Impossibility of Distributed Consensus

Distributed Systems 2
Matteo Franzil <matteo.franzil@studenti.unitn.it>

October 5, 2021

1 Problem Description

The problem of consensus can be formulated as following:

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

In this paper, we analyze the result given by the paper by Fischer, Lynch, Paterson, that says that no algorithms can solve consensus in an asynchronous distributed system in which a single process may crash. Another important paper in the literature is the one by Chandra and Toeug, that shows how consensus can be solved in asynchronous distributed systems provided with a failure detector, even if processes may crash.

The first paper has a strong system model, but faulty processes and weak consensus model; the opposite is true for the second paper.

The point of all of this is that it's easy to reach a decision (well, at least theoretically) when the networks and the processes are perfectly functioning, but in real distributed systems, both can fail and lead to weird results. In particular, for the moment we will analyze the case of normal crash and omission failures, and not Byzantine failures. Still, we are not able to solve the problem even with normal failures!

2 Preliminaries

- $\Pi = P_1, \cdots, P_n$ is the set of processes
- 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.

Properties:

- **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 Π set.
- No communication failures
- At most one crash failure, with the crashed process stopping executing actions

We can assess that no matter what protocol, bad or good, we come up with, we can get a contradictory results. Examples of bad protocols include ones in which processes decide all 0 or 1, majority ones, and more.
Notation:

- P_i is an infinite deterministic automata
- σ_i is the internal state of the process P_i
  - $IN_i \in \{0, 1\}$ is the initial proposed value
  - $OUT_i \in \{0, 1\}$ is the decided value and is write-once
- a state is called initial if $OUT_i = \bot$, else it is a decision state

Regarding the communication, we have:

- B: a magic Internet buffer, akin to a perfect channel
- `send`$(m, q) \Rightarrow B = B \cup \{\langle q, m\rangle\}$
- $r =$ `receive()` that *nondeterministically* either outputs:
  - $r = m$ if $(p, m) \in B$, therefore $B = B - \{\langle p, m\rangle\}$
  - $\bot$ if there's nothing for the process

We must remember that we're dealing with perfect channels, so `receive()` will eventually deliver each pair to the intended destination. Finally, some more definitions on our event system:

- $C = \sigma_1 \cdots \sigma_m \cup B$ is a configuration, i.e., a global state consisting of all states of each processes and the magic buffer B. If each state is the initial state (and the buffer is empty), the configuration is called initial.
- $e = (p_i, m)$ is an event, that brings the system to a configuration $C \Rightarrow^e C'$ with a primitive step of the process itself.
- $s = e_i \cdots e_k$ is a schedule, i.e., a series of events that are orderly applied to a process: $e_k(e_{k-1}(\cdots e_2(e_1(p_i))))$
- Each configuration has a decision value v if $\exists p_i \in \Pi : OUT_i = vv \neq \bot$.
- A run is a potentially infinite schedule that
  - is called **admissible** is there is at most a faulty process and all messages are eventually delivered
  - is called **deciding** if at the end a value is decided

Our system is very powerful and extensible: we consider only 0/1 decisions (but can be easily adapted) and the set of processes is globally known. Still, our system is asynchronous, and this will cause some issues.

3 Theorem

Theorem 1 (Impossibility of Distributed Consensus). *No consensus protocol is totally correct if a single crash is possible.*

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

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

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

We will prove this theorem with the following 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

We want to prove that given any protocol we can create an infinite loop of going from bivalent to bivalent configuration and thus a decision is never made.

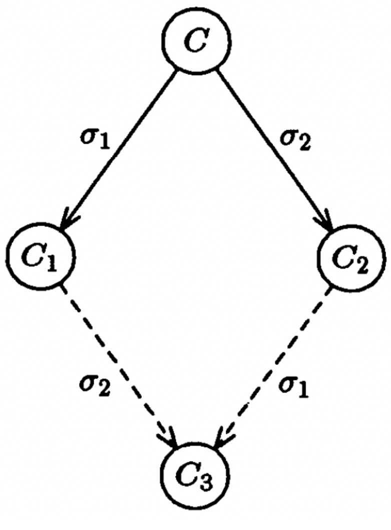

Figure 1 Lemma 1

Lemma 1.1 (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 sets of processes that execute 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 the same configuration D:*

$$\{p_i | (p_i, m) \in s_1\} \cap \{p_j | (p_j, m) \in s_2\} = \varnothing \Rightarrow s_2(C_1) = D = s_1(C_2)$$

The proof follows from the definition of applicability, as s_1 and s_2 do not interact.

Theorem 2 (Bivalent Initial Configuration). *P has a bivalent initial configuration. Informally, 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 initial configurations are either 0-valent or 1-valent. Each initial configuration could be described by a binary string of proposed values, such as $1000\cdots0$. P must have both 0-valent and 1-valent initial configurations ($00\cdots00$ and $11\cdots11$); 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 configurations is linked by a chain of initial configurations, each adjacent to the next. E.g. 0000 - 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 configurations 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 configurations. Let R be an admissible, deciding run from C_0 in which p_i never executes steps (it is faulty), and let s be the associated schedule. We know that s can be applied also to C_1. Indeed, $s(C_0)$, $s(C_1)$ differ only for the initial value of p0, 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, else if this decision is 0, in reality C_1 is bivalent. In both cases, we obtained a contradiction.

$\square$

Theorem 3 (Bivalent Configuration to Bivalent Configuration)**.** *P can move from any bivalent configuration to any bivalent configuration.*

- *Let C be a bivalent configuration*
- *Let $e = (p_i, m)$ be an event applicable to C*
- *Let $\hat{C} = \{s(C) | e \notin s\}$ be the set of configurations reachable from C without applying e;*
- *Let $\hat{D} = e(\hat{C}) = \{e(C_0) | C_0 \in \hat{C}\}$ be the set of configurations reachable from the configurations 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 configurations. We know that:

- $\hat{D}$ does not contain any bivalent configurations (by contradiction)
- 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:

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

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, $E_i \in \hat{C}, Fi \in \hat{D}$.

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

$$C \rightarrow \bullet \xrightarrow{e} F_i \rightarrow E_i$$

  where the whole succession is a schedule s, E_i is i-valent; F_i is not bivalent, is not $(1-i)$-valent, so it is i-valent. Therefore, $F_i \in \hat{D}$.

Part 2.

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

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)
- $s = e_1 e_2 \cdots e_n$.

We get:

$$\begin{array}{ll}
e(C) & \text{0-valent}, \in \hat{D} \\
e(e_1(C)) & \text{0-valent}, \in \hat{D} \\
D_0 = e(e_2(e_1(C))) = e(C_0) & \text{0-valent}, \in \hat{D} \\
D_1 = e(e_3(e_2(e_1(C)))) = e(C_1) & \text{1-valent}, \in \hat{D} \\
\cdots & \text{1-valent}, \in \hat{D} \\
e(s(C)) & \text{1-valent}, \in \hat{D}
\end{array}$$

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

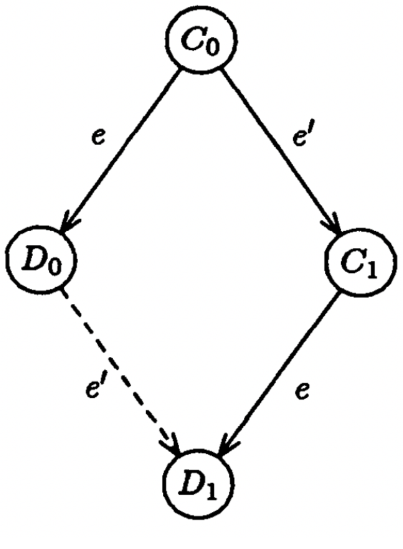

Figure 2 Lemma 2, part 3, case $p_i \neq p_j$

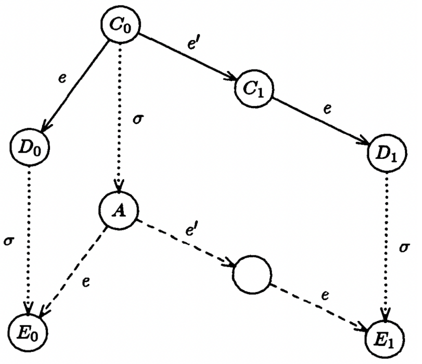

Figure 3 Lemma 2, part 3, case $p_i = p_j$

Part 3.

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

- $p_i \notin p_j$: it is possible to apply Lemma 1, and we obtain the diamond of Figure 2. This is a contradiction, because any successor of a 0-valent configuration is 0-valent.
- $pi = pj$: Let σ 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, σ 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 σ is a deciding run.

What does it mean? That given a bivalent configuration, 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. □

4 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 $p_1 p_2 p_3 \cdots p_n$
- A message queue ordered by sending time. The building is based on stages; ad 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 theend.

Final results:

$$C_0 \xrightarrow{s_1} \bullet \xrightarrow{e_1} C_1 \xrightarrow{s_2} \bullet \xrightarrow{e_2} C_2 \to \cdots$$

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