The consensus problem in asynchronous environments

http://people.irisa.fr/Emmanuelle.Anceaume/
In this problem processors are trying to reach a consensus on a value.

Each processor initially proposes a value $v$ taken from a given set of value $V$.

At the end of the protocol, all processors agree on a single value, called the decided value, or decision.
Each processor $p_i$ has special state components

- $x_i$: the input
- $y_i$: the output (also called decision)

Initially, $x_i$ holds a value, and $y_i$ is undefined. Any assignment to $y_i$ is irreversible.

A solution to the Consensus problem must guarantee the following:

- **Termination**: $y_i$ is eventually assigned a value for every non-faulty processor $p_i$.
- **Agreement**: If $y_i$ and $y_j$ are assigned, then $y_i = y_j$ for all non-faulty processors $p_i$ and $p_j$.
- **Validity**: If for some value $v$, $x_i = v$ for all processors $p_i$, and if $y_i$ is assigned for some non-faulty processor $p_i$ then $y_i = v$. 
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 »
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, 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

- processors 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
  - \(\text{send}(p, m)\) : places \((p, m)\) in the message buffer
  - \(\text{receive}(p)\) :
    - delete some message \((p, m)\) from the buffer and returns \(m\) to \(p\). We also 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

\[ p \text{send}(p', m) \quad p' \text{receive}(p') \]

May return \(m\) or null

Global message buffer
A **configuration** $C$ (or global state) of the system consists of the internal state $s_i$ of each process $p_i$ plus the content of the message buffer

- $C = (s, B)$ 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.
The system moves from one configuration to the next one by an event. Let $C = (s, B)$ be a configuration.

A event executed by process $p$ consists of the following set of actions executed atomically:

- $p$ performs $\text{receive}(p)$ on the message buffer in $B$ of $C$: $p$ delivers a value $m \in \{M, \text{null}\}$
- based on its local state in $C$ and $m$, $p$ enters a new state and sends a finite number of messages (i.e., deposits them in the message buffer)
- $C.e$ 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.
A schedule is a finite or infinite linear sequence of events taken by the processes from a given configuration of the system.
Lemma (Diamond)

Suppose that from some configuration $C$, the schedules $\sigma_e$ and $\sigma_{e'}$ lead to configurations $C_e$, $C_{e'}$, respectively. If the sets of processes taking steps in $C_e$ and $C_{e'}$, respectively, are disjoint, then $\sigma_{e'}$ can be applied to $C_e$ and $\sigma_e$ can be applied to $C_{e'}$, and both lead to the same configuration $C'$. 
Commutativity property of schedules

Proof.

- Suppose that $\sigma_e$ (resp. $\sigma_{e'}$) is reduced to a single event $e$ (resp. $e'$)
- Observe that $e'$ can be applied to $C_e$ since $m'$ is still in transit and $e$ does not change the state of $p'$.
- The same argument applies for $e$.
- Thus both $C_{ee'}$ and $C_{e'e}$ are well-defined.
Commutativity property of schedules

Proof (cont’d).
- The same set of messages $m$ and $m'$ is consumed after the application of both events.
- The same set of messages is sent after the application of both events.
- Thus both $C_{ee'}$ and $C_{e'e}$ are equal.
- Note that this argument applies for two sets of processes that do not interact with each other.

![Diagram showing the commutativity property of schedules with events and processes labeled C1, C2, C3, and events e=(p,m), e'=(p',m'), e'=(p',m'), e=(p,m).]
A configuration \( C \) has decision value \( \nu \) if some process \( p \) is in a decision state (i.e. \( y_p = 0 \) or \( y_p = 1 \)).

A consensus algorithm is partially correct if

1. No configuration has more than one decision value.
2. For each \( \nu \in \{0, 1\} \), some configuration has decision value \( \nu \).
A process $p$ is **nonfaulty** in a run provided that it takes infinitely many steps, and it is **faulty** otherwise.

An execution is **admissible** provided that at most one process is faulty and that all messages sent to nonfaulty processes are eventually received.

An execution is a **deciding execution** provided that some process reaches a decision state in that execution.

A consensus protocol $P$ is totally correct in spite of one fault if it is partially correct, and every admissible run is a deciding run.

We will show that every partially correct protocol for the consensus problem has some admissible execution that is not a deciding execution.
Proving the correctness of a distributed algorithm is a game

When designing fault-tolerant algorithms, we often assume the presence of an adversary that plays against the algorithm

- 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 is the scheduler:
  - it will select the next process to take a step
  - It will select the message the process will receive

However

- It cannot prevent a message from being eventually received
- It cannot make more than one process crashed
The idea behind the theorem is to show that there exists some admissible execution 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.
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 which depends on
  - the initial configuration
  - the schedule which depends on the asynchrony of the system messages, i.e., receipt out of order, arbitrary delays
  - and the potential failures
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.

An execution $\sigma$ is 0-valent (or 1-valent) if 0 (1) is the only value that can ever be decided by any process in $\sigma$.

An execution $\sigma$ is bivalent if 0 appears in a decide state and 1 appears in a decide state.
0-decided configuration

- A configuration with decide “0” on some process

```
0-decided configuration
{ state2,
  state5
}

At least one of them is in state DECIDE-0
```

```
state7
{msg1, msg2}
```

```
P1 state2
P2 state5
P4 state7
P3 decide0
```

Diagram:
- P1 in state2
- P2 in state5
- P3 in decide0
- P4 in state7

Edges:
- P1 to P2
- P1 to P3
- P1 to P4
- P2 to P3
- P2 to P4
- P3 to P4

Messages:
- msg1
- msg2
0-valent configuration

- No 1-decided configurations are reachable
- Future determined, means ”everyone will decide 0”
1-valent configuration

- No 0-decided configurations are reachable
- Future determined, means "everyone will decide 1"
Bivalent configuration

- Both 0 and 1-decided configurations are reachable
- Future undetermined, could go either way…
Lemma (2)

Any consensus protocol that tolerates at least one faulty process has at least one bivalent initial configuration.
Proof of the existence of bivalent initial configurations

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

→ the difference between any two adjacent configurations is the starting value of one process
Proof of the existence of bivalent initial configurations

0-valent

[0, 0, ..., 0]  [1, 0, ..., 0]  [1, 1, ..., 0]  [1, ..., 1, 0, 0, ..., 0]  [1, ..., 1, 1, 0, ..., 0]  [1, 1, ..., 1]
Proof of the existence of bivalent initial configurations

[0,0,...,0]  [1,0,...,0]  [1,1,...,0]  [1,...,1,0,0,...,0]  [1,...,1,1,0,...,0]  [1,1,...,1]

0-valent  1-valent

0  1  2  \ldots  i  i+1  \ldots  n
Proof of the existence of bivalent initial configurations

- [0,0,...,0]
  - 0
  - 0-valent

- [1,0,...,0]
  - 1

- [1,1,...,0]
  - 2

- [1,...,1,0,0,...,0]

- [1,...,1,1,0,...,0]
  - i
  - 0-valent

- [1,...,1,1,...,1]
  - i+1
  - 0-valent

- [1,1,...,1]
  - n
  - 1-valent
Proof of the existence of bivalent initial configurations

\[ [0,0,\ldots,0] \quad [1,0,\ldots,0] \quad [1,1,\ldots,0] \quad [1,\ldots,1,0,0,\ldots,0] \quad [1,\ldots,1,1,0,\ldots,0] \quad [1,1,\ldots,1] \]

0-valent \quad 1 \quad 2 \quad \cdots \quad i \quad i+1 \quad \cdots \quad n

0-valent \quad 0-valent \quad 1-valent \quad 1-valent
Proof of the existence of bivalent initial configurations

- Let $\sigma$ be a deciding execution applicable to configuration $C_0$
- $\sigma$ does not contain any step from process $p$
Intuition

Given any algorithm, we can find some start state, that depending on the failure of one process will either lead to a 0-decide or a 1-decide
Proof of the existence of bivalent initial configurations

In $\sigma$ processor $p_i$ takes no step
So its initial value cannot be observed by someone else

All processors must eventually decide
(f-failure tolerant protocol)

Since $C_0$ is 0-valent the decision state is 0
Proof of the existence of bivalent initial configurations

In \( \sigma \) processor \( p_i \) fails initially (i.e., does not receive nor send messages).

So its initial value cannot be observed.

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

Since \( C_0 \) is 0-valent the decision state is 0

Run \( \sigma \) can be made from \( C_1 \) too since no processor has ever heard about \( p_i \).
Thus all the processors (except \( p_i \)) should reach the "0" deciding state.

This is a contradiction since by assumption \( C_1 \) is a "1"-valent configuration.
Proof of the existence of bivalent initial configurations

- $C_0$ can lead to a "0" decision state or to a "1"-decision state
- Thus $C_0$ is a bivalent configuration

- Clearly, the outcome of the consensus algorithm is not uniquely predetermined by the initial configurations

Initial bivalent configuration

Any asynchronous 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
Any bivalent configuration is exactly one of these three:

- Bivalent
- 0-valent
- 1-valent

Why?

- We know that bivalent configurations exist
- Any configuration leads to a decide
  - i.e., must lead to either a 0-decide or a 1-decide
- So any configuration is either 0-valent or 1-valent
Bivalent configuration

Case 1

bivalent \rightarrow bivalent

Case 2

bivalent \rightarrow 0-valent
\rightarrow 1-valent
A little bit more formally . . .

**Lemma (Bivalent extension)**

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

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

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

Then $D$ contains a bivalent configuration.

- Note that step $e$ is always applicable in $C$ since
  - $e$ is applicable to $C$
  - $C$ is the set of configurations reachable from $C$
  - and messages can be delayed arbitrarily long
Assume $e$ involves process $p$

Let $C$ be all possible configs reachable from $C$ without applying $e$
  - $C$ is in $C$ as well

Apply event $e$ to all configs in $C$ and call the resulting configs $\mathcal{D}$
The proof is by contradiction

1. We assume that $\mathcal{D}$ contains no bivalent configurations.
2. So every configuration $D \in \mathcal{D}$ is univalent, i.e., either 0-valent or 1-valent.
3. We proceed to derive a contradiction.
Proof of the bivalent extension lemma

We start from a bivalent configuration $C$ ($C$ exists by the Lemma 1)
There must exist a 0-valent configuration $E_0$ reachable from $C$
$D$ contains both 0-valent and 1-valent configurations

There must exist a 1-valent configuration $E_1$ reachable from $C$
Case 1: If $E_i$, $i = 0, 1$, belongs to $\mathcal{C}$ (that is step $e$ is not applied along $\sigma_i$) 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 \( \sigma_i \)).
\( \mathcal{D} \) 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 \( D_i \in \mathcal{D} \).
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.
So far we have shown that $D$ contains both 0-valent and 1-valent configurations.

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

We want to prove that $C$ contains two neighbor configurations $C_0$ and $C_1$ that lead to $D_0$ and $D_1$ in $D$. 
What do we want to prove?

\[
C_0 \xrightarrow{e = (p, m)} D_0 \xleftarrow{e' = (p', m')} C_1 \xrightarrow{e = (p, m)} D_1 \quad \text{1-valent}
\]
Two neighbor configurations $C_0$ and $C_1$ in $C$ exist

Let $C$ be a bivalent configuration, and $C_0$ reachable from $C$ that leads to $D_0$ a 0-valent configuration of $D$ by applying step $e$
Two neighbor configurations $C_0$ and $C_1$ in $C$ 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 $C$ exist

All these configurations belong to $D$. 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.

\[
\begin{align*}
C_0 & \quad C_0' \quad C_1' \quad C_1 \\
D_0 & \quad D_0' \quad D_1' \quad D_1 \\
\end{align*}
\]

$e = (m, p)$

0-valent 0-valent 0-valent 1-valent
Two neighbor configurations $C_0$ and $C_1$ in $C$ exist. Otherwise all of them of 0-valent.
Two neighbor configurations \( C_0 \) and \( C_1 \) in \( C \) exist.

Then consider \( C_1 \) a configuration in \( C \) reachable from \( C \) that leads to \( D_1 \) a 1-valent configuration in \( D \) by applying step \( e \).
Two neighbor configurations $C_0$ and $C_1$ in $C$ 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 $C$ exist. All these configurations belong to $D$. Hence they are all univalent. Some of them can be 1-valent as is $D_1$. 

$e = (m, p)$
Two neighbor configurations $C_0$ and $C_1$ in $C$ 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 $C$ 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 $C$ exist.

Or this configuration of $D$ is 1-valent, and thus we can identify the hook we were looking for.
Where have we been so far?

\[ C_0 \xrightarrow{e = (p, m)} D_0 \xrightarrow{e = (p, m)} C_1 \xrightarrow{e' = (p', m')} D_1 \]

0-valent

1-valent
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 $\sigma$ be a deciding execution that can be applied to $C_0$ such that

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

- Let $A = \sigma(C_0)$ be such a decision configuration
- By the validity property of the consensus protocol, configuration $A$ must be univalent
Since $p$ takes no step in $\sigma$, $\sigma$ can be applied to $D_0$ and to $D_1$ (Diamond lemma)
Leading to a 0-valent configuration $E_0$ and 1-valent configuration $E_1$
Now the adversary allows $p$ to make its step $e$ from configuration $A$. This leads to configuration $E_0 = e(A)$ by applying the same argument as before.
Thus configuration $A$ must be 0-valent
Both $e'$ and $e$ can be applied to configuration $A$ and leads to $E_1 = e(e'(A))$. 
Thus $A$ must be 1-valent. But $A$ is 0-valent. A contradiction.
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.
Consensus problem

- Agreement in distributed systems
- Solutions exist in synchronous systems
- Impossible to solve in an asynchronous system (e.g. Internet)
  - key idea: with even one adversarial crash-stop process failure, there are always sequences of events for the system that prevent process from deciding
  - It holds true regardless of the algorithm you choose!
- 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?