Lecture notes: Studying distributed systems – Consensus

M2 MOSIG: Large-Scale Data Management and Distributed Systems

Thomas Ropars

2023

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 [2].

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 two main points to understand about this algorithm are:

```
Implements:
Consensus, instance c.

Uses:
BestEffortBroadcast, instance beb
PerfectFailureDetector, instance $\mathcal{P}$

Variables:
correct = $\Pi$ # The set of processes considered correct
round = 1
decision = $\perp$
receivedfrom = [$\emptyset$]$^N$
W = [$\emptyset$]$^N$ # The set of proposed values
receivedfrom[0] = $\Pi$

Upon event crash(q) raised by $\mathcal{P}$:
    correct = correct \ {q}
    trydecide()

Upon c.propose(v):
    W[1] = W[1] $\cup$ v
    beb.broadcast([PROPOSAL, 1, W[1]])

Upon beb.deliver(p, [PROPOSAL, r, ps]):
    receivedfrom[r] = receivedfrom[r] $\cup$ p
    W[r] = W[r] $\cup$ ps
    trydecide()

Upon beb.deliver(p, [DECIDED, v]):
    if p $\in$ correct and decision == $\perp$:
        decision = v
        beb.broadcast([DECIDED, decision])
        Trigger c.decide(decision)

Function roundFinishing():
    return (correct $\subseteq$ receivedfrom[round] and decision == $\perp$)

Function tryDecide():
    if roundFinishing():
        if receivedfrom[round] == receivedfrom[round-1]:
            decision = Min(W[round])
            beb.broadcast([DECIDED, decision])
            Trigger c.decide(decision)
        else:
            round = round + 1
            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 might. On the other hand, if in two consecutive rounds $r-1$ and r, the set `receivedfrom` is the same, then no process has crashed in 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).

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

Another way of describing the algorithm 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.

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.

The alternative description of the algorithm is presented in Figure 2. 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 $\forall q \in \Pi$ : ((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              $W_p$ = $W_p \cup W_q$
55      DECIDE(Min($W_p$))
```

Figure 2: Another version of the Flooding consensus.

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.

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. □

Discussion It should be noted that there are differences in the properties of the algorithms presented in Figure 1 and 2. Algorithm 1 has the advantage that it terminates as soon as no process crash in one round. Expressing this idea makes the algorithm more complex. On the other hand, Algorithm 1 only considers the worst case to simplify the description: it always executes $f + 1$ rounds.

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 means to 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].

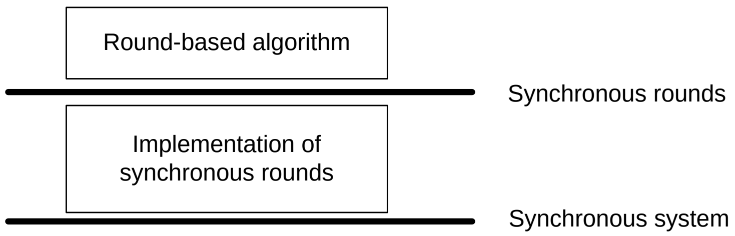

Figure 3: Synchronous round model vs. synchronous system

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* [3]. 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.

```
Variables:
$W_p$ = {$v_p$}

Round $s$:
  $S_p^r$ :
      send ($W_p$) to all processes
  $T_p^r$ :
      for all $q$ from which the set $W_q$ is received:
          $W_p$ = $W_p \cup W_q$
      if $r$ == f+1:
          DECIDE(Min($W_p$))
```

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

4 Consensus in a partially synchronous system

TO BE CONTINUED

References

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

[2] 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.

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