Consensus in Asynchronous distributed Systems

November 29, 2018

Agreement among the processes in a distributed system is a fundamental requirement for a wide range of applications. Many forms of coordination require the processes to exchange information to negotiate with one another and eventually reach a common understanding or agreement, before taking application-specific actions. A classical example is that of the *commit* decision in database systems, wherein the processes collectively decide whether to *commit* or *abort* a transaction that they participate in.

Our reference scenario consist of *asyncrhonous message passing* distributed systems. In such systems each process performs operations according to its own clock, and no assumption is made about the relative speed of different clocks. Processes communicate uniquely by sending messages which are neither corrupted nor lost, but there is no guarantee on the time of delivery. This scenario is actually common in real systems such as large-scale distributed data storage systems.

The *consensus problem* is to get a group of processes in a distributed system to agree on a value. A *consensus protocol* is an algorithm that produces such an agreement. A consensus protocol can be described as follows. Initially, each process proposes a value v taken from a given set of values V. At the end of the protocol, all processes must agree on a single value, called the *decision value*, or simply decision. This decision value must have been proposed by one of the processes. For example, the processes can be replica managers, and each replica managers has a copy of a shared data whose logical clocks can't determine which copy is the latest, i.e., the copies have been generated concurrently. The replica managers, before replying to a read operation, must decide which copy must be returned, so they need to reach a consensus on the decision.

The consensus problem in a distributed system is trivial in the absence of faults but that becomes more interesting if one is looking for a fault-tolerant solution. In this case we assume *crash failures*, occurrying when a process stops functioning forever ("it dies"). A process who dies is said to be *faulty*, and *correct* otherwise.

Formally, a consensus protocol is an algorithm solving the consensus problem that must satisfy the following conditions:

- **Termination**: All correct processes must eventually decide a value.
- **Integrity**: At most one decision per process.

- **Agreement**: All correct processes must choose the same decision value.
- **Validity**: The value decided by a process must have been initially proposed[1].

Now that the problem is formalized the question whether a protocol satisfying these conditions exists becomes a mathematical question, amenable to a proof. In 1985, M. Fisher, N. Lynch and M. Paterson have proved the following theorem:

Theorem (FLP). *In message passing asynchronous distributed systems, there exists no deterministic algorithm that solves the consensus problem even in presence of a single faulty process, that is, no consensus protocol can tolerate one (on more) crash failures.*

The model of the distributed system used in the FLP theorem is an asynchronous message passing system composed by n processes. The processes communicate by exchanging messages, and the communication mechanism is secure and point-to-point, i.e., messages are neither corrupted nor lost and are sent directly to the recipient. The systems is asynchronous, i.e., processes have no clocks and there is no guarantee meither on the delivery of the messages nor on the amount of time processes may take to receive, process and respond to a message.

In order to prove the FLP Theorem, we need a formalization of the model. Each process p_i has a one-bit input register x_i with values in $\{0, 1\}$ and a output register y_i with values in $\{0, 1, ?\}$. The *(local)state* s_i of process p_i comprises the value of x_i and the value of y_i (and its program counter, and its internal storage, etc.). In the *initial state*, a process p_i has $x_i = 0$ or $x_i = 1$ and $y_i =?$. A process p_i is in a *decision state* when $y_i = 0$ or $y_i = 1$. The output register y_i is *writable only once*, i.e., whenever a decision has been taken, it can't be changed (to enforce the integrity condition).

This model is extremely simple (values are uniquely binary digits, i.e., $\{0, 1\}$), but since we are looking for an impossibility result, if such result holds in such a simple model, it will hold in more constrained models as well.

Message passing is modeled as an abstract data structure called *message buffer* B, which is a multi-set, i.e., a set where more than one of any element is allowed. A *message* m is a pair (p, c) where p is the recipient of m and c is some message value from a message set. The message buffer stores messages that have been sent but not yet delivered. It provides two operations

- `q.send(m)`: places $m = (p, c)$ in the message buffer.
- `q.receive()`: if there are no messages for q in the message buffer, q will simply receive a *null marker* $\perp$. If there are messages for q in the message buffer, either delete some (random) message $m = (q, c)$ from the buffer and returns m to q or return simply $\perp$ and leave the buffer unchanged.

[1]The idea behind validity is that we want to exclude trivial solutions that just decide 'No' whatever the initial set of values is. Such an algorithm would satisfy termination and agreement, but would be completely vacuous, and no use to use at all.

The `q.receive()` operation captures the idea that messages may be delivered non-deterministically and that they may be received in any order, as well as being arbitrarily delayed as per the asynchronous model. The only requirement is that calling `q.receive()` infinitely many times will ensure that all messages in the message buffer are eventually delivered to q. This means that messages may be arbitrarily delayed, but not completely lost. That is, every message is received within a finite number of attempts.

Processes are modeled as deterministic finite state machines. A process p works in *steps*; in each step, a correct process p goes on forever performing the following actions:

1. performs `p.receive()`,
2. computes in finite time its next local state,
3. sends a finite number of messages.

A *protocol* consists of a collection of finite state machines, one for each process. Form now on, we will assume that we are given a protocol.

A *configuration* C (or global state) of the system consists of the internal state of each process and the content of the message buffer. An *initial configuration* is a configuration in which each process starts at an initial state and the message buffer is empty.

An *event* e is a pair $e = (p, m)$ where p is a process and m is a message or the null marker $\perp$. An event $e = (p, m)$ can be applited to a configuration C if m is in the message buffer of if $m = \perp$, i.e., $(p, \perp)$ is always applicable. This will move the distributed system to a new configuration, which we write as $e(C)$.

A protocol "moves" the system among configurations, forming what is called an *execution*: a possibly infinite sequence of events from a specific initial configuration. Since the receive operation is non-deterministic there are many different possible executions from an initial configuration. To show that some algorithm solves the consensus problem one has to show that for any possible execution, the termination, agreement, integrity and validity properties hold.

Definition 1. Given a protocol, a *run* $\sigma = e_1 e_2 e_3 \cdots e_n$ (from configuration C) is a (possible empty) sequence of events that can be applied in turn, starting from C, which we write as $\sigma(C)$.

Let e be an event; a run σ is said to be *e-free* if it does not contain e. Finally, a configuration is *accessible* if it can be reached from an initial configuration. From now on, when we talk of a configuration we will mean an accessible configuration.

Configuration can be classified into the following three categories:

- *0-valent* configuration, if some process has decided 0 or if in all configurations which are accessible from it the decision value is 0;
- *1-valent* configuration, if some process has decided 1 or if in all configurations which are accessible from it the decision value is 1;

- *bivalent* configuration, if for some configuration(s) accessible from it the decision value is 0 and for other(s) the decision value is 1.

A configuration is *univalent* if it is either 0-valent or 1-valent.

Non faulty processes take infinitely many steps in a run (presumably eventually just receiving $\emptyset$ once the algorithm has finished its work), Otherwise the process is faulty. We formalize the concept of crash fault with the following definition.

Definition 2. A run is *admissible* if every process, except at most one, takes infinitely many steps.

An admissible run is a run where at most one process is faulty (capturing the failure requirements of the system model), and every message is eventually delivered. Without loss of generality we shall consider admissible runs only.

Definition 3. A run is *undecidable* if every process, except at most one, takes infinitely many steps without deciding.

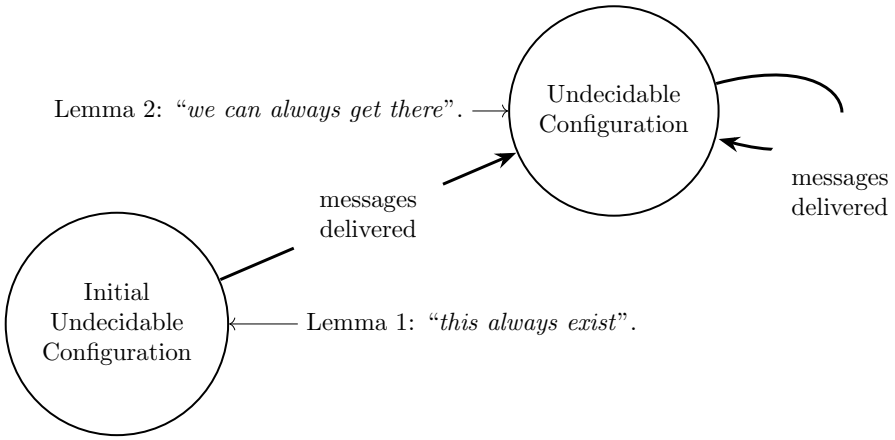
A *decidable run* is a run where some process eventually decides, i.e., some process enters a decision state.

Hence, a consensus protocol is *correct* if every admissible run is a deciding run. Alternatively, if, given a protocol, we can exhibit a undecidable run, that protocol is not correct.

The FLP theorem can then be stated as follows.

Theorem. *No correct consensus protocol exists.*

The idea behind it is to show that there is some admissible run, i.e., one with only one processor failure and eventual delivery of every message, that is not a deciding run, i.e., in which no processor eventually decides and the result is a protocol which runs forever (because no processor decides).



Lemma 1 (Initial Configuration)**.** *There exists a bivalent initial configuration.*

Proof. Let C_0 be the initial configuration where $x_i = 0$ for all processes and, for $1 \leq j \leq n$, let C_j be the initial configuration where $x_i = 1$ for the first $1 \leq i \leq j$ processes and $x_i = 0$ for the remaining processes. Bt the validity property, C_0 is 0-valent, and C_n is 1-valent. We will show that at least one of the remaining $C_1, \ldots, C_n$ configurations is bivalent by contradiction. If not, let j be the lowest number such that C_j is 1-valent. Obviously, C_{j-1} is 0 valent. Since our protocol tolerates a crash fault, we can assume that process p_j is dead from the very beginning. By the termination property, there is still a finite run σ from C_j not involving p_j such that a decision is made. But x_j is the only input register where C_{j-1} and C_j differ. Therefore σ is a run also for C_{j-1} and, in particular, it will lead to the same decision there. This will contradict either C_{j-1} being 0-valent or C_j being 1-valent.

□

So this lemma shows that there exists at least one bivalent initial configuration. Now we prove two technical lemmas.

Commutativity Lemma. *Let σ_1 and σ_2 be two runs from C such that the set of processes executing steps in σ_1 is disjoint from the set that execute steps in σ_2. Then $\sigma_1\sigma_2$ and $\sigma_2\sigma_1$ can both be applied from C, and they lead to the same configuration, i.e., $\sigma_1(\sigma_2(C)) = \sigma_2(\sigma_1(C))$.*

Proof. We proceed by induction on $k = \max(|\sigma_1|, |\sigma_2|)$. If $k = 1$, we want to prove that $e_1(e_2(C)) = e_2(e_1(C))$. Suppose $e_1 = (p_1, m_1)$ and $e_2 = (p_2, m_2)$. Since e_1 can be applied to C, it means either m_1 is $\emptyset$ or (p_1, m_1) is in the message buffer B. The same is for e_2. Because $p_1 \neq p_2$, e_1 can be applied to $e_2(C)$ and e_2 can be applied to $e_1(C)$. Let $C_1 = e_1(e_2(C))$ and $C_2 = e_2(e_1(C))$. Then the state of the message buffer is the same in C_1 as in C_2. The states of all processes are the same in C_1 and C_2 as well. Thus $C1 = C2$.

Now we distinguish three cases.

1. $|\sigma_1| = k + 1$ and $|\sigma_2| \leq k$.
   Suppose that the first event in σ_1 is e and $\sigma_1 = (\sigma, e)$. Then $\sigma_1(\sigma_2(C)) = \sigma(e(\sigma_2(C))) = \sigma(\sigma_2(e(C))) = \sigma_2(\sigma(e(C))) = \sigma_2(\sigma_1(C))$.

2. $|\sigma_1| \leq k$ and $|\sigma_2| = k + 1$.
   As in the previous Case.

3. $|\sigma_1| = k + 1$ and $|\sigma_2| = k + 1$.
   Suppose the first event in σ_2 is e and $\sigma_2 = (\sigma, e)$. Then $\sigma_1(\sigma_2(C)) = \sigma_1(\sigma(e(C))) = \sigma(\sigma_1(e(C))) = \sigma(e(\sigma_1(C))) = \sigma_2(\sigma_1(C))$ (we used Case 1 here).

□

Delayed Message Lemma. *Let C be a configuration, and $e = (p, m)$ is an event that can be applied to C. Let W be the (possible empty) set of e-free*

configurations reachable from C without faults; then e can be applied to any configuration in W.

Proof. Event e is always applicable in W since e is applicable to C, W is the set of configurations reachable from C without faults and messages can be delayed arbitrarily long. □

Now we are ready to prove that we can keep the system in a bivalent state. All we need is the following lemma.

Lemma 2. *Let C be a bivalent configuration, and let $e = (p, m)$ be an event that is applicable to C. Let W be the (possibly empty) set of configurations reachable from C without doing e. Let V be the set of configurations of the form $e(C')$, where $C' \in W$. Then V contains a bivalent configuration.*

Proof. The proof is by contradiction: we assume that V contains only univalent configurations, we prove that V contains both 0-valent and 1-valent configurations V_0 and V_1, we prove that W contains two configurations C_0 and C_1 that respectively lead to V_0 and V_1 by applying event e and derive a contradiction.

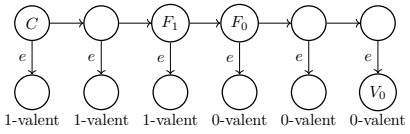
We start from a bivalent configuration C that we know to exists because of Lemma 1. We will need three intermediate claims.

Claim 1. "There is a 0-valent (resp. 1-valent) configuration F_0 (resp. F_1) reachable from C that includes event e".
Since C is bivalent, there is a 0-valent configuration F_0 such that $\sigma(C)$. If the run δ contains e, we are done, and $F_0 = \sigma(C)$; otherwise, by the delayed message lemma, e can be applied to $\sigma(C)$ and $F_0 = e(\sigma(C))$ is a 0-valent configuration. □

Claim 2. "V contains at least a 0-valent (resp. 1-valent) configuration V_0 (resp. V_1)."
Consider the F_0 configuration of Claim 1, where $F_0 = \sigma(C)$, and let δ be the prefix events of σ whose last event is e, hence $\delta(C) \in V$. Since V does not contain bivalent state, any event in $\sigma - \delta$ applied to $\delta(C)$ must be univalent, and since F_0 is 0-valent, $V_0 = \delta(C)$ is 0-valent. □

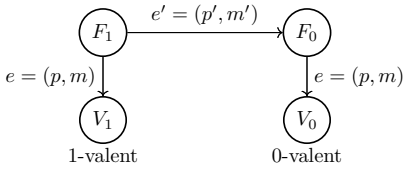
So far we have shown that V contains both 0-valent and 1-valent configurations V_0 and V_1. Now we prove the W contains two neighbor configurations F_0 and F_1 leading respectively to V_0 and V_1. Two configurations are neighbors if they are separate by just a single event d, i.e., either $F_0 = d(F_1)$ or $F_1 = d(F_0)$.

Claim 3. W contains a configuration F_0 and configuration F_1 such that $V_0 = e(F_0)$, $V_1 = e(F_1)$ and F_0 and F_1 are neighbors.
The proof is by construction. Without loss of generality we assume $e(C)$ is 1-valent. We know that, starting from the bivalent configuration C, we can reach a configuration F_0 such that $e(F_0)$ is 0-valent. Since step e is applicable from C then one can apply this step all configurations along the path from C to F_0. All the configurations where e is applied are in V, hence univalent. If one of them, V_1, is 1-valent we are done.



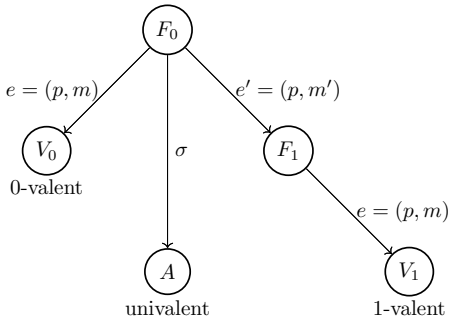
Otherwise, moving backwards to $e(C)$, we are done, since the its right configuration is 0-valent. □

So now we are in the following situation.



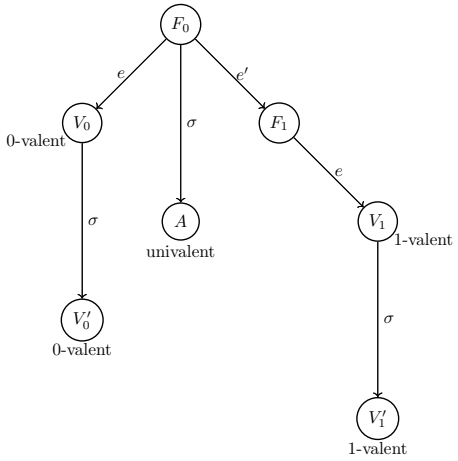
If $p \neq p'$ then events e and e' do not interact, and, by the Commutativity Lemma, event e' can be applied to V_1. However we reach a contradiction, since a 1-valent configuration (V_1) cannot lead to a 0-valent configuration (V_0). Hence we must have $p = p'$.

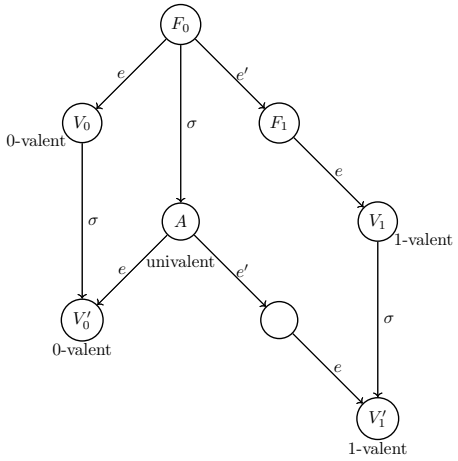
Let σ be a run that can be applied to F_0 such that all processes decide excpet for p, that does not take any step in σ (we must be able to tolerate one crash fault). Let $A = \sigma(F_0)$. By the Validity Property, all non-faulty processes must decide, hence A must be univalent.



Since p takes no step in σ, by the Commutativity Lemma the events of the run σ can be applied to both V_0 and to V_1, leading, respectively, to the 0-valent and 1-valent configurations V_0' and V_1'.

Now we can apply e to A, leading to $e(A) = e(\sigma(F_0)) = \sigma(e(F_0)) = \sigma(V_0) = V_0'$, hence A is 0-valent. But both e and e' can be applied to A, since p did not take any step in σ, leading to $e'(e(A)) = e'(e(\sigma(F_0))) = e'(\sigma(e(F_0))) = \sigma(e'(e(F_0))) = \sigma(e(e'(F_0))) = \sigma(e(F_1)) = \sigma(V_1) = V_1'$, hence A is 1-valent, and this is a contradiction.





□

Now we are ready to prove the FLP Theorem. We first construct as follows an admissible run by starting with an initial bivalent configuration C_0, whose existence is guaranteed by Lemma 1. Let $p_1, \ldots, p_n$ be any ordering of the precesses. Pick any applicable event $e_1 = (p_1, m_1)$ and apply Lemma 2, moving into another bivalent configuration $C_1 = e_1(\sigma_1(C_0))$. Note that e_1 can be selected such that m_1 was the first event ever sent to p_1 or $\perp$ if there is none. The, apply Lemma 2 again by picking an event $e_2 = (p_2, m_2)$ applicable to C_1 with $p_1 \neq p_2$, obtaining another bivalent configuration $C_2 = e_2(\sigma_2(C_1))$. As before, m_2 can be the first message ever sent to p_2 still present in the message buffer or $\perp$ if there is none. By proceeding in a round robin fashion – making process p_i perform step $e_{i+kn}(k \geq 0)$ – we obtain an undecidable run.