

## Chapter 22

# Consistency & Logical Time

You submit a comment on your favorite social media platform using your phone. The comment is immediately visible on the phone, but not on your laptop. Is this level of consistency acceptable?

### 22.1 Consistency Models

**Definition 22.1** (Object). An *object* is a variable or a data structure storing information.

**Remarks:**

- Object is a general term for any entity that can be modified, like a queue, stack, memory slot, file system, etc.

**Definition 22.2** (Operation). An *operation*  $f$  accesses or manipulates an object. The operation  $f$  starts at wall-clock time  $f_*$  and ends at wall-clock time  $f_{\dagger}$ .

**Remarks:**

- An operation can be as simple as extracting an element from a data structure, but an operation may also be more complex, like fetching an element, modifying it and storing it again.
- If  $f_{\dagger} < g_*$ , we simply write  $f < g$ .

**Definition 22.3** (Execution). An *execution*  $E$  is a set of operations on one or multiple objects that are executed by a set of nodes.

**Definition 22.4** (Sequential Execution). An execution restricted to a single node is a *sequential execution*. All operations are executed sequentially, which means that no two operations  $f$  and  $g$  are concurrent, i.e., we have  $f < g$  or  $g < f$ .

**Remarks:**

- Arguing about correctness of executions is generally difficult. Even in the much simpler case of sequential executions there are advanced tools to argue about correctness, such as model checking or formal methods. We consider a sequential execution to be correct if the operations manipulate the objects as expected, e.g. if we add  $2+2$  we want to see a result of 4.

**Definition 22.5** (Semantic Equivalence). *Two executions are **semantically equivalent** if they contain exactly the same operations. Moreover, each pair of corresponding operations has the same effect in both executions.*

**Remarks:**

- For example, when dealing with a stack object, corresponding `pop` operations in two different semantically equivalent executions must yield the same element of the stack.
- In general, the notion of semantic equivalence is non-trivial and dependent on the type of the object.

**Definition 22.6** (Linearizability). *An execution  $E$  is called **linearizable** (or **atomically consistent**), if there is a sequence of operations (sequential execution)  $S$  such that:*

- $S$  is correct and semantically equivalent to  $E$ .
- Whenever  $f < g$  for two operations  $f, g$  in  $E$ , then also  $f < g$  in  $S$ .

**Definition 22.7.** *A **linearization point** of operation  $f$  is some  $f_\bullet \in [f_*, f_\dagger]$ .*

**Lemma 22.8.** *An execution  $E$  is linearizable if and only if there exist linearization points such that the sequential execution  $S$  that results in ordering the operations according to those linearization points is correct and semantically equivalent to  $E$ .*

*Proof.* Let  $f$  and  $g$  be two operations in  $E$  with  $f_\dagger < g_*$ . Then by definition of linearization points we also have  $f_\bullet < g_\bullet$  and therefore  $f < g$  in  $S$ .  $\square$

**Definition 22.9** (Sequential Consistency). *An execution  $E$  is called **sequentially consistent**, if there is a sequence of operations  $S$  such that:*

- $S$  is correct and semantically equivalent to  $E$ .
- Whenever  $f < g$  for two operations  $f, g$  **on the same node** in  $E$ , then also  $f < g$  in  $S$ .

**Lemma 22.10.** *Every linearizable execution is also sequentially consistent, i.e., linearizability  $\implies$  sequential consistency.*

*Proof.* Since linearizability (order of operations *on any* nodes must be respected) is stricter than sequential consistency (only order of operations *on the same node* must be respected), the lemma follows immediately.  $\square$

**Definition 22.11** (Quiescent Consistency). *An execution  $E$  is called **quiescently consistent**, if there is a sequence of operations  $S$  such that:*

- $S$  is correct and semantically equivalent to  $E$ .
- Let  $t$  be some quiescent point, i.e., for all operations  $f$  we have  $f_{\dagger} < t$  or  $f_{*} > t$ . Then for every  $t$  and every pair of operations  $g, h$  with  $g_{\dagger} < t$  and  $h_{*} > t$  we also have  $g < h$  in  $S$ .

**Lemma 22.12.** *Every linearizable execution is also quiescently consistent, i.e., linearizability  $\implies$  quiescent consistency.*

*Proof.* Let  $E$  be the original execution and  $S$  be the semantically equivalent sequential execution. Let  $t$  be a quiescent point and consider two operations  $g, h$  with  $g_{\dagger} < t < h_{*}$ . Then we have  $g < h$  in  $S$ . This order is also guaranteed by linearizability since  $g_{\dagger} < t < h_{*}$  implies  $g < h$ .  $\square$

**Lemma 22.13.** *Sequentially consistent and quiescent consistency do not imply one another.*

*Proof.* There are executions that are sequentially consistent but not quiescently consistent. An object initially has value 2. We apply two operations to this object: *inc* (increment the object by 1) and *double* (multiply the object by 2). Assume that *inc*  $<$  *double*, but *inc* and *double* are executed on different nodes. Then a result of 5 (first *double*, then *inc*) is sequentially consistent but not quiescently consistent (which demands that the result is 6 because *inc* must happen first).

There are executions that are quiescently consistent but not sequentially consistent. An object initially has value 2. Assume to have three operations on two nodes  $u$  and  $v$ . Node  $u$  calls first *inc* then *double*, node  $v$  calls *inc* once with  $inc_{*}^v < inc_{\dagger}^u < double_{*}^u < inc_{\dagger}^v$ . Since there is no quiescent point, quiescent consistency is okay with a sequential execution that doubles first, resulting in  $((2 \cdot 2) + 1) + 1 = 6$ . The sequential execution demands that  $inc^u < double^u$ , hence the result should be strictly larger than 6 (either 7 or 8).  $\square$

**Definition 22.14.** *A system or an implementation is called **linearizable** if it ensures that every possible execution is linearizable. Analogous definitions exist for sequential and quiescent consistency.*

**Remarks:**

- In the introductory social media example, a linearizable implementation would have to make sure that the comment is immediately visible on any device, as the *read* operation starts after the *write* operation finishes. If the system is only sequentially consistent, the comment does not need to be immediately visible on every device.

**Definition 22.15** (restricted execution). *Let  $E$  be an execution involving operations on multiple objects. For some object  $o$  we let the **restricted execution**  $E|o$  be the execution  $E$  filtered to only contain operations involving object  $o$ .*

**Definition 22.16.** *A consistency model is called **composable** if the following holds: If for every object  $o$  the restricted execution  $E|o$  is consistent, then also  $E$  is consistent.*

**Remarks:**

- Composability enables to implement, verify and execute multiple concurrent objects independently.

**Lemma 22.17.** *Sequential consistency is not composable.*

*Proof.* We consider an execution  $E$  with two nodes  $u$  and  $v$ , which operate on two objects  $x$  and  $y$  initially set to 0. The operations are as follows:  $u_1$  reads  $x = 1$ ,  $u_2$  writes  $y := 1$ ,  $v_1$  reads  $y = 1$ ,  $v_2$  writes  $x := 1$  with  $u_1 < u_2$  on node  $u$  and  $v_1 < v_2$  on node  $v$ . It is clear that  $E|x$  as well as  $E|y$  are sequentially consistent as the write operations may be before the respective read operations. In contrast, execution  $E$  is *not* sequentially consistent: Neither  $u_1$  nor  $v_1$  can possibly be the initial operation in any correct semantically equivalent sequential execution  $S$ , as that would imply reading 1 when the variable is still 0.  $\square$

**Theorem 22.18.** *Linearizability is composable.*

*Proof.* Let  $E$  be an execution composed of multiple restricted executions  $E|x$ . For any object  $x$  there is a sequential execution  $S|x$  that is semantically consistent to  $E|x$  and in which the operations are ordered according to wall-clock-linearization points. Let  $S$  be the sequential execution ordered according to all linearization points of all executions  $E|x$ .  $S$  is semantically equivalent to  $E$  as  $S|x$  is semantically equivalent to  $E|x$  for all objects  $x$  and two object-disjoint executions cannot interfere. Furthermore, if  $f_{\dagger} < g_*$  in  $E$ , then also  $f_{\bullet} < g_{\bullet}$  in  $E$  and therefore also  $f < g$  in  $S$ .  $\square$

## 22.2 Logical Clocks

To capture dependencies between nodes in an implementation, we can use logical clocks. These are supposed to respect the so-called happened-before relation.

**Definition 22.19.** *Let  $S_u$  be a sequence of operations on some node  $u$  and define “ $\rightarrow$ ” to be the **happened-before relation** on  $E := S_1 \cup \dots \cup S_n$  that satisfies the following three conditions:*

1. *If a local operation  $f$  occurs before operation  $g$  on the same node ( $f < g$ ), then  $f \rightarrow g$ .*
2. *If  $f$  is a send operation of one node, and  $g$  is the corresponding receive operation of another node, then  $f \rightarrow g$ .*
3. *If  $f, g, h$  are operations such that  $f \rightarrow g$  and  $g \rightarrow h$  then also  $f \rightarrow h$ .*

**Remarks:**

- If for two distinct operations  $f, g$  neither  $f \rightarrow g$  nor  $g \rightarrow f$ , then we also say  $f$  and  $g$  are *independent* and write  $f \sim g$ . Sequential computations are characterized by  $\rightarrow$  being a total order, whereas the computation is entirely concurrent if no operations  $f, g$  with  $f \rightarrow g$  exist.

**Definition 22.20** (Happened-before consistency). *An execution  $E$  is called **happened-before consistent**, if there is a sequence of operations  $S$  such that:*

- $S$  is correct and semantically equivalent to  $E$ .
- Whenever  $f \rightarrow g$  for two operations  $f, g$  in  $E$ , then also  $f < g$  in  $S$ .

**Lemma 22.21.** *Happened-before consistency = sequential consistency.*

*Proof.* Both consistency models execute all operations of a single node in the sequential order. In addition, happened-before consistency also respects messages between nodes. However, messages are also ordered by sequential consistency because of semantic equivalence (a receive cannot be before the corresponding send). Finally, even though transitivity is defined more formally in happened-before consistency, also sequential consistency respects transitivity.

In addition, sequential consistency orders two operations  $o_u, o_v$  on two different nodes  $u, v$  if  $o_v$  can see a state change caused by  $o_u$ . Such a state change does not happen out of the blue, in practice some messages between  $u$  and  $v$  (maybe via “shared blackboard” or some other form of communication) will be involved to communicate the state change.  $\square$

**Definition 22.22** (Logical clock). *A logical clock is a family of functions  $c_u$  that map every operation  $f \in E$  on node  $u$  to some logical time  $c_u(f)$  such that the happened-before relation  $\rightarrow$  is respected, i.e., for two operations  $g$  on node  $u$  and  $h$  on node  $v$*

$$g \rightarrow h \implies c_u(g) < c_v(h).$$

**Definition 22.23.** *If it additionally holds that  $c_u(g) < c_v(h) \implies g \rightarrow h$ , then the clock is called a **strong logical clock**.*

**Remarks:**

- In algorithms we write  $c_u$  for the current logical time of node  $u$ .
- The simplest logical clock is the *Lamport clock*, given in Algorithm 22.24. Every message includes a timestamp, such that the receiving node may update its current logical time.

---

**Algorithm 22.24** Lamport clock (code for node  $u$ )

---

- 1: Initialize  $c_u := 0$ .
  - 2: Upon local operation: Increment current local time  $c_u := c_u + 1$ .
  - 3: Upon send operation: Increment  $c_u := c_u + 1$  and include  $c_u$  as  $T$  in the message.
  - 4: Upon receive operation: Extract  $T$  from message and update  $c_u := \max(c_u, T) + 1$ .
- 

**Theorem 22.25.** *Lamport clocks are logical clocks.*

*Proof.* If for two operations  $f, g$  it holds that  $f \rightarrow g$ , then according to the definition three cases are possible.

1. If  $f < g$  on the same node  $u$ , then  $c_u(f) < c_u(g)$ .
2. Let  $g$  be a receive operation on node  $v$  corresponding to some send operation  $f$  on another node  $u$ . We have  $c_v(g) \geq T + 1 = c_u(f) + 1 > c_u(f)$ .

3. Transitivity follows with  $f \rightarrow g$  and  $g \rightarrow h \Rightarrow f \rightarrow h$ , and the first two cases.

□

**Remarks:**

- Lamport logical clocks are not strong logical clocks, which means we cannot completely reconstruct  $\rightarrow$  from the family of clocks  $c_u$ .
- To achieve a strong logical clock, nodes also have to gather information about other clocks in the system, i.e., node  $u$  needs to have a idea of node  $v$ 's clock, for every  $u, v$ . This is what *vector clocks* in Algorithm 22.26 do: Each node  $u$  stores its knowledge about other node's logical clocks in an  $n$ -dimensional vector  $c_u$ .

**Algorithm 22.26** Vector clocks (code for node  $u$ )

- 
- 1: Initialize  $c_u[v] := 0$  for all other nodes  $v$ .
  - 2: Upon local operation: Increment current local time  $c_u[u] := c_u[u] + 1$ .
  - 3: Upon send operation: Increment  $c_u[u] := c_u[u] + 1$  and include the whole vector  $c_u$  as  $d$  in message.
  - 4: Upon receive operation: Extract vector  $d$  from message and update  $c_u[v] := \max(d[v], c_u[v])$  for all entries  $v$ . Increment  $c_u[u] := c_u[u] + 1$ .
- 

**Theorem 22.27.** *Define  $c_u < c_v$  if and only if  $c_u[w] \leq c_v[w]$  for all entries  $w$ , and  $c_u[x] < c_v[x]$  for at least one entry  $x$ . Then the vector clocks are strong logical clocks.*

*Proof.* We are given two operations  $f, g$ , with operation  $f$  on node  $u$ , and operation  $g$  on node  $v$ , possibly  $v = u$ .

If we have  $f \rightarrow g$ , then there must be a happened-before-path of operations and messages from  $f$  to  $g$ . According to Algorithm 22.26,  $c_v(g)$  must include at least the values of the vector  $c_u(f)$ , and the value  $c_v(g)[v] > c_u(f)[v]$ .

If we do not have  $f \rightarrow g$  nor  $g \rightarrow f$ , then  $<$  will not be defined for  $c_v(g)$  and  $c_u(f)$ :  $c_v(g)[u]$  cannot know about  $c_u(f)[u]$ , and hence  $c_v(g)[u] < c_u(f)[u]$ , since  $c_u(f)[u]$  was incremented when executing  $f$  on node  $u$ .

□

**Remarks:**

- Usually the number of interacting nodes is small compared to the overall number of nodes. Therefore, we do not need to send the full length clock vector, but only a vector containing the entries of the nodes that are actually communicating. This optimization is called the *differential technique*.

## 22.3 Consistent Snapshots

**Definition 22.28** (cut). A *cut* is some prefix of a distributed execution. More precisely, if a cut contains an operation  $f$  on some node  $u$ , then it also contains all the preceding operations of  $u$ . The set of last operations on every node included in the cut is called the **frontier** of the cut. A cut  $C$  is called **consistent** if for every operation  $g$  in  $C$  with  $f \rightarrow g$ ,  $C$  also contains  $f$ .

**Definition 22.29** (consistent snapshot). A **consistent snapshot** is a consistent cut  $C$  plus all messages in transit at the frontier of  $C$ .

**Remarks:**

- In a consistent snapshot it is forbidden to see an effect without its cause.
- Imagine a bank having lots of accounts with transactions all over the world. The bank wants to make sure that at no point in time money gets created or destroyed. This is where consistent snapshots come in: They are supposed to capture the state of the system. Theoretically, we have already used snapshots when we discussed configurations in Definition [16.17](#).

**Definition 22.30** (configuration). We say that a system is fully defined (at any point during the execution) by its **configuration**. The configuration includes the state of every node and all messages that are in transit (sent but not yet received).

**Remarks:**

- While a configuration describes the intractable state of a system at one point in time, a snapshot extracts all relevant tractable information of the systems state.
- One application of consistent snapshots is to check if certain invariants hold in a distributed setting. Other applications include distributed debugging or determining global states of a distributed system.
- In Algorithm [22.31](#) we assume that a node can record only its internal state and the messages it sends and receives. There is no common clock so it is not possible to just let each node record all information at precisely the same time.

**Theorem 22.32.** Algorithm [22.31](#) collects a consistent snapshot.

*Proof.* Let  $C$  be the cut induced by the frontier of all states and messages forwarded to the initiator. For every node  $u$ , let  $t_u$  be the time when  $u$  gets the first snap message  $m$  (either by the initiator, or as a message tag). Then  $C$  contains all of  $u$ 's operations before  $t_u$ , and none after  $t_u$  (also not the message  $m$  which arrives together with the tag at  $t_u$ ).

Assume for the sake of contradiction we have operations  $f, g$  on nodes  $u, v$  respectively, with  $f \rightarrow g$ ,  $f \notin C$  and  $g \in C$ , hence  $t_u \leq f$  and  $g < t_v$ . If  $u = v$  we have  $t_u \leq f < g < t_v = t_u$ , which is a contradiction. On the other

**Algorithm 22.31** Distributed Snapshot Algorithm

- 
- 1: Initiator: Save local state, send a snap message to all other nodes and collect incoming states and messages of all other nodes.
  - 2: All other nodes:
  - 3: Upon receiving a snap message for the first time: send own state (before message) to the initiator and propagate snap by adding snap tag to future messages.
  - 4: If afterwards receiving a message  $m$  without snap tag: Forward  $m$  to the initiator.
- 

hand, if  $u \neq v$ : Since  $t_u \leq f$  we know that all following send operations must have included the snap tag. Because of  $f \rightarrow g$  we know there is a path of messages between  $f$  and  $g$ , all including the snap tag. So the snap tag must have been received by node  $v$  before or with operation  $g$ , hence  $t_v \leq g$ , which is a contradiction to  $t_v > g$ .  $\square$

**Remarks:**

- It may of course happen that a node  $u$  sends a message  $m$  before receiving the first snap message at time  $t_u$  (hence not containing the snap tag), and this message  $m$  is only received by node  $v$  after  $t_v$ . Such a message  $m$  will be reported by  $v$ , and is as such included in the consistent snapshot (as a message that was *in transit* during the snapshot).
- The number of possible consistent snapshots gives also information about the degree of concurrency of the system.
- One extreme is a sequential computation, where stopping one node halts the whole system. Let  $q_u$  be the number of operations on node  $u \in \{1, \dots, n\}$ . Then the number of consistent snapshots (including the empty cut) in the sequential case is  $\mu_s := 1 + q_1 + q_2 + \dots + q_n$ .
- On the other hand, in an entirely concurrent computation the nodes are not dependent on one another and therefore stopping one node does not impact others. The number of consistent snapshots in this case is  $\mu_c := (1 + q_1) \cdot (1 + q_2) \cdot \dots \cdot (1 + q_n)$ .

**Definition 22.33** (measure of concurrency). *The concurrency measure of an execution  $E = (S_1, \dots, S_n)$  is defined as the ratio*

$$m(E) := \frac{\mu - \mu_s}{\mu_c - \mu_s},$$

where  $\mu$  denotes the number of consistent snapshot of  $E$ .

**Remarks:**

- This measure of concurrency is normalized to  $[0, 1]$ .
- In order to evaluate the extent to which a computation is concurrent, we need to compute the number of consistent snapshots  $\mu$ . This can be done via vector clocks.



## Chapter Notes

In his seminal work, Leslie Lamport came up with the happened-before relation and gave the first logical clock algorithm [Lam78]. This paper also laid the foundation for the theory of logical clocks. Fidge came up with vector clocks some time later [JF88]. An obvious drawback of vector clocks is the overhead caused by including the whole vector. Can we do better? In general, we cannot if we need strong logical clocks [CB91].

Lamport also introduced the algorithm for distributed snapshots, together with Chandy [CL85]. Besides this very basic algorithm, there exist several other algorithms, e.g., [LY87], [SK86].

Throughout the literature the definitions for, e.g., consistency or atomicity slightly differ. These concepts are studied in different communities, e.g., linearizability hails from the distributed systems community whereas the notion of serializability was first treated by the database community. As the two areas converged, the terminology got overloaded.

This chapter was written in collaboration with Julian Steger.

## Bibliography

- [CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.
- [CL85] K Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. 3:63–75, 02 1985.
- [JF88] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 10:56–66, 02 1988.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.
- [LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153 – 158, 1987.
- [SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *ICDCS*, pages 382–388. IEEE Computer Society, 1986.