Consensus in Asynchronous distributed Systems

November 23, 2017

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.

Informally, a consensus algorithm 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.

More formally, a consensus protocol must exhibit the following properties:

- **Termination**: All correct processes must eventually decide a value.
- **Integrity**: At most one decision per process.
- **Agreement**: All processes that decide (correct or not) must decide the same value.
- **Validity**: The value decided by a process must have been initially proposed.

The idea behind integrity 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.

In 1985, M. Fisher, N. Lynch and M. Paterson have proved the following theorem:

Theorem. *There exists no deterministic algorithm that solves the binary consensus problem even in presence of a single faulty process.*

where a binary consensus problem involves processes that have uniquely two possible decision values, 0 and 1. This result is known as the "**FLP impossibility theorem**".

The model of the distributed system used in the FLP theorem is an asynchronous broadcast system. We consider distributed systems where processes can communicate and synchronize by exchanging messages (*message-passing* model). The communication model is asynchronous, i.e., there is no upper bound on the amount of time processors may take to receive, process and respond to an incoming message. Communication links between processors are assumed to be reliable. It is well known that given arbitrarily unreliable links no solution for consensus could be found even in a synchronous model. Processors are allowed to fail according to the crash fault model – this simply means that processors that fail do so by ceasing to work correctly[1]. This model is extremely simple, 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.

An asynchronous broadcast system is composed of n *processes* usually denoted by $\Pi = \{p_1, \ldots, p_n\}$ and a broadcast channel 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..

Each process p_i has a one-bit input register x_i and output register y_i with values in $\{0, 1, \perp\}$. The *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 $y_i = \perp$. 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 (integrity).

Processes communicate by sending messages. A *message* is a pair (p, m) where p is the recipient of m and m is some message value from a message set M. The message buffer stores messages that have been sent but not yet delivered. It provides two operations

- `send(`p`, `m`)`: places (p, m) in the message buffer,
- `receive(`p`)`: delete some (random) message (p, m) from the buffer and returns m to p (we say that (p, m) is delivered) or return $\emptyset$ and leave the buffer unchanged.

The `receive(`p`)` operations 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 `receive(`p`)` infinitely many times will ensure that all messages in the message buffer are eventually delivered to p. This means that messages may be arbitrarily delayed, but not completely lost.

A *configuration* C (or global state) of the system consists of the internal state of each process and the content of the message buffer, i.e., $C = (s, B)$ with $s = (s_1, s_2, \ldots, s_n)$. An *initial configuration* is a configuration in which each

[1]There are more general failure models, such as byzantine failures where processors fail by deviating arbitrarily from the algorithm they are executing.

process starts at an initial state and the message buffer is empty. The system moves from one configuration to the next by a *step* which consists of a processor p performing `receive(`p`)` and moving to another configuration, i.e.:

1. Let $C = (s, B)$ be a configuration,
2. p performs `receive(`p`)` on the message buffer in B of C,
3. p delivers a value $m \in M$ or $\emptyset$,
4. based on its local state in C and the received message m, process p enters a new state and sends a arbitrary but finite number of messages,

Each step is therefore uniquely defined by the message that is received (possibly $\emptyset$) and the process p that received it. That pair is called an *event* (equivalent to a message) and denoted by e. Configurations move from one to another through events. An event e can be applied to a configuration $C = (s, B)$ if either m is $\emptyset$ or $(p, m) \in B$, i.e., (p, m) is in the message buffer. $C' = e(C)$ means that if we apply event e to configuration C we move to configuration C'.

A consensus protocol "moves" the system among configuration, 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 must hold.

A particular execution σ, defined by a possibly infinite sequence of events from a initial configuration, is called a *run* (or a schedule). $C' = \sigma(C)$ means that if we apply the sequence of events in σ to configuration C we move to configuration C'.

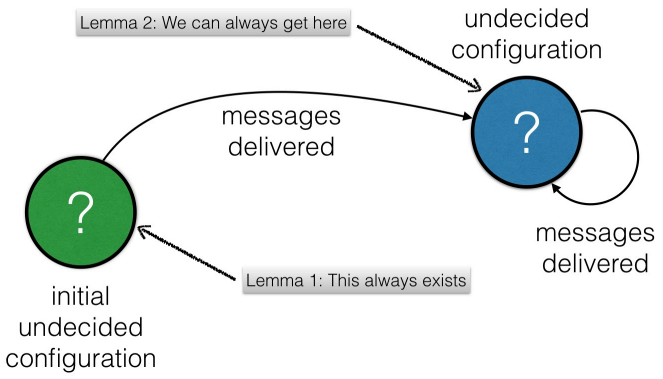
Non faulty processes take infinitely many steps in a run (presumably eventually just receiving $\emptyset$ once the algorithm has finished its work). Otherwise the process is faulty.

An *admissible run* is a run where

1. at most one process is faulty (capturing the failure requirements of the system model), and
2. every message is eventually delivered.

A *decidible 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. The FLP theorem can then be stated as "*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).



Before starting with the proof, we need a final definition, the *valency* of a configuration. Let C be any configuration. Let V be the set of decision values of configurations reachable from C, i.e., the set of all decision values of all non-faulty processes in a configuration reachable from C. So, V can be:

- $\{0\}$, then C *0-valent*;
- $\{1\}$, then C *1-valent*;
- $\{0, 1\}$, then C *bivalent*;

A configuration that is not bivalent is said *univalent*, i.e., either 0-valent or 1-valent. A 0-valent configuration necessarily leads to decision 0, and 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. In the following, we restrict ourselves to the case of two processes and binary decision values.

Lemma 1. *Any consensus protocol that tolerates at least one faulty process has at least one bivalent initial configuration.*

Proof. We proceed 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, there are initial configurations such that 0 is decided and initial configurations such that 1 is decided.

We can order initial configurations in a chain of configurations, where two initial configurations are next to each other if they differ by only one value. Hence, the difference between two adjacent initial configurations is the starting value of a one process. In this chain, we will end up with a 0-valent initial configuration (referred as C_0) adjacent to 1-valent initial configuration (referred as C_1). Let be p_i the process whose initial value differs in C_0 and C_1. There

is a run σ from C_0 in which process p_i does not take any step (i.e., it does not receive not send any message) since it is faulty from the very beginning. This means that its initial value can't be observed by other processes, but in any case, all processes must eventually decide, by the termination property, and they will decide 0. But the same run can be made from C_1 too, since no process has ever heard about process p_i, so all non-faulty processes should reach the 0 deciding state, since nothing else changes. But this is a contradiction, since C_1 is 1-valent. □

So this lemma shows that there exists at least one bi-valent initial configuration.

Now we prove two technical lemmas.

Commutativity Lemma. *Let σ_1 and σ_2 be two schedules such that the set of processes executing steps in σ_1 are disjoint from the set that execute steps in σ_2. Then for any configuration C that σ_1 and σ_2 can both be applied, we have $\sigma_1(\sigma_2(C)) = \sigma_2(\sigma_1(C))$.*

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 system 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 system 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 set of configurations that is reachable from C without applying e and 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, as stated in 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 set of configurations reachable from C without doing e and without failing any process. 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 four intermediate claims.

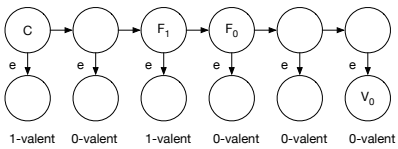
Claim 1. Let e be an event that is applicable to C. Hence, there is a 0-valent configuration F_0 such that $F_0 = \sigma(C)$ and the sequence σ contains event e.
Configuration C is bivalent, so there must be a 0-valent configuration $C_0 = \delta(C)$ reachable from C. If the sequence δ contains e, we are done, $C_0 = F_0$ and $\sigma = \delta$, otherwise, by the delayed message lemma, $F_0 = e(C_0)$ and $\sigma = (\delta, e)$.

Claim 2. V contains a 0-valent configuration V_0.
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, $V_0 = \delta(C)$.

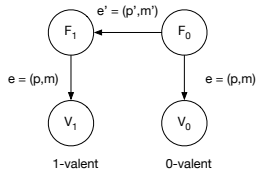
Claim 3. V contains a 1-valent configuration V_1.
Proceed as per Claim 2.

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 $F_1 = d(F_0)$.

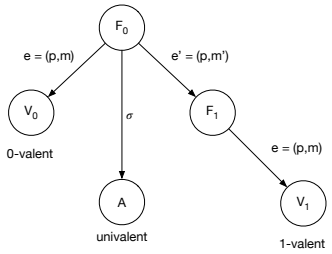
Claim 4. W contains a configuration F_0 and configuration F_1 such that $V_0 = e(F_0)$, $V_1 = e(F_1)$ and F_0 and F_1 are neighbors.
Without loss of generality we assume $e(C)$ is 1-valent. We know that, starting from the bivalent configuration C, we can reach a configuration F_0 such that $e(F_0)$ 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 univalent, 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.



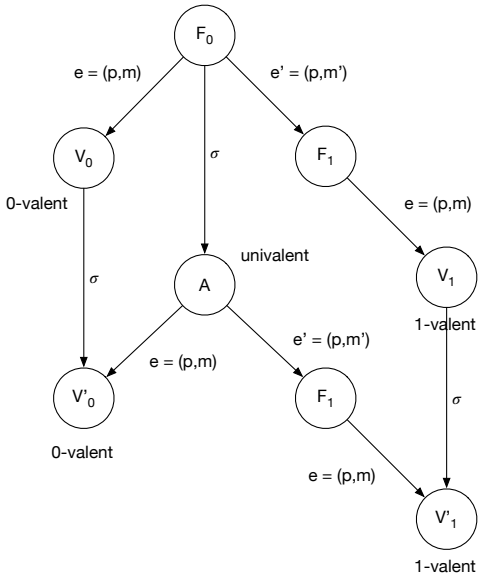
If $p \neq p'$ then events e and e' do not interact, and, by the Commutativity Lemma, event e' can be applied to V_0. However we reach a contradiction, since a 0-valent configuration cannot lead to a 1-valent configuration. Hence we must have $p = p'$. Let σ be a schedule that can be applied to F_0 such that all processes decide expect 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, σ can be applied to 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 $V_0' = e(A) = e(\sigma(F_0)) = \sigma(e(F_0)) = \sigma(V_0)$, 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 $e'(e(A)) = e'(e(\sigma(F_0))) = e'(\sigma(e(F_0))) = \sigma(e'(e(F_0))) = \sigma(e(e'(F_0))) = \sigma(e(F_1)) = \sigma(V_1)$, hence A is 1-valent, and this is 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 Lemma 2, 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.

□


F_0
e = (p,m)
e' = (p,m')
V_0
σ
F_1
0-valent
e = (p,m)
univalent
σ
A
V_1
1-valent
e' = (p,m')
e = (p,m)
V'_0
F_1
σ
0-valent
e = (p,m)
V'_1
1-valent