

## UNSOLVABILITY OF CONSENSUS IN ASYNCHRONOUS SYSTEMS

In these notes we present an important negative result, which states that Consensus is unsolvable in asynchronous systems. Indeed, this is so even for a weaker version of Consensus, and under quite strong assumptions about the number and type of failures: not even a single crash failure can be tolerated! This result is known as the “FLP impossibility result” after Fischer, Lynch and Paterson, the three researchers who proved it in 1983.

### The Very Weak Consensus problem

Assume that the possible proposals of the processes are 0 and 1 (this assumption simplifies the discussion and strengthens the impossibility result). Throughout these notes,  $v$  stands for either 0 or 1, and  $\bar{v} = 1 - v$ . Consider the following variation of Consensus, called the *Very Weak Consensus problem*:

- *Uniform Agreement*: No two processes decide different values.
- *Nontriviality*: Both 0 and 1 are possible decisions.
- *Weak Termination*: Some correct process eventually decides.

These conditions differ from those of the Consensus problem in two respects: Instead of Validity, here we simply require that 0 and 1 both be possible decisions. Furthermore, we only require that *some* correct process decide (Consensus requires all correct processes to decide).

### The Model

We describe a model of asynchronous computations. The salient feature of asynchronous systems that we wish to capture in this model is the possibility of *arbitrary finite* delays in process execution and message delivery.

First we describe the model in somewhat informal terms; the formal details will be given later. The system consists of a set of processes and a message buffer that contains a set of pairs of the form  $(m, q)$ . The existence of this pair in the message buffer indicates that message  $m$  has been sent to process  $q$ , and  $q$  has not yet received  $m$ . A process is a (possibly infinite-state) automaton. The automaton’s computation proceeds in steps. In each step the process performs the following actions atomically:

- it receives a single message previously sent to it or a “null” message, denoted  $\lambda$ , meaning that there is no message to be received yet; based on the message it receives, the process
  - sends messages to other processes; and
  - changes its state.

The message actually delivered to the process  $p$  taking a step is chosen *nondeterministically* from among the messages sent to  $p$  that are presently in the message buffer, and the null message  $\lambda$ . The null message may be delivered *even if* there are messages of the form  $(m, p)$  in the message buffer: The fact that  $(m, p)$  is in the message buffer indicates that  $m$  has been sent to but not yet received by  $p$ . Since messages may experience arbitrary delays,  $(m, p)$  may remain in the message buffer for an unbounded amount of time before it is received. Though message delays are arbitrary, we also want them to be finite. In other words, we want to consider message buffers that are

not lossy (a lost message being one that is infinitely delayed). We model this by introducing a fairness assumption: Every message sent will eventually be delivered, provided its recipient makes “sufficiently many” attempts to receive it. All this will be made more precise below. We also remark that the nondeterminism arising from the choice of the message to be delivered reflects the asynchrony of the message buffer — it is not due to nondeterministic choices made by the process. A process is deterministic in the sense that the set of messages it sends and the new state it enters are uniquely determined by its present state and the message it receives during the step.

We now describe the model formally. Let  $\Pi$  be a finite set of *process names*, and  $\mathcal{M}$  be a (possibly infinite) set of *messages*. We assume that  $\lambda$  is a symbol such that  $\lambda \notin \mathcal{M}$ , and we use  $\mathcal{M}^\lambda$  to denote the set  $\mathcal{M} \cup \{\lambda\}$ . An *algorithm* consists of  $\Pi$  and, associated with each  $p \in \Pi$ , an automaton  $(\Sigma_p, I_p, \mu_p, \delta_p)$ , where:

- $\Sigma_p$  is a (possibly infinite) set of *states*;
- $I_p$ , which is a subset of  $\Sigma_p$ , is a set of *initial states*;
- $\mu_p : \Sigma_p \times \mathcal{M}^\lambda \rightarrow 2^{\mathcal{M} \times \Pi}$  is the *message sending function*. Informally, this function specifies the messages that  $p$  sends when it takes a step. More precisely, if  $p$ 's present state is  $\sigma$  and  $p$  takes a step, then  $p$  sends a message  $m$  to  $q$  if and only if  $(m, q) \in \mu_p(\sigma)$ . We require that for each  $\sigma \in \Sigma_p$ ,  $m \in \mathcal{M}^\lambda$ , and  $q \in \Pi$ , there is at most one pair of the form  $(-, q)$  in  $\mu_p(\sigma, m)$ . That is, in a single step  $p$  sends at most one message to each process. Since we do not restrict the size of messages, this assumption can be made without loss of generality.
- $\delta_p : \Sigma_p \times \mathcal{M}^\lambda \rightarrow \Sigma_p$  is the *state transition function*. Informally, this function specifies the new state of  $p$  when it takes a step. More precisely, if  $p$ 's present state is  $\sigma$  and  $p$  takes a step in which it receives message  $m$ , then  $p$ 's new state is  $\delta_p(\sigma, m)$ .

In what follows we do not distinguish between a process name  $p$ , and its associated automaton; we refer to either as “process  $p$ ”.

For the sake of simplicity we assume that a message  $m$  is sent to a process at most once. This allows us to speak of the contents of the message buffer as a set, rather than a multiset. We can easily enforce this by including, in each message sent by  $p$  to  $q$ , the name of a message's sender,  $p$ , as well as a counter indicating the number of messages sent by  $p$  to  $q$  so far. Thus, this assumption does not damage generality.

We model the global state of the system as an object called a configuration. Formally, a *configuration* (of a given algorithm) is a pair  $C = (s, M)$  where

- $s : \Pi \rightarrow \cup_{q \in \Pi} \Sigma_q$ , where we require that  $s(p) \in \Sigma_p$  for each  $p \in \Pi$ ; and
- $M \subseteq \mathcal{M} \times \Pi$ .

Intuitively,  $s(p)$  is the state of  $p$  and  $M$  is the set of messages in the message buffer in the global state of the system represented by configuration  $C$ . We write **state**( $p, C$ ) to denote the state of process  $p$  in configuration  $C$ , and **buffer**( $p, C$ ) to denote the messages addressed to  $p$  in  $C$ . More precisely, if  $C = (s, M)$ , then **state**( $p, C$ ) =  $s(p)$  and **buffer**( $p, C$ ) =  $\{(m, p) : (m, p) \in M\}$ .

An *initial configuration* (of a given algorithm) is a configuration  $C = (s, M)$ , where  $s(p) \in I_p$  for each  $p \in \Pi$ , and  $M = \emptyset$ . That is, each process is in an initial state, and no messages have been sent yet in the global state of the system represented by  $C$ .

A *step* (of a given algorithm) is an element of the set  $\mathcal{M}^\lambda \times \Pi$ . We say that the step  $e = (m, p)$  is a step of process  $p$ . If  $m = \lambda$ , then  $e$  is a *null step*; otherwise,  $e$  is a *nonnull step*. An step  $e$  is *applicable* to configuration  $C = (s, M)$ , if  $e \in M$  or  $e$  is a null step. (Thus, any null step is applicable to any configuration.)

If  $e = (m, p)$  is applicable to  $C$ , we write  $e(C)$  to denote the (unique) configuration that results if the current configuration is  $C$  and  $e$  occurs. More precisely,  $e(C) = (s', M')$ , where

$$s'(q) = \begin{cases} s(q), & \text{if } q \neq p \\ \delta_p(s(p), m), & \text{if } q = p \end{cases}$$

$$M' = (M \cup \mu_p(s(p), m)) - \{e\}$$

From this definition, the following lemma follows immediately:

**Lemma 1.1.** *Let  $e, e'$  be steps of different processes that are applicable to  $C$ . Then  $e'(e(C)) = e'(e'(C))$ .*

Note that this “commutativity” result may not apply if  $e$  and  $e'$  are steps of the same process.

A *schedule*  $S$  (of a given algorithm) is a (possibly infinite) sequence of steps  $e_1 e_2 e_3 \dots$  (of the algorithm). We say that  $S$  is *applicable* to a configuration  $C$  if, for all  $i \geq 1$  (and, if  $S$  is finite,  $i \leq |S|$ ),  $e_i$  is applicable to  $C_{i-1}$  where  $C_0 = C$  and  $C_i = e_i(C_{i-1})$ . If  $S$  is finite and applicable to  $C$ , we write  $S(C)$  to denote the last configuration in this sequence. A configuration  $C'$  is *reachable* from configuration  $C$ , if there is a finite schedule  $S$  applicable to  $C$  such that  $S(C) = C'$ . A configuration is *accessible* if it is reachable from an initial configuration. Intuitively, a schedule (of a given algorithm) that is applicable to an initial configuration  $\hat{C}$  corresponds to a possible run of the algorithm starting from  $\hat{C}$ .

**Lemma 1.2.** *Let  $p$  be a process and  $C, C'$  be configurations such that for every process  $q \neq p$ , (i)  $\mathbf{state}(q, C) = \mathbf{state}(q, C')$ , and (ii)  $\mathbf{buffer}(q, C) \subseteq \mathbf{buffer}(q, C')$ . Let  $S$  be a schedule that is applicable to  $C$  and contains no step of  $p$ . Then  $S$  is applicable to  $C'$  and, furthermore, for every process  $q \neq p$ ,  $\mathbf{state}(q, S(C)) = \mathbf{state}(q, S(C'))$ .*

PROOF: A straightforward induction proves that, for every prefix  $S'$  of  $S$ , the following invariant holds: For each process  $q \neq p$ ,  $\mathbf{state}(q, S'(C)) = \mathbf{state}(q, S'(C'))$ , and  $\mathbf{buffer}(q, S'(C)) \subseteq \mathbf{buffer}(q, S'(C'))$ .  $\square$

Let  $S$  be an *infinite* schedule. A process  $p$  is *faulty* in  $S$  if there are only finitely many steps of  $p$  in  $S$ ; otherwise  $p$  is *correct* in  $S$ .

Next we define “ $t$ -admissible” schedules. These are schedules in which no more than  $t$  processes are faulty and messages are not lost. The reason we want to focus on such schedules is that the liveness properties of the algorithms we are interested in need only be satisfied *provided that* certain assumptions regarding the number of faulty processes and the reliability of communications hold. Thus, we are not interested in *all* schedules of the algorithm, but only in schedules where these assumptions are true.

Let  $S$  be an infinite schedule that is applicable to configuration  $C$ .  $S$  is  *$t$ -admissible* if

- at most  $t$  processes are faulty in  $S$ ; and
- for any prefix  $S'$  of  $S$ , if the message buffer of  $S'(C)$  contains  $(m, p)$  and  $p$  is correct in  $S$ , then  $S$  contains  $(m, p)$ .

The second requirement is the formal statement of the assumption that messages are not lost: a message sent to a correct process will eventually be received.

So far the model we have described applies to all algorithms, regardless of the problem that they are designed to solve. For Consensus algorithms, we must make some additional assumptions

to capture such predicates as “a process proposes  $v$ ” or “a process decides  $v$ ”. Specifically, we assume that for each process  $p$ ,  $I_p = \{0, 1\}$ . Intuitively, the initial state of a process indicates the value the process proposes. As a result, if the Consensus algorithm consists of  $n$  processes, it has  $2^n$  initial configurations.

We also assume that each process  $p$  has two disjoint sets of *decision states*,  $D_p^0$  and  $D_p^1$ . Intuitively, if  $\sigma \in D_p^v$ , for some  $v \in \{0, 1\}$ , then  $p$  has decided  $v$  in state  $\sigma$ . We require that each of these sets is closed under the state transition function; i.e., for all  $p \in \Pi$ , for all  $v \in \{0, 1\}$ , and for all  $m \in \mathcal{M}^\lambda$ , if  $\sigma \in D_p^v$ , then  $\delta_p(\sigma, m) \in D_p^v$ . This requirement reflects the fact that decisions are irrevocable.

The three properties of a Very Weak Consensus algorithm that is  $t$ -resilient (i.e., tolerates up to  $t$  faulty processes) can now be stated formally as follows.

- *(Uniform) Agreement:* There is no accessible configuration  $C$  and processes  $p, q$  such that  $\mathbf{state}(p, C) \in D_p^0$  and  $\mathbf{state}(q, C) \in D_q^1$ .
- *Nontriviality:* For each  $v \in \{0, 1\}$ , there is an accessible configuration  $C$  and a process  $p$  such that  $\mathbf{state}(p, C) \in D_p^v$ .
- *Weak Termination:* For each initial configuration  $C$  and each  $t$ -admissible schedule  $S^*$  that is applicable to  $C$ , there is a prefix  $S$  of  $S^*$  and a correct process  $p$  such that  $\mathbf{state}(p, S(C)) \in D_p^0 \cup D_p^1$ .

For any configuration  $C$  (of a Consensus algorithm), the *valence* of  $C$ ,  $val(C)$ , is the set of possible decision values in configurations reachable from  $C$ . More precisely,

$$val(C) = \{v \in \{0, 1\} : \exists C' \exists p \text{ such that } C' \text{ is reachable from } C \text{ and } \mathbf{state}(p, C') \in D_p^v\}$$

We say that  $C$  is *univalent* if  $|val(C)| = 1$ , and *bivalent* if  $|val(C)| = 2$ . If  $val(C) = \{0\}$  we say that  $C$  is *0-valent*, and if  $val(C) = \{1\}$  we say that  $C$  is *1-valent*. By Weak Termination, for any accessible configuration  $C$ ,  $val(C) \neq \emptyset$ ; therefore every accessible configuration is either univalent or bivalent. The Uniform Agreement property immediately implies,

**Lemma 1.3.** *Let  $C$  be an accessible configuration of a Consensus algorithm. If some process  $p$  has decided  $v$  in  $C$  (i.e.,  $\mathbf{state}(p, C) \in D_p^v$ ), then  $C$  is  $v$ -valent.*

## Proof of the FLP impossibility result

**Theorem 1.1.** *There is no one-resilient Very Weak Consensus algorithm.*

PROOF: Suppose, by way of contradiction, that there is a one-resilient Very Weak Consensus algorithm  $P$ . All configurations and schedules mentioned in this proof are relative to this algorithm. We will prove the following two facts:

**Fact 1.**  *$P$  has a bivalent initial configuration.*

**Fact 2.** *Let*

- $C$  be any bivalent configuration and  $e$  any step applicable to  $C$ ,
- $\mathcal{C} = \{S(C) : S \text{ is a finite schedule not containing } e\}$ , and
- $\mathcal{D} = \{e(C') : C' \in \mathcal{C}\}$ .

*Then  $\mathcal{D}$  contains a bivalent configuration.*

We postpone the proofs of these two facts and show how, using them, we reach a contradiction to the assumed existence of  $P$ . The strategy is to construct a 1-admissible (infinite) schedule of the algorithm in which no univalent configuration is ever reached — therefore contradicting Weak Termination. The following “algorithm” indicates how this schedule is constructed.

---

```

initialise a queue with the set of processes in any order
 $C :=$  a bivalent initial configuration  $\hat{C}$   %  $\hat{C}$  exists by Fact 1
 $S^\infty :=$  empty schedule
repeat
  let  $p$  be the process at the head of the process queue
  if the message buffer in  $C$  contains some step of  $p$  applicable to  $C$  then
    let  $e$  be the oldest such step
  else
    let  $e$  be  $(\lambda, p)$   %  $e$  is applicable to  $C$ 
  end if
  let  $S$  be a schedule such that  $e(S(C))$  is bivalent  %  $S$  exists by Fact 2
  move  $p$  to the end of the process queue
   $C := e(S(C))$ 
   $S^\infty := S^\infty \circ S \circ e$ 
forever

```

---

It is easy to see that the infinite schedule “generated” as the “limit” value of the variable  $S^\infty$  has the following properties:

- (i) no process is faulty (since each process takes infinitely many steps in it), and
- (ii) every message sent is eventually received (since every message sent to a process  $p$  will eventually become the oldest not-yet-received message sent to  $p$ ).

Therefore, the generated schedule is 1-admissible. Furthermore, it is easy to show that the following is an invariant of the loop:  $S^\infty(\hat{C})$  is bivalent. Hence the generated 1-admissible schedule has the property that for every finite prefix  $S$  of it,  $S(\hat{C})$  is bivalent, and therefore (by Lemma 1.3) no process has decided in  $S(\hat{C})$ . This contradicts Weak Termination.

We now turn to the proofs of Facts 1 and 2.

**PROOF OF FACT 1:** Assume, by way of contradiction, that all initial configurations are univalent. By Nontriviality there exist both 0- and 1-valent initial configurations. Two initial configurations are *neighbours* if they differ in the proposal (as indicated by the initial states) of exactly one process. There must exist two initial configurations  $C_0$  and  $C_1$  which are 0-valent and 1-valent, respectively, and are neighbours. (This is because we can go from any initial configuration to any other by a sequence of configurations, adjacent elements of which are neighbours.) Let  $p$  be the process in whose proposal  $C_0$  and  $C_1$  differ, and let  $S^*$  be a 1-admissible schedule applicable to  $C_0$  that has no step of process  $p$  (clearly, such a schedule exists). Since  $P$  is one-resilient, by Weak Termination, there is a prefix  $S$  of  $S^*$  and a correct process  $q$  such that  $q$  decides in  $S(C_0)$ . Because  $C_0$  is 0-valent,  $q$  decides 0 in  $S(C_0)$ . By Lemma 1.2,  $S'$  is applicable to  $C_1$ , and every process other than  $p$  has the same state in  $S(C_1)$  as in  $S(C_0)$ . In particular,  $q$  decides 0 in  $S(C_1)$ , contradicting that  $C_1$  is 1-valent. □ Fact 1

**PROOF OF FACT 2:** Assume, by way of contradiction, that all configurations  $D \in \mathcal{D}$  are univalent. By definition of  $\mathcal{C}$ ,  $e$  is applicable to every configuration in  $\mathcal{C}$ , so that for any  $C' \in \mathcal{C}$ ,  $e(C')$  is well-defined (and belongs to  $\mathcal{D}$ ).

**Claim 1.** *There are both 0- and 1-valent configurations in  $\mathcal{D}$ .*

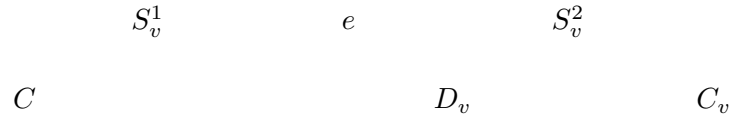
PROOF: Since  $C$  (the configuration mentioned in the statement of Fact 2) is bivalent, for each  $v \in \{0, 1\}$ , there is a finite schedule  $S_v$  that is applicable to  $C$  such that  $S_v(C)$  is  $v$ -valent; let  $C_v = S_v(C)$ .

*Case 1:*  $S_v$  does not contain  $e$ . Let  $D_v = e(C_v)$ . Clearly  $D_v \in \mathcal{D}$ .



Since  $C_v$  is  $v$ -valent,  $D_v$  is also  $v$ -valent.

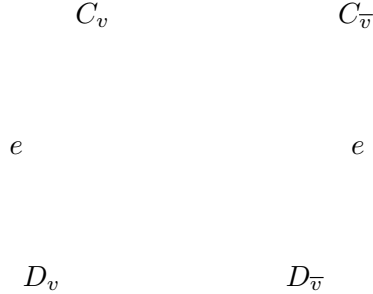
*Case 2:*  $S_v$  contains  $e$ . Then, for some schedules  $S_v^1$  and  $S_v^2$ ,  $S_v = S_v^1 e S_v^2$ .



Let  $D_v = e(S_v^1(C))$ . Clearly  $D_v \in \mathcal{D}$ . Since, by assumption, every configuration in  $\mathcal{D}$  is univalent, so is  $D_v$ . Since  $C_v = S_v^2(D_v)$  is  $v$ -valent then  $D_v$  is also  $v$ -valent.  $\square$  Claim 1

Two configurations  $F$  and  $F'$  are *adjacent* if, for some step  $f$ ,  $F = f(F')$  or  $F' = f(F)$ .

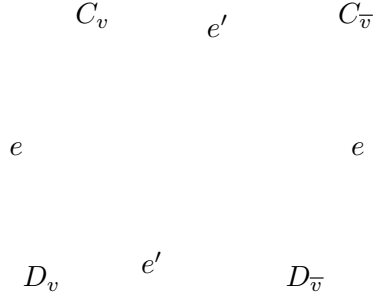
**Claim 2.** *There exist adjacent configurations  $C_0, C_1 \in \mathcal{C}$  such that  $e(C_0)$  is 0-valent and  $e(C_1)$  is 1-valent.*



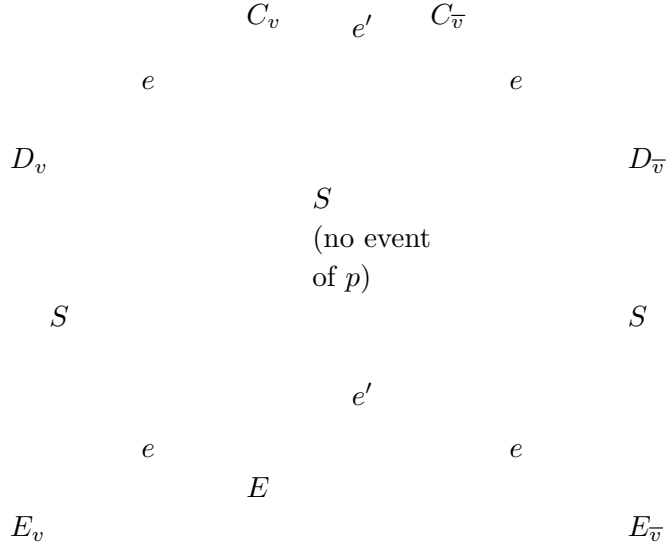
PROOF: Construct an undirected graph where the nodes represent the configurations in  $\mathcal{C}$  and there is an edge between each pair of nodes that represent adjacent configurations. Label a node  $C'$  in the graph with value  $v$  iff  $e(C') \in \mathcal{D}$  is  $v$ -valent. Since, by assumption, every configuration in  $\mathcal{D}$  is univalent, every node has a well-defined label. By Claim 1 there are both nodes labeled 0 and nodes labeled 1. Clearly the graph is connected, since there is a path between any two nodes through the node representing  $C$ . Therefore there must be adjacent nodes, one labeled 0 and the other labeled 1, as required.  $\square$  Claim 2

Let  $C_0, C_1$  be as in Claim 2 and  $e'$  be the step that makes  $C_0$  and  $C_1$  adjacent.

*Case 1:*  $e$  and  $e'$  are steps of different processes. By Lemma 1.1,  $e'(D_v) = D_{\bar{v}}$ , contradicting that  $D_v$  is  $v$ -valent.



*Case 2:*  $e$  and  $e'$  are steps of the same process. Let  $p$  be that process.



Let  $S^*$  be an infinite 1-admissible schedule that is applicable to  $C_v$  but contains no step of  $p$  (obviously such a schedule exists). By Weak Termination, there is a prefix  $S$  of  $S^*$  such that some correct process, say  $q$ , decides  $w$  in  $S(C_v)$ , for some  $w \in \{0, 1\}$ . Since  $e$  and  $e'$  are both steps of  $p$ , we have that for every process  $q \neq p$ , (i)  $\mathbf{state}(q, C_v) = \mathbf{state}(q, e(C_v)) = \mathbf{state}(q, e'(C_v))$ , and (ii)  $\mathbf{buffer}(q, C_v) \subseteq \mathbf{buffer}(q, e(C_v))$  and  $\mathbf{buffer}(q, C_v) \subseteq \mathbf{buffer}(q, e'(C_v))$ . Since  $S$  does not contain a step of  $p$ , Lemma 1.2 implies that  $S$  is applicable to  $e(C_v)$  and to  $e'(C_v)$  and, for every process  $q \neq p$ ,  $\mathbf{state}(q, S(C_v)) = \mathbf{state}(q, S(e(C_v))) = \mathbf{state}(q, S(e'(C_v)))$ . Therefore,  $q$  decides  $w$  in  $S(e(C_v))$  and in  $S(e'(C_v))$  (as it does in  $S(C_v)$ ). If  $w = v$ , this contradicts that  $e(C_v)$  is  $v$ -valent; and if  $w = \bar{v}$ , it contradicts that  $e'(C_v)$  is  $\bar{v}$ -valent. In either case, we get a contradiction. □ Fact 2

This completes the proof of Theorem 1. □

## Refinements of the result

As we have remarked, there are two sources of asynchrony in a distributed system: the processes and the communication system. The model we used to prove the impossibility result, incorporates both sources of asynchrony. Thus, the question arises: Is the impossibility really due to the fact that both processes and communication are asynchronous, or is the presence of a single source of asynchrony enough to make consensus impossible? Indeed, upon more detailed examination, we can see that communication asynchrony is manifested in two ways in the model we used: first, messages may be delayed arbitrarily and, second, they may be delivered out of order: a message sent before another may be delivered first. Thus, we may refine our question and ask whether both of these forms of communication asynchrony are needed for the impossibility result. Finally, we would like to know the influence of other aspects of the model on the possibility or impossibility of consensus. Of special interest is the question of what a process is capable of doing in an atomic step. In the model we used, a process could do quite a lot in a single step: It could broadcast a message to all processes; also, sending and receiving were accomplished in a single step. The use of such strong assumptions, makes the impossibility result all the stronger. However, if our investigation of the effect of the various sources of asynchrony leads us to the discovery that, for some types of asynchronous systems, consensus is achievable, then we would like to know if this is possible only under the same strong assumptions about what can be done within a step, or if weaker — and more realistic — assumptions would suffice.

Thus, in trying to refine the impossibility result, we can identify five “parameters” of the model that turn out to be relevant. Each parameter admits a binary choice, one of which can be thought of as “favourable” and the other as “unfavourable”.

- Process synchrony: This refers to whether there is a known bound on the relative speed of correct processes. The favourable case is when such a bound exists.
- Communication synchrony: This refers to whether there is a known bound on the time it takes for a message to arrive at its destination. The favourable case is when such a bound exists.
- Message order: This refers to whether messages are guaranteed to be delivered in the order sent.† The favourable case is when messages are guaranteed to be delivered in order.
- Broadcast or point-to-point communication: This refers to whether a process can send a message to all processes or only one process in a single atomic step. The favourable case is when messages can be broadcast.
- Atomic send/receive step: This refers to whether the sending and receiving actions are parts of a single step, or they occur in separate steps. The favourable case is when send and receive are parts of a single atomic step.

By associating the favourable choice for each parameter to the value 0 and the unfavourable to 1, we can think of each model as a vector of five binary values, and we can arrange the resulting  $2^5 = 32$  models into a “lattice of difficulty”, where one model is “more difficult” than the other if its vector dominates (i.e., is component-wise no less than) the other’s. It is easy to see that if a model  $M$  is more difficult than  $M'$  and consensus is solvable in  $M$  then it is certainly solvable in  $M'$ . Dolev, Dwork and Stockmeyer, who identified these five “parameters” and investigated the resulting models, discovered a set of “minimal” models in which consensus is solvable and which

---

† Message order preservation is stronger than the FIFO property. Message order preservation requires that if  $p_1$  and  $p_2$  send to process  $q$  messages  $m_1$  and  $m_2$  at times  $t_1$  and  $t_2$  respectively and  $t_1 < t_2$  then  $m_1$  is received by  $q$  before  $m_2$ . FIFO is the special case of this property when  $p_1 = p_2$ .

separate the models in which consensus is solvable from those in which it is not: In any model dominated by one in that set of “minimal” models consensus is solvable but in any other model it is not.

When we say that consensus is solvable we mean that there is an algorithm that can tolerate *some* (nonzero) number of faulty processes. The impossibility result we have proved states that (in the fully asynchronous model) there is no algorithm that can tolerate even *a single* faulty process. It is interesting to note that, with only one exception, when consensus becomes solvable in a model it becomes solvable in a strong sense: there are algorithms that tolerate *any* number of faulty processes — up to  $n$  of them! (In the one exception, there is an algorithm that can tolerate up to one faulty process, but no more.) The set of minimal conditions under which consensus is solvable can be summarised in the three propositions below.

**Proposition 1.** *Suppose processes are synchronous.*

- *If either communication is synchronous or message order is preserved then there is an  $n$ -resilient Consensus algorithm (even if the other three parameters are unfavourable).*
- *If communication is asynchronous and message order is not preserved then there is no 1-resilient Consensus algorithm (even if the other three parameters are favourable).*

**Proposition 2.** *Suppose communication is synchronous but processes are asynchronous.*

- *If atomic send/receive steps are available then:*
  - *If broadcast communication is available, then there is an  $n$ -resilient Consensus algorithm (even if message order is not preserved)*
  - *If communication is point-to-point, then an 1-resilient Consensus algorithm exists (even if message order is not preserved), but no 2-resilient Consensus algorithm exists (even if message order is preserved).*
- *If send and receive steps are separate and message order is not preserved then there is no 1-resilient Consensus algorithm (even if broadcast communication is available).*

**Proposition 3.** *Suppose message order is preserved but processes are asynchronous.*

- *If broadcast communication is available, then there is an  $n$ -resilient Consensus algorithm (even if communication is asynchronous and send and receive steps are separate).*
- *If communication is point-to-point, then there is no 1-resilient Consensus algorithm (even if communication is synchronous and atomic send/receive steps are available).*