocol is correct despite a single fault if is partially correct and every admissible run is a deciding run

Main result

Theorem

No consensus protocol is correct in spite of one fault

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

Valence of configurations

The core of FLP argument is a strategy allowing the adversary (who controls the scheduling) to steer the execution away from any configuration in which the processes reach agreement.

The strategy relies on the notion bivalence.

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**

An execution σ is 0-valent if 0 is the only value that can ever be decided by any process in σ.

An execution σ is bivalent if 0 appears in a decide state and 1 appears in a decide state

Strategy of the adversary

- Any configuration where some process decides is not bivalent.
- So if the adversary can keep the protocol in a bivalent configuration forever, then it can prevent the processes from ever deciding.

Strategy :

1. Make the protocol start in a bivalent configuration C_0 (we must prove that such a configuration always exists)
2. Choose only bivalent successor configurations (we must prove that it is always possible)

Bivalent initializations

Lemma

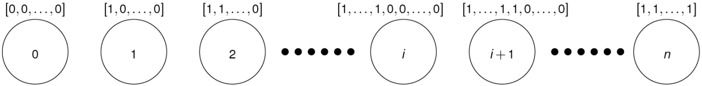
Any consensus protocol that tolerates at least one faulty process has at least one bivalent configuration.

What does it means ?

- The final decision cannot be determined from just the inputs
- If there are not failures, then it is simple to build a consensus algorithm that have only univalent configurations

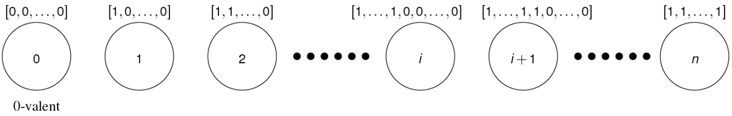
There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



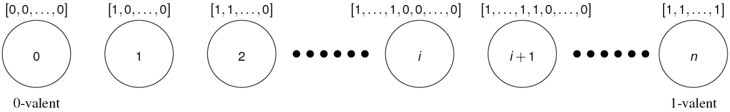
There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



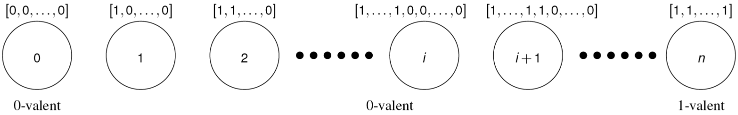
There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



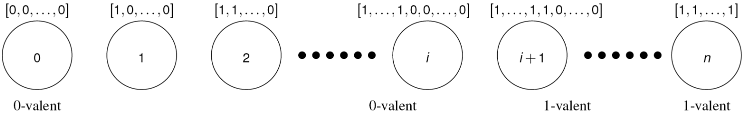
There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



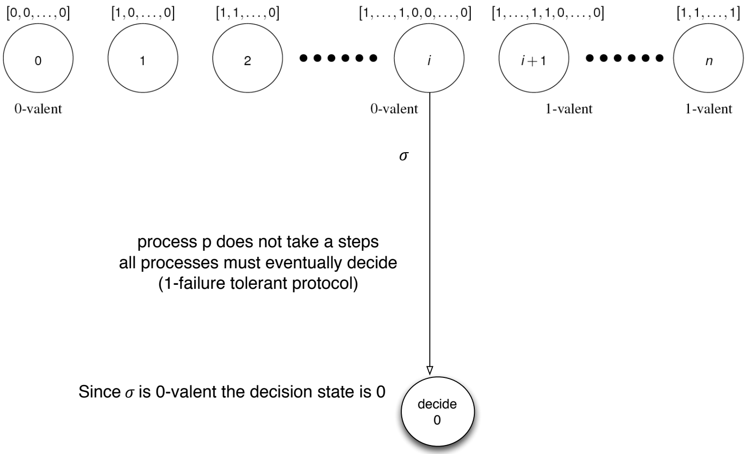
There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



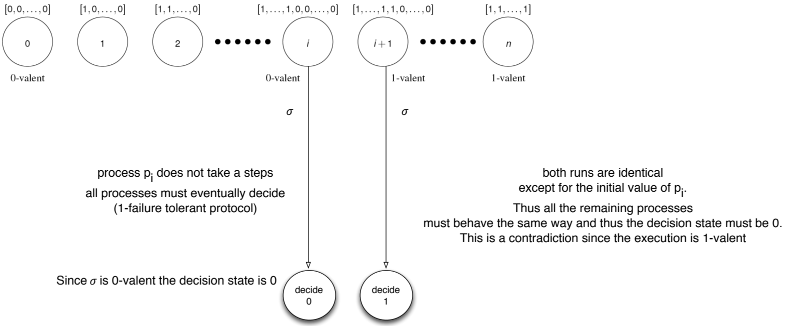
There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



There exists at least one bivalent configuration

Proof : By contradiction.
Suppose that all the initial configurations are univalent.



Bivalent extension

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

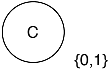
Proof of the bivalent extension lemma

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

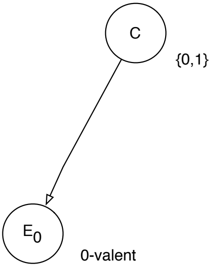
Proof of the bivalent extension lemma

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



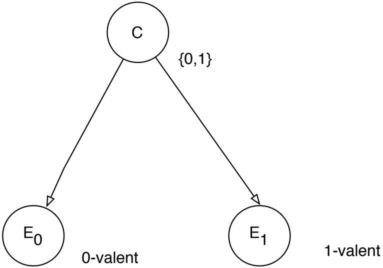
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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



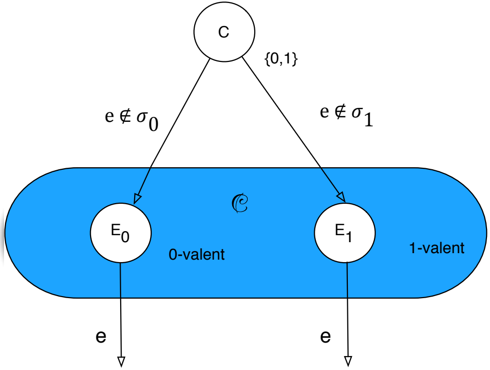
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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



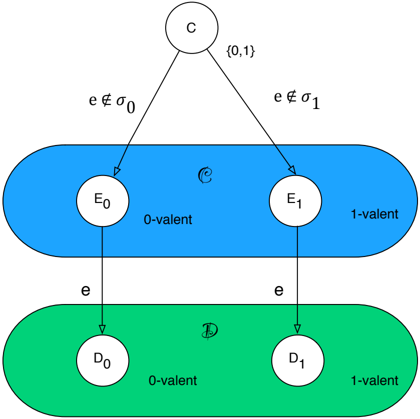
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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



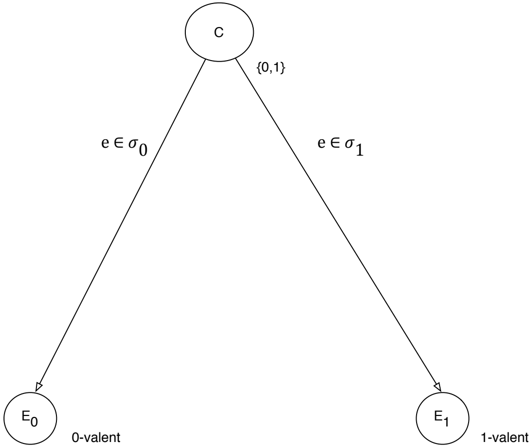
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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.



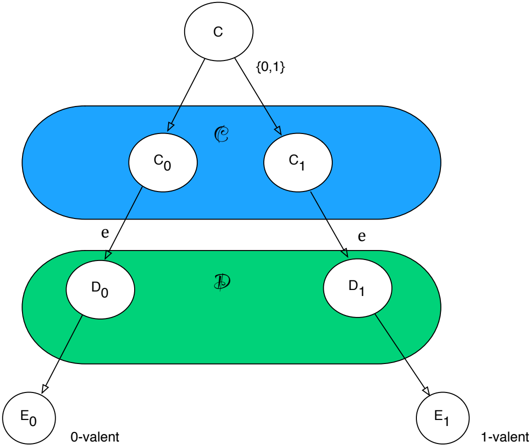
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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



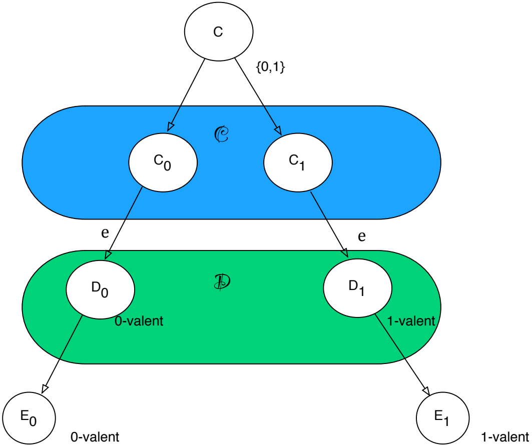
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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.



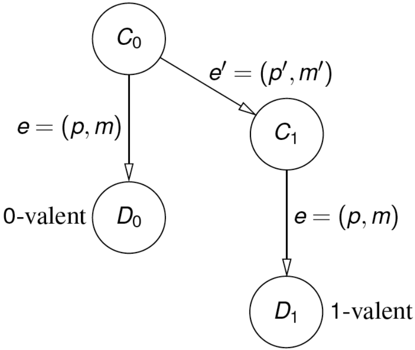
$\mathcal{D}$ contains both 0-valent and 1-valent configurations

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

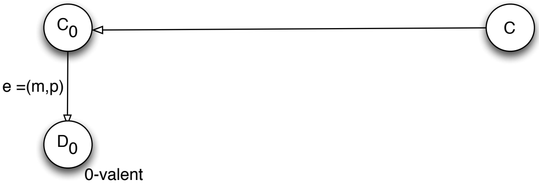
- Definition :
  - Configurations C_0 and C_1 are neighbor if one results from the other by application of a single step.

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?

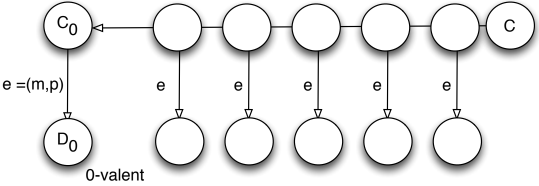


Let C be a bivalent configuration, and C_0 reachable from C that leads to D_0 a 0-valent configuration of $\mathcal{D}$ by applying step e



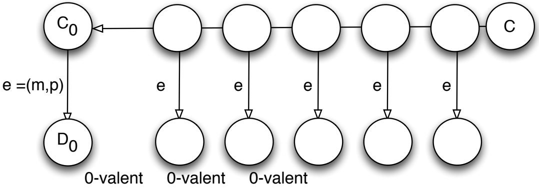
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



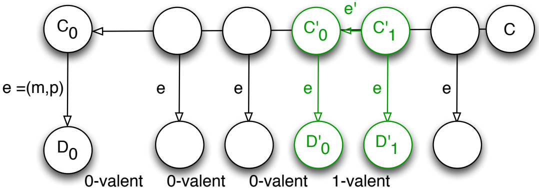
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



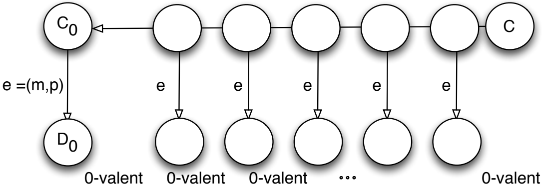
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



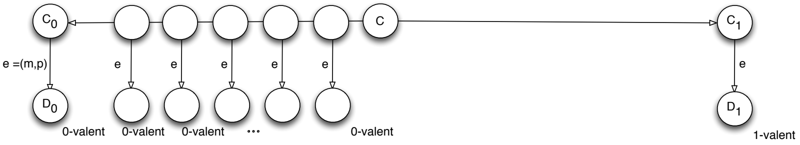
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Otherwise all of them of 0-valent.



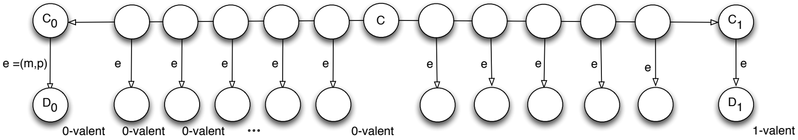
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



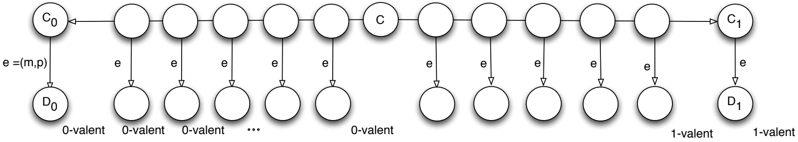
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



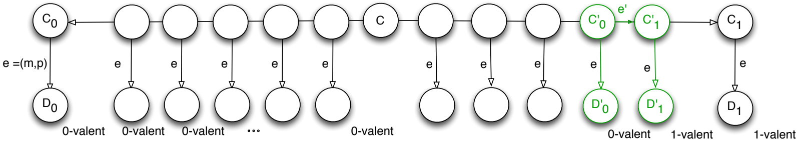
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



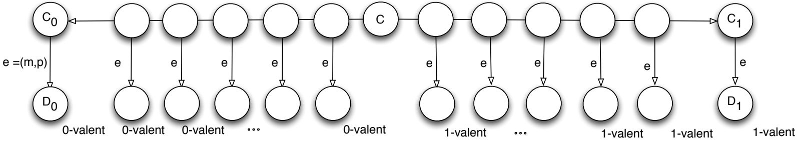
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



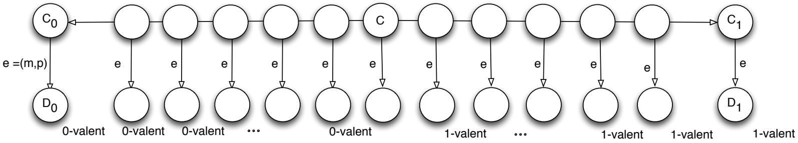
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

Otherwise all of them of 1-valent.



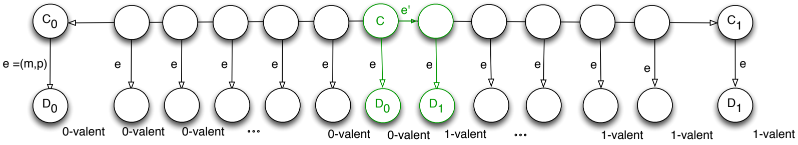
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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



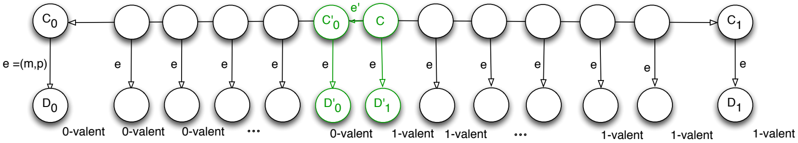
Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

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

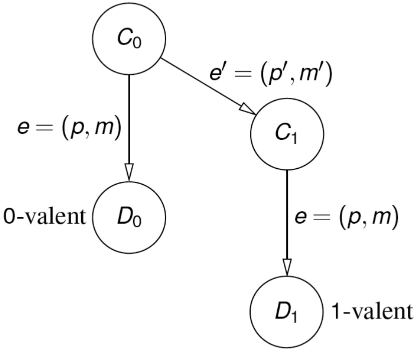


Two neighbor configurations C_0 and C_1 in $\mathcal{C}$ exist

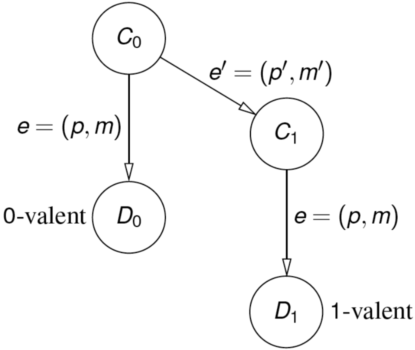
Or this configuration of $\mathcal{D}$ is 1-valent, and thus we can identify the hook we were looking for



Where have we been so far ?



Where have we been so far ?



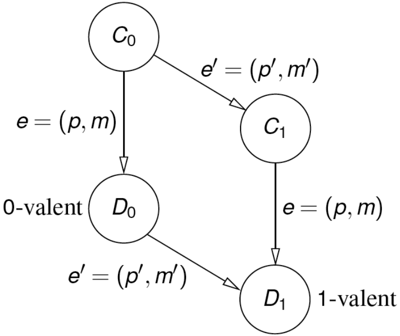
We are almost done. We need to consider two cases :

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

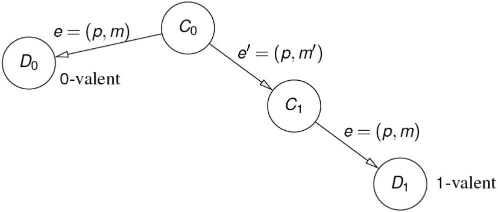
$p \neq p'$

- 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.



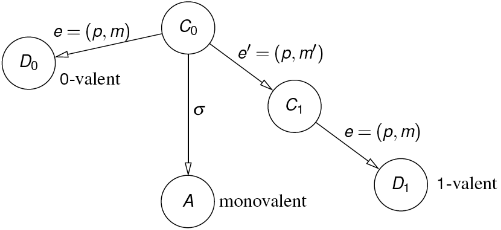
$p = p'$



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

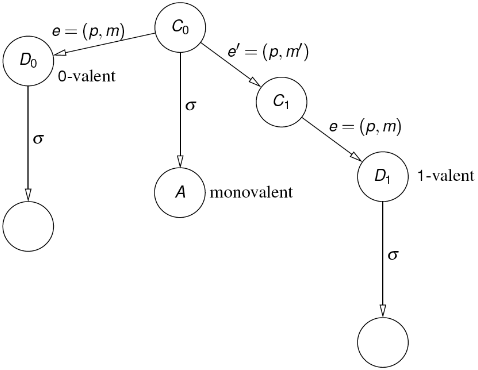
1. All the processes decide
2. Except p that does not make any step in σ (the protocol tolerates one crash thus it must allow $n-1$ processes to decide)

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



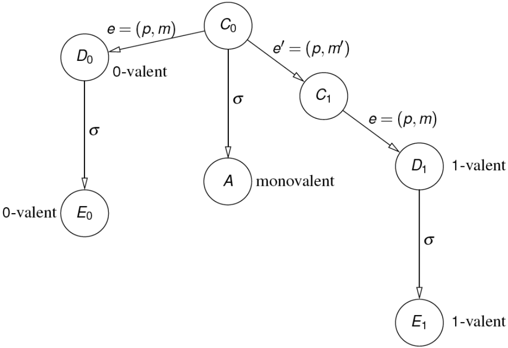
$p = p'$

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



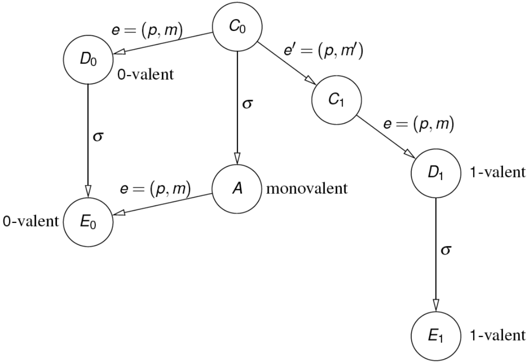
$p = p'$

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

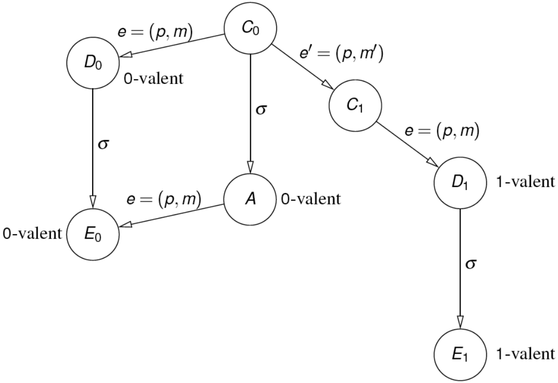


$p = p'$

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.

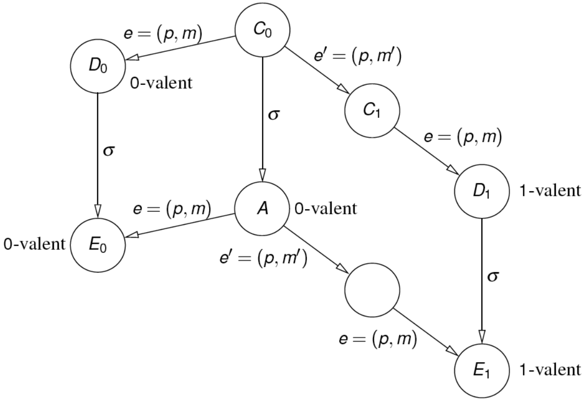


Thus configuration A must be 0-valent

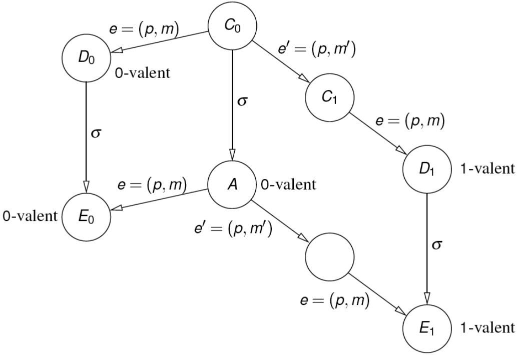


$p = p'$

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



What we have shown

- There exists at least one initial configuration which is bivalent. We start our infinite execution from this configuration C
- 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.

Any questions ?