Lecture notes: Studying distributed systems – Consensus

M2 MOSIG: Large-Scale Data Management and Distributed Systems

Thomas Ropars

2024

This lecture studies consensus in distributed systems[1]. Consensus is one of the most fundamental problem in distributed systems.

For instance, consensus is required to implement *active* replication. Active replication is used to made a service fault tolerant by creating multiple replicas of that service. Active replication refers to replication techniques where all replicas are active and are able to process clients requests (as opposed to *passive* replication). To guarantee that a client receives the same answer no matter which replica answers its request, the only solution is that all replicas process all the requests from all clients in the same order: They need to reach consensus on the order in which requests should be processed.

As we will see soon, solving the consensus problem in a distributed system where some processes may crash is a difficult problem.

1 Definition of consensus

In the consensus problem we consider a set of n processes, each process p_i with an initial value v_i. These n processes have to agree on a common value v that is the initial value of one of the processes.

Formally, the consensus problem is defined by the primitives $propose(v_i)$ by which process p_i proposes its initial value, and $decide(v)$ by which a process decides (irrevocably) on a value. The decision must satisfy the following properties:

- *Termination:* Every correct process eventually decides.
- *Validity:* If a process decides v, then v is the initial value of some process (i.e., v was proposed by some process).
- *Uniform Agreement:* Two processes cannot decide differently.

Validity and uniform agreement are *safety* properties, while termination is a *liveness* property.

[1]Acknowledgments: Parts of these notes are strongly inspired by the lectures notes of Andre Schiper on *Distributed Algorithms*.

Remark: Uniform agreement has been defined above. The corresponding non-uniform property is defined as follows:

- *Agreement:* Two *correct* processes cannot decide differently.

As discussed when studying reliable broadcast, the *uniform* property is what we are most often looking for in practice. Hence, we consider *uniform agreement* in the following.

2 Impossibility results

In a distributed system, consensus can be solved only under certain conditions. We present below some important *impossibility results*.

2.1 Simple impossibility result

We start with a simple impossibility result. We note f, the number of processes that may crash in the system. We assume a crash-stop failure model for processes.

Theorem 1. *Consensus cannot be solved in an asynchronous system with reliable channels if a majority of processes may be faulty ($f \geq n/2$).*

The proof uses the "*indistinguishable run*" argument.

Indistinguishable run argument: Let R_0 be a run of some (deterministic) algorithm A, in which some process p has taken action a at time t. Let R_1 be another run of A, such that (i) p's initial state is the same in R_0 and R_1, (ii) until t, process p observes the same environment (e.g., sequence of messages received) in R_0 and R_1. It follows that p has also taken action a at time t in run R_1.

This is because runs R_0 and R_1 of A are indistinguishable for p until time t, and since A is deterministic, p does not behave differently in R_0 and R_1 until time t.

Proof. The proof of Theorem 1 is by contradiction. Suppose that algorithm A solves consensus in the above system model. Partition the processes into two sets Π_0 and Π_1 such that Π_0 contains $\lceil n/2 \rceil$ processes, and Π_1 contains the remaining $\lfloor n/2 \rfloor$ processes.

Consider run R_0 in which all processes propose 0. All processes in Π_0 are correct, while all processes in Π_1 crash at the beginning of the run. By the validity property, all correct processes decide 0 in R_0. Consider $q_0 \in \Pi_0$ and let q_0 decide at time t_0.

Consider run R_1 in which all processes propose 1. All processes in Π_1 are correct, while all processes in Π_0 crash at the beginning of the run. This is possible since $f \geq n/2$. By the validity property, all processes decide 1 in R_1. Consider $q_1 \in \Pi_1$ and let q_1 decide at time t_1.

We now construct a run R, in which no process crashes. In run R all processes in Π_0 propose 0 and all processes in Π_1 propose 1. In run R, the reception of all messages from Π_0 to Π_1 and from Π_1 to Π_0 is delayed until time $t = max(t_0, t_1)$; all other messages are received as in R_0 resp. R_1. Therefore, until time t, R is indistinguishable from R_0 for processes in Π_0. So, in run R, process q_0 decides 0 at time t_0. Until time t, R is indistinguishable from R_1 for processes in Π_1. So, in run R, process q_1 decides 1 at time t_1.

Therefore, in R some processes decide 0 and some other decide 1. This violates the agreement property of consensus, i.e., shows the contradiction. □

2.2 The FLP impossibility result

The above result puts restrictions on solving consensus in an asynchronous system. However, there is a much more fundamental impossibility result, called the FLP impossibility result, established by Fischer, Lynch and Paterson [6].

Theorem 2. *There exists no deterministic algorithm that solves consensus in an asynchronous system with reliable channels if one single process may crash.*

The proof is complex, and will not be presented.

It is important to understand correctly this result. *Solving* some problem P, means solving P in *"all"* runs compatible with the system model. It does not mean *not being able to solve* P *in "any" run.* For example, consider the following simple algorithm for consensus in a system with three processes p_1, p_2, p_3:

- Process p_1 sends its initial value v_1 to all other processes, and decides v_1:
- Processes p_2 and p_3 wait the initial value of p_1, and upon reception decide on the value received.

This algorithm solves consensus in runs in which p_1 does not crash. However, this algorithm does not solve consensus in *all* runs.

Intuition of the proof: The intuition behind the FLP impossibility result boils down to the fact that in an asynchronous system, a process p cannot know whether a non-responsive process q has crashed or if it just slow. If p waits for q, it might wait forever. If p does not wait and decides, it might find out later that q took a different decision.

3 Consensus in a synchronous system

Because of the FLP impossibility result, to be able to solve consensus, we need to make additional assumptions about the system. First, let us consider synchronous systems.

We start discussing the solution to consensus first in the synchronous system, and then in the synchronous round model, an abstraction that can be implemented on top of the synchronous system.

3.1 Consensus algorithm in a synchronous system

Flooding consensus To implement a consensus algorithm in a synchronous system, we assume a perfect failure detector (which is trivially implemented in a synchronous system). We also rely on quasi-reliable channels, and on the best-effort broadcast primitive introduced in the previous lecture.

The solution presented in Figure 1 is called *flooding consensus* [1] and implements non-uniform agreement. The basic idea of the algorithm is as follows. The algorithm executes in *rounds*. In each round, each process *floods* the system with all proposed values it already knows (use of best-effort broadcast – lines 22 and 46). If at the end of round, a process has collected all proposed values by other processes that can possibly be seen, it decides by selecting one value from the set using a deterministic function (lines 40-43).

The three main points to understand about this algorithm are:

```
1   Implements:
2   Consensus, instance c.

4   Uses:
5   BestEffortBroadcast, instance beb
6   PerfectFailureDetector, instance 𝒫

8   Variables:
9   correct = Π # The set of processes considered correct
10  round = 1
11  decision = ⊥
12  receivedfrom = [∅]^N
13  W = [∅]^N # The set of proposed values
14  receivedfrom[0] = Π

16  Upon event crash(q) raised by 𝒫:
17      correct = correct \ {q}
18      trydecide()

20  Upon c.propose(v):
21      W[1] = W[1] ∪ v
22      beb.broadcast([PROPOSAL, 1, W[1]])

24  Upon beb.deliver(p, [PROPOSAL, r, ps]):
25      receivedfrom[r] = receivedfrom[r] ∪ p
26      W[r] = W[r] ∪ ps
27      trydecide()

29  Upon beb.deliver(p, [DECIDED, v]):
30      if p ∈ correct and decision == ⊥:
31          decision = v
32          beb.broadcast([DECIDED, decision])
33          Trigger c.decide(decision)

35  Function roundFinishing():
36      return (correct ⊆ receivedfrom[round] and decision == ⊥)

38  Function tryDecide():
39      if roundFinishing():
40          if receivedfrom[round] == receivedfrom[round-1]:
41              decision = Min(W[round])
42              beb.broadcast([DECIDED, decision])
43              Trigger c.decide(decision)
44          else:
45              round = round + 1
46              beb.broadcast([PROPOSAL, round, W[round-1]])
```

Figure 1: Flooding consensus.

- A round finishes at process q when q has received a PROPOSAL message in that round from every process that q has not been detected to have crash
- It is safe to decide in round r for process p_i, if p_i is sure that it has seen all values that other

processes might have in their `proposals` set. To be sure of this, it is not enough to finish a round because one process p_j might crash in the middle of a round, and process p_i might have not have received the message from p_j while another process p_k might. On the other hand, if in two consecutive rounds $r-1$ and r, the set `receivedfrom` is the same for p_i, then all alive processes in round $r-1$ have sent their message to p_i in round r and by the properties of best-effort broadcast, and p_i is sure that it has seen all possible values: it can decide (line 40).

- This algorithm only implements *non-uniform* agreement because p_i might be the only process in round r to have received a message from all alive processes in round $r-1$ (another process p_l might have crashed in the middle of the round). Hence, p_i might crash immediately after deciding, and the remaining process might decide differently. Hence, we don't have uniform agreement.

Note that this algorithm might take up to N rounds to terminate if $N-1$ processes crash one by one, each in one round.

Uniform agreement with the flooding approach As we can see in Figure 1, expressing such an algorithm based in rounds using an *event-based* representation, as we did for the broadcast algorithm, can lead to a verbose and difficult-to-read code. Hence, we propose below another way of presenting the algorithm. In this section, we study an algorithm that provides *uniform* agreement.

To simplify the presentation of the algorithm, we assume that the failure detector is accessible through the predicate $crashed_p(q)$: the predicate is *true* if and only if p has detected the crash of q.

We use the alternative description for the algorithm presented in Figure 2 that provides uniform agreement. It still assumes quasi-reliable channels.

The parameter f represents the maximum number of processes that can crash. The algorithm assumes $f < n$ and consists of a loop that is executed $f+1$ times. Every process p maintains a set variable W_p that initially contains only p's initial value. In each execution of the loop every process p sends W_p to all other processes, and includes in W_p the values received. At line 52, p stops waiting for a message from q if $crashed_p(q)$ holds (otherwise p would block forever). At the end of loop $f+1$ every process p decides on the smallest value in W_p.

```
47    Variables:
48    W_p = {v_p} # set of proposed value, initially the singleton set with p's proposed value

50    for i_p in 1 to f + 1:
51        send [W_p, i_p] to all processes
52        wait until ∀q ∈ Π : ((received a message (W_q, i_p) from q) or (crashed_p(q)))
53        for all q from which the set W_q is received:
54            W_p = W_p ∪ W_q
55    DECIDE(Min(W_p))
```

Figure 2: Flooding consensus with Uniform agreement

It is easy to see that Algorithm 2 satisfies validity. Termination follows from the fact that no process is blocked forever at line 52. This follows from the quasi-reliable channels assumption. Consider q sending a message to p at line 51. If q does not crash, its message is eventually received by p. If q crashes, then $crashed_p(q)$ is eventually true.

Uniform agreement follows from the following lemma:

Lemma 3.1. *Let us denote by $W_p(i)$ the value of variable W_p at the end of loop i. If two processes p and q both reach the end of round $f+1$, then we have $W_p(f+1) = W_q(f+1)$.*

Proof. Since the loop is executed $f+1$ times and at most f processes can crash, there exist at least one execution of the loop in which no process crashes. At the end of this execution of the loop, all surviving processes have the same set W_p. This property trivially holds at the end of all subsequent execution of the loop. Therefore all surviving processes have the same set W_p upon execution of line 55. □

3.2 Consensus in a synchronous round model

The synchronous round model is a computational model: It defines the way to write algorithms. The synchronous round computational model has been introduced as a more convenient way to express consensus algorithms in a synchronous system model.

The synchronous round model hides some low level details, and allows therefore a more concise and simple algorithmic expression (see Figure 3). The synchronous round model be simply implemented in a synchronous system[2].

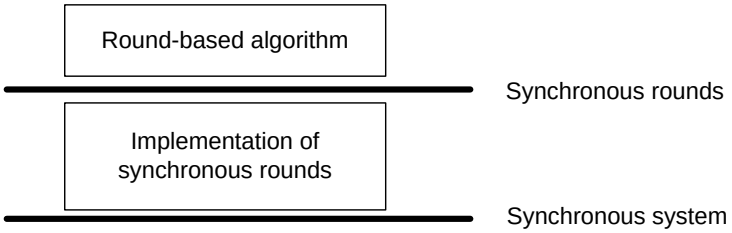

Figure 3: Synchronous round model vs. synchronous system

In the *synchronous round model* the computation is divided into rounds of message exchange. Each round r consists of a sending step, a receive step, and a state transition step:

1. In the sending step of round r, each process p sends a message to all processes.
2. In the receive step of round r, each process q receives all messages sent in round r by processes that are alive (i.e., not crashed) at the end of round r (if p crashes in round r, its message of round r might be received only by a subset of processes).
   The receive step is implicit, i.e., it does not appear in the algorithm.
3. In the state transition step, each process p executes a state transition function, with the set of messages received as parameter.

The synchronous round model allows us to slightly simplify the expression of Algorithm 2, see Algorithm 4. The loop and the round number r are now implicit. The algorithm is called *FloodSet* [8]. The sending step is denoted by S_p^r (sending step of p in round r), the receive step is implicit, and the state transition step is denoted by T_p^r.

[2]Describing how to implement synchronous round computational model in a synchronous system is outside the scope of these lecture notes.

```
56      Variables:
57      W_p = {v_p}

59      Round s:
60        S_p^r :
61            send (W_p) to all processes
62        T_p^r :
63            for all q from which the set W_q is received:
64                W_p = W_p ∪ W_q
65            if r == f+1:
66                DECIDE(Min(W_p))
```

Figure 4: *FloodSet* consensus algorithm ($f < n$)

4 Consensus in a partially synchronous system

The partially synchronous system model has been defined by Dwork, Lynch, and Stockmayer [5]. Roughly speaking a partially synchronous system is initially asynchronous and eventually becomes synchronous. We first give a precise definition, and then we define the *basic round model*, a computational model that can be implemented in a partially synchronous system. Finally we give two consensus algorithms in the basic round model. For simplicity we do not show here how to implement the basic round model in a partially synchronous model.

4.1 Partially synchronous system

In a synchronous system, processes and communication are synchronous. The partially synchronous system distinguishes partial synchrony for processes and partial synchrony for communication. Two versions of the definitions are given in [5]:

1. *Unknown bound*: There is a bound on the transmission delay of messages and a bound on the process relative speed, but the values of these bounds are unknown (bounds depend on the run).

2. *Known bound Δ and β hold eventually*: There exist values Δ and β with the following property: For every run R, there is a time T such that the transmission delay of messages is bounded by Δ and the process relative speed is bounded by β after T. Such a time T is called the *Global Stabilization Time* (GST).

   Channels can lose messages before GST, but are quasi-reliable after GST.

4.2 Basic round model: definition

As done for a synchronous system with the definition of the "synchronous round model", we can define a round model that can be implemented in the partially synchronous system, which allows us to simplify the expression of consensus algorithms. This round model is called *basic round model* [5].

Similarly to the synchronous round model, the computation is divided into rounds, where each round consists of a sending step, a receive step and a state transition step. The sending step and the state transition step are similar in the synchronous round model and in the basic round model.

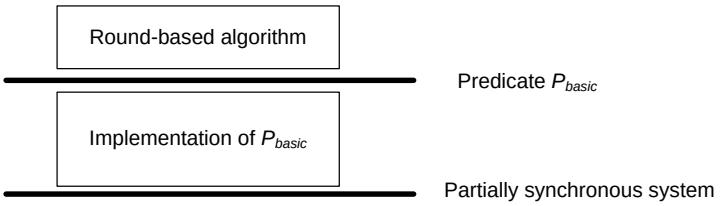

Figure 5: Partially synchronous system and basic round model.

The difference between the synchronous round model and the basic round model is in the receive step: in the basic round model, all messages from alive processes are guaranteed to be received only from round GSR (*Global Stabilization Round*), which is the first round after GST. In other words, channels can loose messages before round GSR, and are quasi-reliable for all rounds $r \geq GSR$. This can be expressed formally by the following predicate:

$$\mathcal{P}_{basic} :: \exists GSR > 0, \text{s.t.} \forall r \geq GSR, \forall p, q \in correct :$$
$$p \text{ sends } m \text{ to } q \text{ in round } r \Rightarrow q \text{ receives } m \text{ in round } r,$$

where *correct* denotes the set of correct processes.

The predicate $\mathcal{P}_{basic}$ is ensured by a lower layer algorithm (see Figure 5), which assumes an underlying partially synchronous system. The implementation of $\mathcal{P}_{basic}$ is not discussed in this course.

4.3 *OneThirdRule* (OTR) algorithm ($f < n/3$)

We start with a simple consensus algorithm expressed in the basic round model. With $f < n/3$, Algorithm 6 ensures validity and uniform agreement. Termination is ensured with predicate $\mathcal{P}_{basic}$. The algorithm is very simple, but includes several clever mechanisms. The proof below will help understanding these mechanisms.

```
67    Variables:
68    x_p = {v_p}

70    Round r:
71      S_p^r : # denotes the sending step of p in round r
72          send (x_p) to all processes

74      T_p^r : # denotes the state transition step of p in round r
75          if number of messages received ≥ n − f:
76              x_p = most frequent value received (if more than one, take the smallest)
77          if at least n − f values received are equal to x̄:
78              DECIDE(x̄)
```

Figure 6: The *OneThirdRule* (OTR) algorithm ($f < n/3$) [4]

Validity trivially holds.

Proof of uniform agreement: Let r_0 be the smallest round in which some process p decides v. So p has received in round r_0 at least $n-f$ messages v, i.e., at least $n-f$ processes have sent v.

Let q be another process that decides v', also in round r_0. So q has received in round r_0 at least $n-f$ messages v', i.e., at least $n-f$ processes have sent v'. However, $(n-f)+(n-f) > n$ if $f < n/3$. Therefore, one process must have sent v and v', i.e., $v = v'$. So p and q decide the same value.

We prove now that in all rounds $r \geq r_0$, if some process updates x_q (line 76), it updates x_q to v. We prove the result by contradiction.

Let $r' \geq r_0$ be the smallest round in which some process updates x_p to a value different from v. Since in round r_0, $n-f$ processes have $x_p = v$, and no process updates x_p to a value different from v before round r', in round r' we have at most f processes that send $v' \neq v$. Assume by contradiction that q updates x_q to v':

1. Process q has received at most f messages with value different from v;
2. Following 1, q has received at most f messages with v';
3. Considering 2, process q must have received at most f messages with v (otherwise q would not have updated x_q to v');
4. Process q has received at least $n-f$ messages (otherwise it would not have reached line 76).

So we have $f + f \geq n - f$ (f from item 1, f from item 3, $\geq n-f$ from item 4), i.e., $3f > n$. A contradiction with $f < n/3$.

In round r_0 we have at least $n-f$ processes with $x_p = v$, and we have shown that in all rounds $\geq r_0$, processes can update x_p only to v. So in all rounds $\geq r_0$ we have at least $n-f$ processes with $x_p = v$; this means that only v can be decided after round r_0. □

Proof of termination (sketch): The termination follows from the following observation. Consider $\mathcal{P}_{basic}$ and round $r_0 \geq GSR$ such that all faulty processes have crashed before round r_0. In round r_0, for all processes, the condition of line 75 is *true*. Moreover, for all p, q, in round r_0 the set of messages received by p and q is the same; so all processes set x_p to the same value, say $\overline{x}$. From here on, for all p we have forever $x_p = \overline{x}$.

Consider now round $r_0 + 1$. The condition of line 75 is again *true*, and since from round r_0 on all processes have $x_p = \overline{x}$, the condition of line 77 is true, and p decides at line 78. □

4.4 Multivalent vs. univalent configuration

The notion of multivalent and univalent configuration is a key notion in the context of consensus algorithms: it helps understanding consensus algorithms.

Configuration and reachable configuration We start by introducing the concepts of *configuration* and *reachable configuration*. Consider the execution of some algorithm A: each process and each channel goes through successive states. A configuration C is defined by the state of each process and of each channel.

For instance, consider algorithm A with three processes p_1, p_2, p_3 and six channels c_{12}, c_{21}, c_{13}, c_{31}, c_{23}, c_{32}. A configuration C is defined by the state of the three processes and by the state of the six channels.

A configuration C' is *reachable* from configuration C if, from C, the execution of A can bring the system in configuration C'.

Valence of a configuration If A is a consensus algorithm, an attribute called *valence* of configuration C — denoted by $val(C)$ — can be attached to C: $val(C)$ is the set of possible decision values in configurations that are reachable from C.

If $|val(C)| = 1$, we say that C is *univalent*; if $|val(C)| > 1$, we say that C is *multivalent*; if $val(C)$ is univalent and $val(C) = \{v\}$, we say that C is *v-valent*.

For example, for any consensus algorithm, if all initial values are equal to v, then the initial configuration C_0 is v-valent. If we consider OTR (Algorithm 6) and $f < n/3$, then any configuration, such that $n - f$ processes have x_p equal to some value v, is v-valent.

Valence and consensus algorithms The notion of univalence allows us to explain the principle of consensus algorithms. A consensus algorithm A usually work as follows:

1. A first tries to bring the system into a univalent configuration.
2. A process p decides v when it was able to observe that the system is in a v-valent configuration.

For example, in the case of OTR:

- A configuration in which $n - f$ correct processes have $x_p = v$ is v-valent.
- If in some round r all correct processes receive the same value v from at least $n - f$ processes, they all set x_p to v and the configuration becomes v-valent.
- If configuration becomes v-valent in round $r \geq GSR$, processes will decide in round $r + 1$.

4.5 Coordinator-based algorithm: Algorithm *"à la Paxos"* ($f < n/2$)

The OTR algorithm is symmetric: all processes execute the same code. This requires for safety $f < n/3$. It is possible to increase f from $f < n/3$ to $f < n/2$ by considering a non-symmetric algorithm. In a non-symmetric algorithm, one process, called the *coordinator*, executes a different code than the other processes.

The Paxos consensus algorithm, proposed by Lamport in 1989 and published in 1998 [7], is such an algorithm. It is probably the consensus algorithm that is most referenced in the literature and the most used in practice. The key features of Paxos are the following:

- *Paxos* requires $f < n/2$.
- *Paxos* is based on a *coordinator* process, which tries to impose a decision. In Paxos the coordinator process is chosen dynamically. To make it simple, we present here a variant, in which the coordinator is chosen by an off-line strategy, called *rotating coordinator* paradigm (explained below).
- *Paxos* always ensures validity and uniform agreement, but requires a condition for termination.

For simplicity, rather than describing the original Paxos algorithm, we give here a version of *Paxos* in the basic round model. The algorithm is called *LastVoting* [4], see Algorithm 7.

In *LastVoting*, contrary to OTR, not all rounds are identical. However,

- rounds 1, 4, 7, etc. are identical;
- rounds 2, 5, 8, etc. are identical;
- rounds 3, 6, 9, etc. are identical.

This leads us to group rounds into *phases*, where a phase consists of three rounds: phase ϕ ($\phi \geq 1$) consists of rounds $3\phi - 2$, $3\phi - 1$ and 3ϕ.

The coordinator changes from one phase to the next phase. More precisely, in phase ϕ, the coordinator is process $\big((\phi - 1)\ mod\ n\big) + 1$, i.e., p_1 is the coordinator for phase 1, p_2 the coordinator for phase 2, ..., p_n the coordinator for phase n, then p_1 again the coordinator for phase $n + 1$, etc. This schema is called *rotating coordinator*. The coordinator for phase ϕ is denoted $Coord(\phi)$.

The algorithm is always safe, i.e., uniform agreement and validity hold. Termination requires predicate $\mathcal{P}_{basic}$.

The best way to understand the *LastVoting* algorithm is to understand when a configuration of the algorithm is univalent.

v-valent configuration: The two variables of *LastVoting* to be considered when defining the state of process p are x_p and ts_p (time-stamp): x_p is initialized to the value proposed by p, and ts_p is the phase number in which p updated x_p most recently. We denote by x_p^C, respt. ts_p^C, the value of x_p, respt. ts_p, in configuration C.

Let Π denote the set of n processes that have to solve consensus, and let $f < n/2$. A configuration C of *LastVoting* is v-valent if

$$\exists Q \subseteq \Pi : |Q| \geq n - f \quad \wedge \quad \forall q \in Q\text{: } (x_q^C = v) \wedge \forall p \in \Pi \backslash Q : (ts_q^C > ts_p^C).$$

To understand why such a configuration is v-valent, consider for example $n = 3$, $f = 1$ and let C be a configuration that satisfies the above definition. Moreover assume the following:

- $Q = \{p_1, p_2\}$
- $x_1^C = x_2^C = v$
- $x_3^C = v' \neq v$, $ts_3^C < ts_1^C$, $ts_3^C < ts_2^C$

Any process that receives (x, ts) from $n - f = 2$ processes, and selects x from the message with the highest time-stamp ts, selects v. Therefore, once a configuration satisfies the above definition, every process updates x_p only to v. This allows a process to decide v in a configuration that satisfies the above definition. This also means that such a configuration is v-valent.

Algorithm: With the above definition of a v-valent configuration, the *LastVoting* algorithm is quite easy to understand:

- *Round* $3\phi - 2$: At line 85 process p sends x_p and ts_p to its coordinator, in order to allow the coordinator to identify a possible v-valent configuration. Identification of a possible v-valent configuration requires the reception of at least $n - f$ messages, see line 89. Lines 90 to 92 ensure that if the configuration is v-valent, then the coordinator sets $vote_p$ to v.[3]

[3]Note that round $3\phi - 2$ can be skipped in the first phase $\phi = 1$. In this case, $commit_p$ must be initialized to *true* and $vote_p$ to x_p.

```
79    Variables:
80    x_p = {v_p}
81    ts_p = 0 # most recent round in which x_p has been updated

83    Round r = 3φ − 2 :
84      S_p^r :
85          send (x_p, ts_p) to Coord(φ)

87      T_p^r :
88          commit_p = false
89          if p = Coord(φ) and number of (x, ts) received ≥ n − f:
90              t̂s = largest ts from (x, ts) received
91              vote_p = one x such that (x, t̂s) is received
92              commit_p = true

94    Round r = 3φ − 1 :
95      S_p^r :
96          if p = Coord(φ) and commit_p:
97              send (vote_p) to all processes

99      T_p^r :
100         if received (v) from Coord(φ):
101             x_p = v; ts_p = φ

103   Round r = 3φ :
104     S_p^r :
105         if ts_p == φ:
106             send (ack, x_p) to all processes

108     T_p^r :
109         if number of (ack, v) ≥ n − f:
110             DECIDE(v)
```

Figure 7: *LastVoting* algorithm ($f < n/2$) [4]

- *Round* $3\phi - 1$: If $vote_p$ has been assigned in round $3\phi - 2$, it is sent to all at line 97. The value of *vote* is adopted at line 101 where ts_p is updated.
- *Round* 3ϕ: A process that has adopted the vote, sends *ack* to all (line 106), in order to allow the identification of a v-valent configuration (condition at line 109), which leads to a decision at line 110.

Validity is obvious.

Proof of uniform agreement (sketch): Let ϕ_0 be the first phase at which some process decides. Let p be such a process and let v be its decision value. So $n - f$ processes have sent $\langle ack, v\rangle$ to p at line 109, and these $n-f$ processes have set x_p to v and ts_p to ϕ_0 at line 101 (*). The other processes q, at most $f < n/2$, trivially have $ts_q < \phi_0$ (**). We denote by Q^v the first set of processes, and by Q^{other} the second set.

Consider the smallest round $\phi_1 > \phi_0$ in which the condition of line 89 is true for some process c. If $f < n/2$, then $n - f > n/2$. Therefore c receives at least one message from Q^v. From (*) and

(**) it follows that c sets $vote_p$ to v at line 91. So in round ϕ_1 only v can be decided. By a simple induction, we can show that the same holds for every phase $> \phi_0$, which shows that agreement holds. □

Proof of termination (sketch): Consider $\mathcal{P}_{basic}$ and phase ϕ_0 such that $3\phi_0 - 2 \geq GSR$, all faulty processes have crashed before phase ϕ_0, and the coordinator c_0 of phase ϕ_0 is correct.

Since in phase ϕ_0 all messages sent are received, the condition at line 89 for some process c_0 is *true*, and c_0 sends a vote to all at line 97. Since in phase ϕ_0 all messages sent are received, the vote is received by all at line 100, x_p and ts_p are updated by all at line 101, and all send *ack* to all at line 106. Since in phase ϕ_0 all messages sent are received, the condition at line 109 is true for all non crashed processes, and all non crashed processes decide at line 110. □

4.6 Consensus in an asynchronous system augmented with failure detectors

We introduced failure detectors in a previous lecture as an alternative means to capture timing assumptions [3]. Failure detectors can also be used to solve consensus in an asynchronous system where a majority of processes is correct.

It has been shown that the weakest failure detector required to solve consensus in an asynchronous system where a majority of processes is correct, does not need to satisfie strong properties [2]. Indeed, this failure detector, called *eventually weak failure detector* and noted $\Diamond\mathcal{W}$, only satisfies the following properties:

- *Weak Completeness*: Eventually every process that crashes is permanently suspected by *some* correct process.
- *Eventual weak accuracy*: There is a time after which some correct process is never suspected by any correct process.

Describing consensus algorithms based on failure detectors is outside the scope of this course, as they are in general more complex to explain than algorithms based on the partially synchronous model. Furthermore, it is interesting to note that algorithms based on partially synchronous model are easier to extend to the *crash-recovery* model compared to algorithms that rely on a failure detector.

References

[1] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[2] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[4] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22:49–71, 2009.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[7] L. Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[8] N. A. Lynch. *Distributed algorithms*. Elsevier, 1996.