Topics in Distributed Computing
Lecture 4
(Lack of) Consensus in Asynchronous Systems

Lecturer: Alessandro Panconesi Scribe: Öjvind Johansson

11 April 1997

4.1 Consensus in Asynchronous Systems

In this lecture we study the consensus problem in *message–passing, asynchronous systems*. In such systems each process runs according to its own clock and no assumption is made about the relative speeds of different clocks. Processes communicate solely by sending messages which are neither corrupted nor lost, but there is no guarantee on the time of delivery. This scenario is actually prevalent in real systems such as distributed networks. The problem we want to study is the same consensus problem defined in the context of Byzantine Agreement. That is, we have a set of $n > 1$ processes denoted by $p_1, \ldots, p_n$. For each i, p_i has an input bit, $b_i \in \{0, 1\}$, and is to decide on an output bit, d_i. The protocol should satisfy the following three conditions:

Non-triviality: If all processes have input bit 0 (1) then, the only possible decision value is 0 (1).

Agreement: All correct processes choose the same decision value.

Limited dithering: Eventually all correct processes decide.

Notice that we replaced *Limited Bureacracy* with *Limited Dithering* which leaves open the possibility that a protocol runs forever (even though the decision must be taken in a finite number of steps). This assumption will make our lower bound result stronger.

An important difference between the situation we study now and byzantine agreement is that we will be considering crash failures only. When a *crash failure* occurs a process stops functioning forever (it "dies"). Obviously, this type of faulty behaviour is much more benign than byzantine failures. A process who dies is said to be *faulty*, and *correct* otherwise. As before, processes communicate by exchanging messages. The message-passing mechanism is secure and point-to-point: messages are sent directly to the recipient and are neither corrupted nor lost. The system is *asynchronous*, i.e.

Processes have no clocks.
There is no guarantee on the time of message delivery.

Intuitively, reaching consensus in such systems is impossible because processes cannot tell whether another process is dead or just temporarily isolated from the outside world because its messages are delayed. If they wait, they might do so forever, and if they decide, they might find out that the other process already came to a different decision. This rough intuition is formalized in the following seminal result by Fischer, Lynch, and Paterson [2].

Theorem 1. *There is no 1-resilient deterministic protocol for consensus in message-passing, asynchronous systems. That is, no protocol can tolerate one (or more) crash failures.*

This result highlights the enourmous difference existing between synchronous and asynchronous systems. As we saw, if the system is synchronous, up to $\frac{n}{3} - 1$ *malicious* faults can be tolerated– there exist protocols which work correctly in spite of the arbitrary behaviour of (almost) one third of the processes. In sharp contrast to this, if the system is asynchronous not even one crash failure can be tolerated. This result does depends on the asynchronous nature of the system. For instance, the same impossibility result (and in fact essentially the same proof) extends to *shared-memory systems*, where processes communicate by asynchronously reading and writing a shared memory accessible to all (see, for instance, [1, 3]).

We now turn to the proof of the Theorem 1. First off, we give a formalization of the model. Message passing is performed by means of **Send** and **Receive** operations. There is a buffer of messages B containing messages that have been sent but not yet received. The buffer contains messages of the form $m = (p, c)$ where c is the content of the message and p is the recipient. If q performs a **Send**(m) operation, the message m is added to the buffer B. To get a message, a process q samples the buffer by executing a **Receive**(q) operation. The semantics of **Receive**(q) is specified as follows. If there are no messages for q, q will simply receive a null marker, $\perp$. If there are messages ready for q in the buffer, say $m_i = (c_i, q)$ $(i \geq 1)$, either q will receive any one of them or q will receive $\perp$. In case a message m is delivered to q, m is removed from the buffer. We make only a minimal assumption about the buffer. Namely,

$$\boxed{\text{Each message is received within a finite number of attempts.}}$$

That is, if a message $m = (c, q)$ is in the buffer and q performs infinitely many **Receive**(q) operations, sooner or later q will receive m. Notice that the message delivery discipline can be assumed to be anything (e.g. FIFO, last-in first-out, etc).

Processes are modeleled as deterministic automata, possibly with infinitely many states. A process p works in *steps*; in each step, p performs

Receive(p), computes (in a finite time) the next state, and sends a finite number of messages. Moreover, we may assume that each correct process goes on like this forever; we can always modify our protocol to satisfy this (we are assuming Limited Dithering). A *protocol* consists of a collection of automata, one for each process. From now on we shall assume that we are given a protocol P.

We continue with some definitions, which are relative to a given protocol P. A *configuration* C is a vector containing the process states and the message buffer content. An *event* is a pair $e = (p, m)$ where p is a process and m is a message or the null marker, $\perp$. The event $e = (p, m)$ is *applicable* to a configuration C if m is in the buffer, or if $m = \perp$, i.e. $(p, \perp)$ is always applicable. When $e = (p, m)$ is applicable to C, the reception of m by p defines the next step in the system, taken by p. This will give a new configuration, which we write as $e(C)$. We let $e_1e_2(C)$ denote $e_2(e_1(C))$, and so forth. We record the following definition for future reference.

Definition 1. *Given a protocol P, a* run *(from C) is a (possibly empty) sequence of events that can be applied in turn, starting from C.*

Finally, a configuration D is *accessible* if it can be reached from an initial configuration, i.e. if $D = \sigma(C)$ for some initial configuration C and run σ. From now on, when we talk of a configuration, we mean an accessible configuration. We classify configurations into three categories:

- A configuration is *0-valent* if some process has decided 0 or if in all configuration which are accessible from it the decision value is 0;
- A configuration is *1-valent* if some process has decided 1 or if in all configuration which are accessible from it the decision value is 1;
- A configuration C is *bivalent* if for some of the configuration accessible from it the decision value is 0 and for others the decision value is 1.

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

Definition 2. *We say that a run is* admissible *if every process, except possibly one, takes infinitely many steps.*

Without loss of generality we shall consider admissible runs only.

Definition 3. *We say that a run is* unacceptable *if every process, except possibly one, takes infinitely many steps without deciding.*

Clearly, if, given a protocol P, we can exhibit an unacceptable run, P is not a valid consensus protocol. We shall show that, given any protocol P, we are always able to exhibit an unacceptable run, thereby proving the impossibility of consensus in asynchronous systems. In fact, we shall exhibit an unacceptable run in which no process crashes (i.e. every process takes infinitely many steps without deciding).

Lemma 1. *There is a bivalent initial configuration.*

Proof. Let C_0 be the initial configuration where $b_i = 0$ for all i, and for $1 \leq j \leq n$, let C_j be the initial configuration where $b_i = 1$ for $1 \leq i \leq j$ and $b_i = 0$ for all remaining i. Notice that the non-triviality property implies that C_0 is 0-valent and C_n is 1-valent. We now show that at least one configuration among $C_1, \ldots, C_{n-1}$ is bi-valent. For if not, let j be the lowest number such that C_j is 1-valent. Obviously, C_{j-1} must then be 0-valent. Since we suppose our protocol to be 1-resilient, we can let p_j be dead from the beginning; there is still a finite run σ (thus not involving p_j) from C_j such that a decision is made. But b_j is the only input bit where C_{j-1} and C_j differ. Therefore, σ is a run also from 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. $\square$

The following is an easy technical lemma which will come handy.

Lemma 2. (Commutativity Lemma) *Let σ_1 and σ_2 be runs from C, and suppose that the set of processes taking steps in σ_1 is disjoint from the set of processes taking steps in σ_2. Then $\sigma_1\sigma_2$ and $\sigma_2\sigma_1$ are both runs from C, and they lead to the same configuration.*

Proof. Homework. $\square$

Let C be a configuration and $e = (p, m)$ an event applicable to C. Notice that if e is applicable to C it remains applicable until its corresponding step is executed. A run $\sigma = e_1 e_2 \ldots e_n$ is *e-free* if it does not contain e.

Lemma 3. *Let C be bivalent, and let e be any event applicable to C. Then, there is a (possibly empty) e-free run σ such that $e(\sigma(C))$ is bivalent.*

With this lemma we can prove the theorem. To see this, construct an admissible run by starting with an initial bivalent configuration C_0, whose existence is guaranteed by Lemma 1, as follows. Let $p_1, p_2, \ldots, p_n$ be any ordering of the processes. Pick any applicable event $e_1 = (p_1, m_1)$ and apply the lemma, thereby obtaining a second bivalent configuration $C_1 = e_1(\sigma_1(C_0))$. To make things more concrete, $e_1 = (p_1, m_1)$ can be selected such that m_1 was the first message ever sent to p_1 or $\perp$ if there is none. Then, apply the lemma again by picking an event $e_2 = (p_2, m_2)$ applicable to C_1 (notice $p_1 \neq p_2$), 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 buffer. By proceeding in a round robin fashion– making p_i perform step $e_{i+kn} (k \geq 0)$– we obtain an unacceptable run.

Proof. Assume there is no such σ. Then $e(C)$ must be uni-valent; assume without loss of generality that it is 0-valent. We want to show the existence of an e-free run σ_0 leading from C to a configuration D such that $e(D)$ is

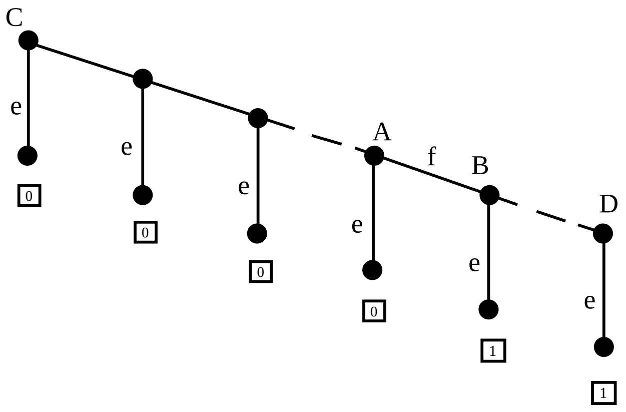

Figure 4.1: Finding A and B

1-valent. This means that on the way from C to D, there must be two neighboring configurations A and B such that $B = f(A)$, where f is some event where $e(A)$ is 0-valent and $e(B)$ is 1-valent. We illustrate the situation in Figure 4.1. To prove the existence of σ_0 notice first that since C is bivalent, there is a run σ_1 such that $E = \sigma_1(C)$ is 1-valent. Then, we distiguish between two cases:

- If σ_1 is e-free, let $\sigma_0 := \sigma_1$ and we are done because $D := \sigma_0(C) = E$ is 1-valent and, obviously, $e(D)$ is also 1-valent.
- Otherwise, let σ_0 be the largest e-free prefix of σ_1, and let $D := \sigma_0(C)$. By assumption, $e(D) = e(\sigma_0(C))$ cannot be bivalent. And since there is a (possibly empty) run from $e(D)$ to E, $e(D)$ cannot be 0-valent, so it must be 1-valent.

Focus now on A and $B = f(A)$ (refer always to Figure 4.1). We first show that the processes taking steps in e and f cannot be different. For if they were, according to Lemma 2, we would have $e(B) = e(f(A)) = f(e(A))$. But this is impossible, since $e(B)$ is 1-valent, and $e(A)$ is 0-valent. Thereafter, we show that we will still end up with a contradiction. Let p be the process taking steps in e and f. Since our protocol is supposed to be 1-resilient, there is a run ρ in which p does not take any steps, leading from A to some configuration R where a decision has been taken. See Figure 4.2.

However, the decision can neither be 0 nor 1:

- By Lemma 2, we have $\rho(e(B)) = \rho(e(f(A))) = e(f(\rho(A))) = e(f(R))$. Since $e(B)$ is 1-valent, R cannot be 0-valent.
- But according to Lemma 2, we also have $\rho(e(A)) = e(\rho(A)) = e(R)$. Since $e(A)$ is 0-valent, R cannot be 1-valent.

This final contradiction concludes the proof. □

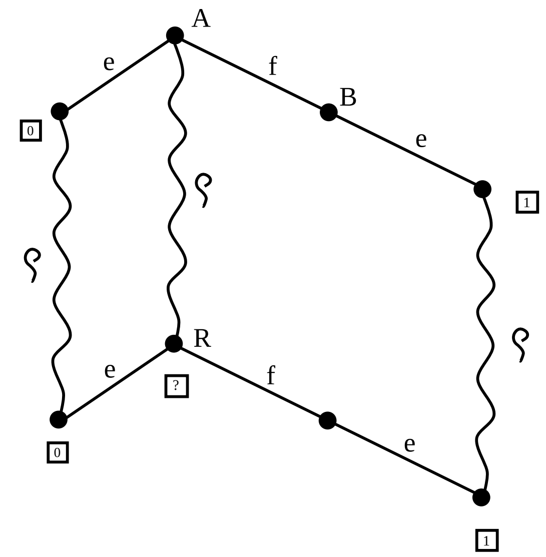

Figure 4.2: R cannot be univalent

Bibliography

[1] H. Attiya, Lecture Notes for Distributed Algorithms, Course # 236357, Dept of CS, The Technion, January 1994.

[2] M. Fischer, N. Lynch, and M. Paterson, Impossibility of distributed consensus with one faulty process, JACM, Vol. 32, No.2, April 1985, pp. 374–382

[3] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, Inc., San Francisco 1996