# The FLP Theorem

Jacopo Notarstefano
jacopo.notarstefano [at] gmail.com

November 26, 2014

# The Distributed Consensus Problem

Let's consider the problem of reaching agreement on a single value in a distributed, asynchronous system of processes.

For example, in a distributed database, all entities that have partecipated in the processing of a particular transaction need to agree wether to commit or rollback.

Whatever decision is made, all entities must make the same decision in order to preserve the consistency of the database.

# Faults

In this course we usually assumed that the partecipating processes and the network were completely reliable.

This is not the case with real systems, which are subject to a number of possible **faults**, such as process crashes, network partitioning, and lost, distorted, duplicated messages.

One can even consider more **Byzantine** types of failures, in which faulty processes might go completely haywire, perhaps even sending messages according to some malevolent plan.

# The FLP Theorem

In 1985, Michael Fischer, Nancy Lynch, and Michael Paterson proved the surprising result that no completely asynchronous consensus protocol can tolerate even a single unannounced process death.

Their proof did not consider Byzantine failures, and assumed that the message system is reliable — it delivers all messages correctly and exactly once.

Nevertheless, even with these assumptions, the stopping of a single process at an inopportune time can cause any distributed consensus protocol to fail to reach agreement.

# Consensus protocol

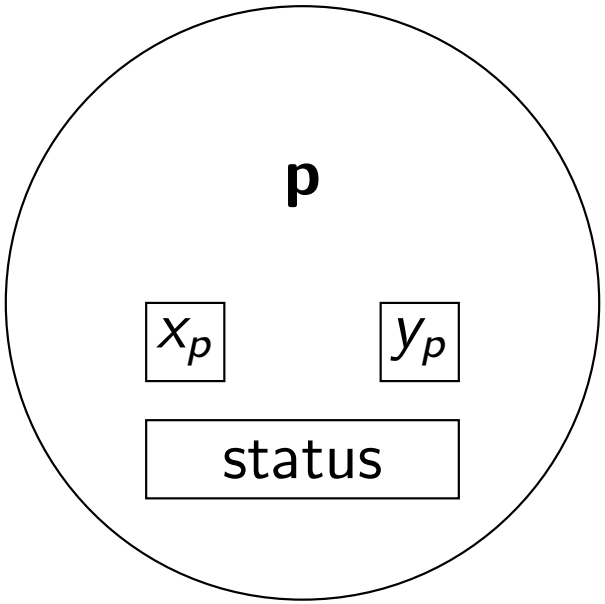
A **consensus protocol** is an asynchronous system of $N$ processes ($N \geq 2$). Each process $p$ has a one-bit **input register** $x_p$, an **output register** $y_p$ with values in $\{\perp, 0, 1\}$ and an unbounded amount of internal storage.

**Initial states** prescribe fixed starting values for all but the input register; in particular, the output register starts with value $\perp$.

$p$ acts deterministically according to a **transition** function.

# Decision states

The states in which the output register has value 0 or 1 are distinguished as being **decision states**. The transition function cannot change the value of the output register once the process has reached a decision state; that is, the output register is "write-once".



# Message system

A **message** is a pair $(p, m)$, where $p$ is the name of the destination process and $m$ is a "message value" from a fixed universe $M$.

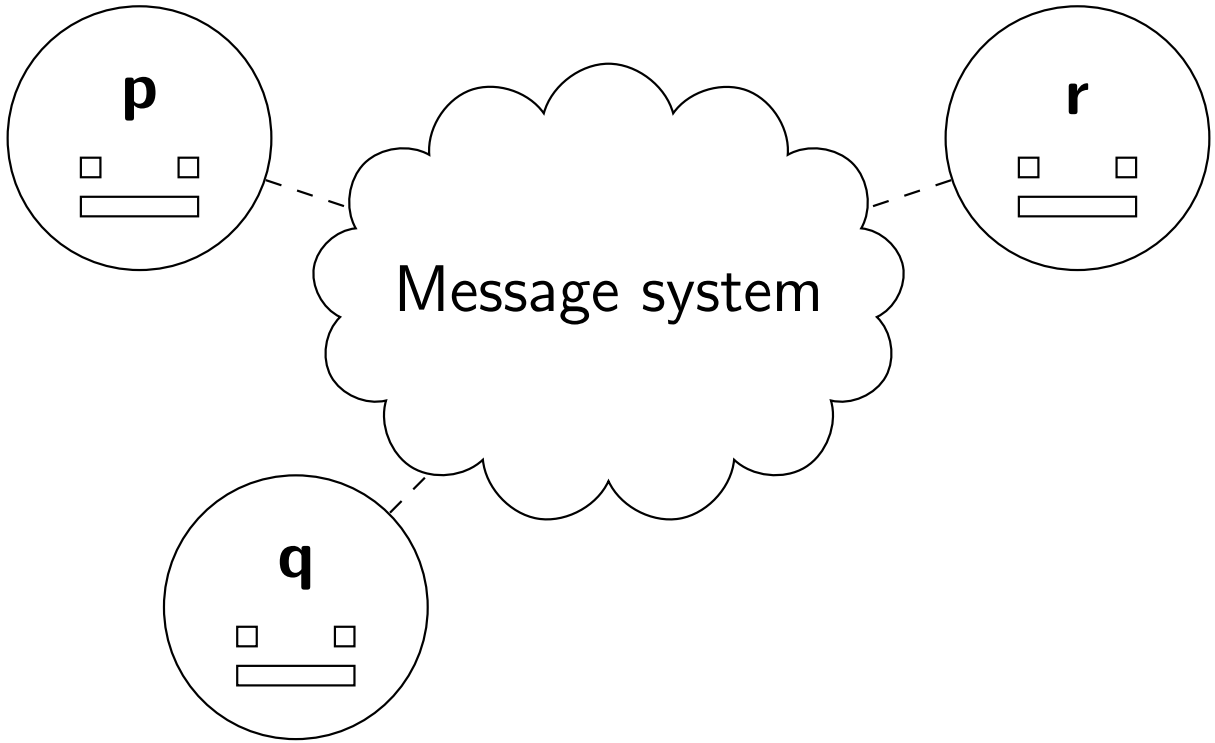
The **message system** mantains a multiset, called the **message buffer**, of messages that have been sent but not yet delivered. It supports two abstract operations:

`send(p,m)`: Places $(p, m)$ in the message buffer.

`receive(p)`: Deletes some message $(p, m)$ from the buffer and returns $m$, in which case we say $(p, m)$ is **delivered**, or returns the special null marker $\varnothing$ and leaves the buffer unchanged.

# Nondeterministic messaging

The message system acts nondeterministically, subject only to the condition that if `receive(p)` is performed infinitely many times, then every message $(p, m)$ in the message buffer is eventually delivered.



# Configurations and steps

A **configuration** of the system consists of the internal state of each process, together with the contents of the message buffer. An **initial configuration** is one in which each process starts at an initial state and the message buffer is empty.

A **step** takes one configuration to another and consists of a primitive step by a single process $p$.

# Primitive steps and applicable events

Let $C$ be a configuration. A step consists of two phases:

1. A `receive(p)` is performed on the message buffer in $C$ to obtain a value $m \in M \cup \{\varnothing\}$.
2. Depending on $p$'s internal state in $C$ and on $m$, $p$ enters a new internal state and sends a finite set of messages to other processes.

The step is completely determined by the pair $e = (p, m)$, which we call an **event**.

An event $e$ that *could happen* at configuration $C$ is called **applicable**, and the resulting configuration is denoted $e(C)$.

# Schedules and runs

A **schedule** from $C$ is a finite or infinite sequences $\sigma$ of applicable events, in turn, starting from $C$. The associated sequence of steps is called a **run**.

If $\sigma$ is finite, we let $\sigma(C)$ denote the resulting configuration, which is said to be **reachable** from $C$.

A configuration reachable from some initial configuration is said to be **accessible**.

# Partial correctness

A configuration $C$ has **decision value** $v$ if some process $p$ is in a decision state with $y_p = v$.

## Definition (Partial correctness)

A consensus protocol is **partially correct** if:

1. No accessible configuration has more than one decision value.
2. For each $v \in \{0, 1\}$, some accessible configuration has decision value $v$.

# Total correctness in spite of one fault

A process $p$ is **nonfaulty** in run if it takes infinitely many steps, otherwise it is **faulty**.

A run is **admissible** if at most one process is faulty and all messages sent to nonfaulty processes are eventually received.

A run is **deciding** if some process reaches a decision state.

**Definition (Total correctness in spite of one fault)**

A consensus protocol $P$ is **totaly correct in spite of one fault** if it is partially correct and every admissibile run is deciding.

# Main result

**Theorem (Fischer, Lynch, Paterson [FLP85])**

*No consensus protocol is totally correct in spite of one fault.*

A configuration is **bivalent** if the set of decision values of configurations reachable from it has 2 elements. It is instead 0-**valent** or 1-**valent** according to the corresponding value.

**Proof (sketch).**

Given an initial bivalent configuration, we construct an admissible run that at each stage results in another bivalent configuration. □

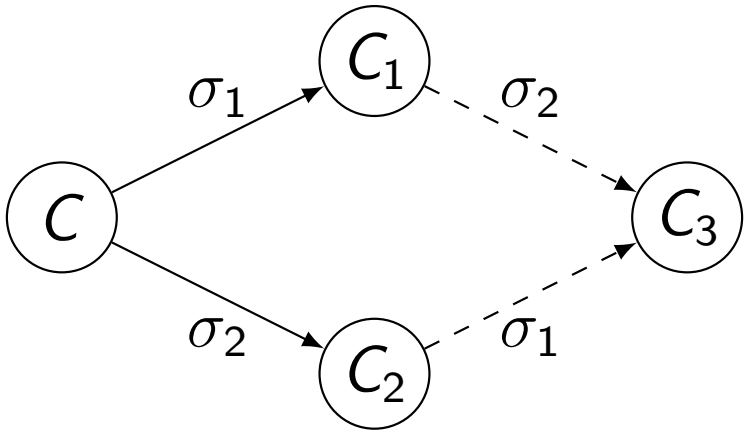
# Lemma 1

**Lemma**

*Suppose that from some configuration $C$, the schedules $\sigma_1$, $\sigma_2$ lead to configurations $C_1$, $C_2$ respectively. If the sets of processes taking steps in $\sigma_1$ and $\sigma_2$, respectively, are disjoint, then $\sigma_2$ can be applied to $C_1$ and $\sigma_1$ can be applied to $C_2$, and both lead to the same configuration $C_3$.*

In other words: **schedules about disjoint processes commute with each other**.

# Proof of Lemma 1




Because the sets of processes are disjoint, an event in $\sigma_1$ applicable to $C$ is applicable to $C_2$ as well.

Because of determinism, after all events are processed they must end up in the same state. □

# Lemma 2

**Lemma**

*$P$ has a bivalent initial configuration.*

Assume by contradiction that there isn't. Then, by partial correctness, $P$ must have both 0-valent and 1-valent initial configurations.

Let's call two initial configurations **adjacent** if they differ only in the initial value $x_p$ of a single process $p$. Any two initial configurations are joined by a chain of inital configurations, each adjacent to the next.

# Proof of Lemma 2

TODO

# Lemma 3

**Lemma**

*Let $C$ be a bivalent configuration of $P$, 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 applying $e$, and let $\mathcal{D} = e(\mathcal{C}) = \{e(E) | E \in \mathcal{C}$ and $e$ is applicable to $E\}$. Then, $\mathcal{D}$ contains a bivalent configuration.*

In other words: given a bivalent configuration and an event $e$ applicable to it, **we construct another bivalent configuration having $e$ as the last applied event**.

# Proof of Lemma 3

**Proof (Lemma 3).**

Assume by contradiction that $\mathcal{D}$ contains no bivalent configuration. We first show that $\mathcal{D}$ contains both 0-valent and 1-valent configurations. □

# Proof of main result

**Proof (main result).**

□

# Conclusions

TODO

# Bibliography

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM (JACM) **32** (1985), no. 2, 374–382.