

# Consensus Problem

Alberto Montresor

## Problem Description

- $n$  processes, each of them *proposes* a value;
- they must *decide*, reaching an agreement, one of the proposed values.

## Results

- Fischer, Lynch, Paterson [FLP85]:  
”*Impossibility of Distributed Consensus with one faulty process*”  
No algorithms can solve Consensus in an asynchronous distributed system in which a single process may crash
- Chandra, Toeug [CT91, CHT92]  
”*Unreliable Failure Detectors for Reliable Distributed Systems*”  
Introduce the concept of Failure Detector (FD) Shows how Consensus can be solved in asynchronous distributed systems provided with a FD, even if processes may crash.

## Some comments

- FLP: Weak consensus specification, strong system model
- CT: Strong consensus specification, weak system model + FD

The difference is given by the FD.

## Motivation

Reaching an agreement on some decision is **the** problem in distributed computing; consider for example the election of a leader or coordinator, the agreement on a replicated sensor, etc. Several problems can be treated like a Consensus problem: Atomic Broadcast, Atomic Commitment, Group Membership.

## Challenges

Reaching an agreement in a system without failures is easy. On the other hand, reaching an agreement in a system with failures is more difficult.

Considering different levels of failure models:

- Crash
- Byzantine failures
- Omission failures

We can show that even in the simplest of the models, the problem is not solvable.

## Specification

### Basics

- Set  $\Pi = \{p_1, \dots, p_n\}$
- Each process  $p_i$  executes *propose*( $v_i$ ) proposing a value  $v_i$
- Each process  $p$  can execute an action *decide*( $v$ ) in which value  $v$  is decided.
- The set of possible values is just  $\{0, 1\}$ .

### (Uniform) Consensus

- **Termination:** Eventually, each correct process decides for some value.
- **(Uniform) Agreement:** All correct processes (all processes) that decide, decide for the same value.
- **Uniform Validity** If a process decides  $v$ , then  $v$  has been proposed by some process.
- **Uniform Integrity** Each process decides at most once.

## System model

- Asynchronous distributed system
  - No upper bound to message delays
  - No upper bound to relative process speeds
  - No synchronous clocks

- All processes know the  $\Pi$  set.
- No communication failures
- At most one crash failure, with the crashed process stopping executing actions

## Examples of bad protocols

- All processes decide 0
- All processes decide 1
- All processes decide their own value
- Since  $\Pi$  is known, all processes elect deterministically the process  $p_1$  and wait for a decision from it;
- Everybody performs a reliable broadcast to everybody; if they receive one zero, they decide for 0; otherwise, if they receive all ones, they decide for 1.
- Majority: everybody decide for the majority it receives

## Processes, the formal way

- A process  $p_i$  is an “infinite deterministic automata”
- It contains two registers:
  - $IN_i \in \{0, 1\}$ : proposed value
  - $OUT_i \in \{0, 1, \perp\}$ : decided value
- The **internal state**  $\sigma_i$  of a process  $p_i$  is given by the two registers plus other variables and information not relevant for the proof here.
- Some internal states are **initial states**; if a state  $\sigma_i$  is initial, then  $OUT_i = \perp$ .
- A **decision state** is a state in which  $OUT_i = 0$  or  $OUT_i = 1$ .
- $OUT_i$  is *write-once* (uniform integrity)
- $p$  acts deterministically according to a **transition function**

## Communication, the formal way

- Message buffer  $B$ : the set of all sent messages not yet delivered
- Messages:  $\langle p, m \rangle$ , where  $p$  is the destination,  $m$  is the message from a fixed universe  $M$
- $send_p(q, m) \Rightarrow B := B \cup \{(q, m)\}$
- $r = receive_p()$ , there are two cases (chosen in a **non-deterministic** way)
  - If  $\langle p, m \rangle \in B$ , then  $r = m$  and  $B := B - \{(p, m)\}$
  - or  $r = \perp$  (here, means empty message) and the buffer is not altered

$receive()$  may return  $\perp$  a finite number of times. But:

If  $p$  executes  $receive()$  an infinite number of times, eventually each pair  $\langle p, m \rangle$  that is included in  $B$  will eventually be received.

## Configurations, events, schedules, etc.

- **Configuration**  $C$ : global state consisting of the internal state  $\sigma_i$  of each process in  $\Pi$  and the content of the message buffer  $B$ ;
- **Initial configuration**: a configuration in which each state is an initial state and the message buffer is empty;
- **Event**: An event  $e = (p_i, m)$  brings the system from a configuration  $C$  to another one and consists of a primitive step of a single process  $p_i$ .
  - $m = receive_i()$ ;  $p_i$  receives a message from the message buffer of  $C$ ;  
 $m \in M \cup \{\perp\}$  and  $\langle p, m \rangle \in B$ .
  - Based on  $\sigma_i$  and  $m$ ,  $p$  enters in a new state
  - $p_i$  may send a finite set of messages to other processes
- Given an event  $e$  and a configuration  $C$ , then:
  - $C' = e(C)$  is the **resulting configuration** of  $e$  and  $C$ ;
  - We say that  $e$  can be **applied** to  $C$
- Given a configuration  $C$ , we say that a sequence of events  $s = e_1 e_2 \dots e_k$  is a **schedule** that can be **applied** from  $C$  if

$$C' = e_k(e_{k-1}(\dots(e_2(e_1(C)))) \dots)$$

- We say that a configuration has a **decision value**  $v$  if:  $\exists p_i \in \Pi : OUT_i = v$ .

- A **run** is a infinite schedule that can be applied to an initial configuration.
- **Admissible run**: a run in which:
  - At most one process is faulty;
  - Eventually, all messages sent to correct processes are delivered
- **Deciding run**: a run in which there is a configuration with a decision value.

## Pros and Cons

### Positive aspects (powerful system)

- Infinite state automata
- Reliable communication and broadcast
- Max 1 failure
- Set  $\Pi$  known to everybody
- Simple problem: we only consider Consensus

### Negative aspects (bad environment)

- Asynchronous system

## The main theorem

**Theorem 1.** *No Consensus protocol is totally correct if a single crash is possible.*

*Proof.* We prove the theorem by contradiction: we assume that a totally correct protocol  $P$  exists, and we derive a contradiction.

Let  $C$  a configuration and let  $V(C)$  the set of decision values in the decision configurations reachable from  $C$ :

- If  $V(C) = \{0\}$ ,  $C$  is **0-valent, univalent**
- If  $V(C) = \{1\}$ ,  $C$  is **1-valent, univalent**
- If  $V(C) = \{0, 1\}$ ,  $C$  is **bivalent**.
- If  $V(C) = \emptyset$ , not possible ( $P$  is totally correct).

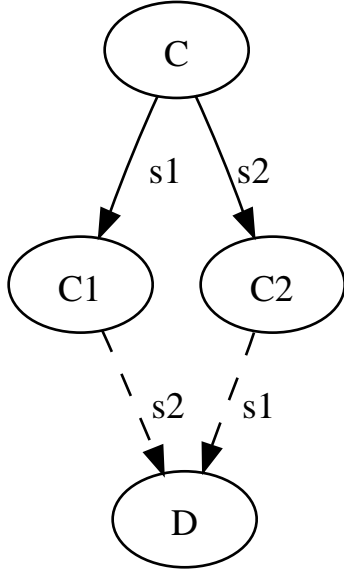


Figure 1: Lemma 1

**Proof plan**

1. We prove that  $P$  has a initial bivalent configuration
2. We prove that is always possible to go from a bivalent configuration to a bivalent configuration

Thus, given a totally correct protocol  $P$ , it is always possible to create an infinite run completely composed by bivalent configuration (i.e., in which a decision is never made).

□

**Lemma 1** (Schedule Commutativity). *Assume that from configuration  $C$  the schedules  $s_1$  and  $s_2$  bring to configurations  $C_1$  and  $C_2$ , respectively. If the set of process that executes actions in  $s_1$  and  $s_2$  are disjoint, then  $s_1$  can be applied to  $C_2$  and  $s_2$  can be applied to  $C_1$  and both lead to configuration  $C_3$ .*

$$\{p_i \mid (p_i, m) \in s_1\} \cap \{p_i \mid (p_i, m) \in s_2\} = \emptyset \Rightarrow s_2(C_1) = C_3 = s_1(C_2)$$

*Proof.* It follows from the definition of applicability, as  $s_1$  and  $s_2$  do not interact.

□

**Lemma 2** (Bivalent Initial Configuration). *P has a bivalent initial configuration.*

Stated in a more descriptive way, it means that for some initial states, the final decision is not deterministically decided by the value of the proposed values, but it depends on the order in which messages are received.

*Proof.* By contradiction, let us assume that  $P$  does not have initial bivalent configurations.

- All configuration are either 0-valent or 1-valent
- Each initial configuration could be described by a binary string of proposed values, such as  $1000\dots 0$ .
- $P$  must has both 0-valent and 1-valent initial configurations ( $00\dots 00$  and  $11\dots 11$ ); if not, validity would be violated.
- A pair of initial configurations are said **adjacent** if their strings differ by a single bit. Example:  $1000$  and  $1100$ .
- Each pair of initial configuration is linked by a chain of initial configuration, each adjacent to the following. Example  $1000 - 1100 - 1110 - 1111$ .
- Let us consider a chain of initial configuration, where the first is 0-valent and the last is 1-valent. In this chain, there should be two initial configuration adjacent to each other, one 0-valent ( $C_0$ ) and the other 1-valent ( $C_1$ ).
- Let  $p_i$  be the process whose proposed value differs in the two configuration.
- Let  $R$  be an admissible, deciding run from  $C_0$  in which  $p_i$  never executes steps (it is faulty), and let  $s$  the associated schedule.
- $s$  can be applied also to  $C_1$
- $s(C_0), s(C_1)$  differ only for the initial value of  $p_0$ , which is faulty;
- so the decision in  $s(C_1)$  is equal to  $s(C_0)$
- if this decision is 1, in reality  $C_0$  is bivalent.
- if this decision is 0, in reality  $C_1$  is bivalent.
- In both cases, we obtained a contradiction.

□

**Lemma 3** (Bivalent-to-bivalent).

- Let  $C$  be a bivalent configuration
- Let  $e = (p_i, m)$  be an event applicable to  $C$
- Let  $\hat{C} = \{s(C) \mid e \notin s\}$  be the set of configuration reachable from  $C$  without applying  $e$ ;
- Let  $\hat{D} = e(\hat{C}) = \{e(C') \mid C' \in \hat{C}\}$  be the set of configuration reachable from the configuration of  $\hat{C}$  by applying  $e$ .

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

*Proof.* By contradiction;  $\hat{D}$  does not contain bivalent configurations.

**Part 1** First, we prove that  $\hat{D}$  contains both 0-valent and 1-valent configuration. We know that:

- $\hat{D}$  does not contain bivalent configuration (by assumption)
- $e$  is applicable to  $C$  implies that  $e$  is applicable to each  $E \in \hat{C}$
- Let  $E_i$  be a  $i$ -valent configuration reachable from  $C$  ( $i = 0, 1$ );  $E_i$  exists because  $C$  is bivalent;

The situation can be depicted in this way:

$$(0\text{-valent}) \quad E_0 \xleftarrow{s_0} C \xrightarrow{s_1} E_1 \quad (1\text{-valent})$$

There are two possibilities:

- **1st case:**  $E_i \in \hat{C}$ ; the situation can be depicted like this:

$$C \xrightarrow{s_i} E_i \xrightarrow{e} F_i$$

where  $E_i, F_i$  are  $i$ -valent,  $E_i \in \hat{C}$ ,  $F_i \in \hat{D}$ .

- **2nd case:**  $E_i \notin \hat{C}$ ; the situation can be depicted like this:

$$\underbrace{C \rightarrow \bullet \xrightarrow{e} F_i \rightarrow E_i}_{s_i}$$

where  $E_i$  is  $i$ -valent;  $F_i$  is not bivalent, is not  $(1 - i)$ -valent, so it is  $i$ -valent.  $F_i \in \hat{D}$ .

## Part 2

We want to prove that: There exists two configurations  $C_0, C_1 \in \hat{C}$  where  $C_1 = e'(C_0)$  such that  $e(C_i) = D_i$  is  $i$ -valent and  $D_i \in \hat{D}$ .

In the paper: “easy induction”

There are two cases:  $e(C) \in \hat{D}$  can be 0-valent or 1-valent. Let assume that  $e(C)$  is 0-valent (the other case is symmetrical).

Let  $s$  be a schedule such that:

- $s(C) \in \hat{C}$  (means:  $e \notin s$ )
- $e(s(C))$  is 1-valent (based on Part 1, there is one)
- $s = e_1 e_2 \dots e_n$ .

$e(C)$	0-valent, $\in \hat{D}$
$e(e_1(C))$	0-valent, $\in \hat{D}$
$D_0 = e(e_2(e_1(C))) = e(C_0)$	0-valent, $\in \hat{D}$
$D_1 = e(e_3(e_2(e_1(C)))) = e(C_1)$	1-valent, $\in \hat{D}$
...	1-valent, $\in \hat{D}$
$e(s(C))$	1-valent, $\in \hat{D}$

In other words, we have found  $C_0$  and  $C_1$  of the claim.

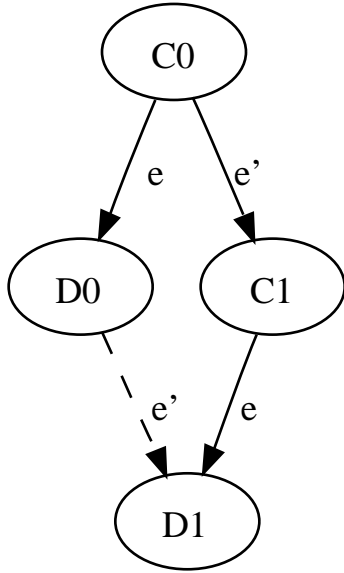


Figure 2: Lemma 2 - Part 3 -  $p_i \neq p_j$

### Part 3

Let  $C_0, C_1 = e'(C_0)$  where  $e' = (p_j, m')$ ; we know that  $C_0, C_1 \in \hat{C}$ ;  $D_i = e(C_i)$  is  $i$ -valent. There are two possibilities:

- $p_i \neq p_j$ ; it is possible to apply lemma 1, and we obtain the diamond of Figure. This is a contradiction, because any successor of a 0-valent configuration is 0-valent.
- $p_i = p_j$ . Let  $\sigma$  be a deciding schedule in which  $p_i$  never executes any event (for example, because it becomes faulty); such schedule exists because of total correctness.

From Lemma 1,  $\sigma$  is applicable to  $D_0$  and  $D_1$ . But this is a contradiction, because from  $A$  we can reach both  $D_0$  and  $D_1$ , which are 0-valent and 1-valent; so  $A$  is bivalent, but this is impossible, because  $\sigma$  is a deciding run.

What does it mean? That given a bivalent, any decision based on the observation that a given process  $p_i$  is crashed, can be contradicted by a later action of process  $p_i$ .

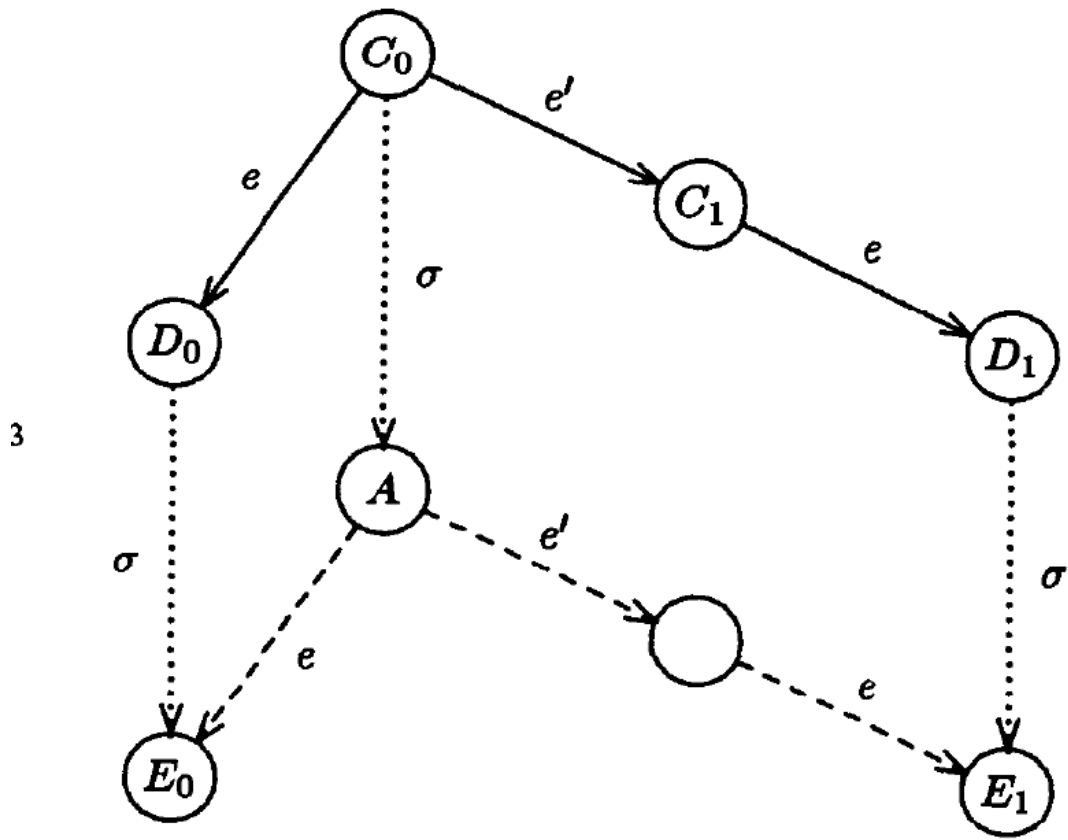


Figure 3: Lemma 2 - Part 3 -  $p_i = p_j$

**Conclusion** We prove now that given a totally correct protocol  $P$ , it is always possible to build an admissible run which is not deciding; a contradiction.

We cannot “crash” more than one process; which means that the other processes must execute an infinite number of actions.

Elements:

- A process queue  $p_1 p_2 p_3 \dots p_n$
- A message queue ordered by sending time

The building is based on stages; at each stage, we make one of the process execute an action (none of them crash).

- **Stage 0:** We select an initial bivalent configuration  $C_0$  (lemma 2)
- **Stage i:** We select an event  $e_i = (p, m)$  where
  - $p_i$  is the first process in the queue
  - $m$  is the first message for  $p_i$ ,  $\perp$  if none present
- Using Lemma 3, we build a bivalent configuration  $C_i$  starting from  $C_{i-1}$  and  $e_i$ .
- We remove the process from the front of the queue and we re-insert it at the end.

Final results:

$$C_0 \xrightarrow{s_1} \bullet \xrightarrow{e_1} C_1 \xrightarrow{s_2} \bullet \xrightarrow{e_2} C_2 \rightarrow \dots$$

Infinite run, with no process failures, in which we never decide.

□