Chapter 13

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?

13.1 Consistency Models

Definition 13.1 (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 13.2 (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 13.3 (Execution)**.** *An* ***execution*** *E is a set of operations on one or multiple objects that are executed by a set of nodes.*

Definition 13.4 (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 $g < f$.*

Definition 13.5 (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 13.6 (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 13.7. *A* ***linearization point*** *of operation f is some $f_\bullet \in [f_*, f_\dagger]$.*

Lemma 13.8. *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 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. □

Definition 13.9 (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 13.10. *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. □

Definition 13.11 (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 13.12. *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 is also guaranteed by linearizability since $g_\dagger < t < h_*$ implies $g < h$. □

Lemma 13.13. *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.

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 $inc^v_* < inc^u_\dagger < double^u_* < inc^v_\dagger$. 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). □

Definition 13.14. *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 13.15 (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 13.16. *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 13.17. *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. □

Theorem 13.18. *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. □

13.2 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 13.19. *Let S_u be a sequence of operations on some node u and define "$\rightarrow$" to be the* ***happened-before relation*** *on $E := S_1 \cup \cdots \cup S_n$ 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 13.20 (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 13.21. *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. □

Definition 13.22 (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 13.23. *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 13.24. Every message includes a timestamp, such that the receiving node may update its current logical time.

Algorithm 13.24 Lamport clock

```
(Code for node $u$)
Initialize $c_u := 0$.
Upon local operation: Increment current local time $c_u := c_u + 1$.
Upon send operation: Increment $c_u := c_u + 1$ and include $c_u$ as $T$ in message.
Upon receive operation: Extract $T$ from message and update $c_u := \max(c_u, T) + 1$.
```

Theorem 13.25. *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 $g \rightarrow h \Rightarrow g \rightarrow h$, 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 13.26 do: Each node u stores its knowledge about other node's logical clocks in an n-dimensional vector c_u.

Algorithm 13.26 Vector clocks

```
(Code for node $u$)
Initialize $c_u[v] := 0$ for all other nodes $v$.
Upon local operation: Increment current local time $c_u[u] := c_u[u] + 1$.
Upon send operation: Increment $c_u[u] := c_u[u] + 1$ and include the whole
    vector $c_u$ as $d$ in message.
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 13.27. *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 $f \to g$, then there must be a happened-before-path of operations and messages from f to g. According to Algorithm 13.26, $c_v(g)$ 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 $f \to g$, then $c_v(g)[u]$ 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 the called the *differential technique.*

13.3 Consistent Snapshots

Definition 13.28 (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.*

Definition 13.29 (consistent snapshot)**.** *A cut C is a **consistent snapshot**, if for every operation g in C with $f \to g$, C also contains f.*

Remarks:

- In a consistent snapshot it is forbidden to see an effect without its cause.

- 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 $\mu_s := 1 + q_1 + q_2 + \cdots + q_n$.

- One 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 $\mu_c := (1 + q_1) \cdot (1 + q_2) \cdots (1 + q_n)$.

Definition 13.30 (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 μ denotes the number of consistent snapshot of E.

Remarks:

- This measure of concurrency is normalized to $[0, 1]$. F

- In order to evaluate the extent to which a computation is concurrent, we need to compute the number of consistent snapshots μ. This can be done via vector clocks.

- 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 8.4:

Definition 13.31 (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:

- 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 13.32 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 13.33. *Algorithm 13.32 collects a consistent snapshot.*

Algorithm 13.32 Distributed Snapshot Algorithm

```
Initiator: Save local state, send a snap message to all other nodes and collect incoming states and messages of all other nodes.
All other nodes:
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.
If afterwards receiving a message $m$ without snap tag: Forward $m$ to the initiator.
```

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 $f \to g$, $f \notin C$ 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 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 $f \to g$ 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 $t_v > g$. □

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).

13.4 Distributed Tracing

Definition 13.34 (Microservice Architecture)**.** *A **microservice architecture** refers to a system composed of loosely coupled services. These services communicate by various protocols and are either decentrally coordinated (also known as "choreography") or centrally ("orchestration").*

Remarks:

- There is no exact definition for microservices. A rule of thumb is that you should be able to program a microservice from scratch within two weeks.

- Microservices are the architecture of choice to implement a cloud based distributed system, as they allow for different technology stacks, often also simplifying scalability issues.

- In contrast to a monolithic architecture, debugging and optimizing get trickier as it is difficult to detect which component exactly is causing problems.

- Due to the often heterogeneous technology, a uniform debugging framework is not feasible.

- Tracing enables tracking the set of services which participate in some task, and their interactions.

Definition 13.35 (Span)**.** *A **span** s is a named and timed operation representing a contiguous sequence of operations on one node. A span s has a start time s_* and finish time $s_\dagger$.*

Remarks:

- Spans represent tasks, like a client submitting a request or a server processing this request. Spans often trigger several child spans or forwards the work to another service.

Definition 13.36 (Span Reference)**.** *A span may causally depend on other spans. The two possible relations are **ChildOf** and **FollowsFrom** references. In a ChildOf reference, the parent span depends on the result of the child (the parents asks the child and the child answers), and therefore parent and child span must overlap. In FollowsFrom references parent spans do not depend in any way on the result of their child spans (the parent just invokes the child).*

Definition 13.37 (Trace)**.** *A **trace** is a series-parallel directed acyclic graph representing the hierarchy of spans that are executed to serve some request. Edges are annotated by the type of the reference, either ChildOf or FollowsFrom.*

Remarks:

- The advantage of using an open source definition like opentracing is that it is easy to replace a specific tracing by another one. This mitigates the lock-in effect that is often experienced when using some specific technology.

- Algorithm 13.38 shows what is needed if you want to trace requests to your system.

Algorithm 13.38 Inter-Service Tracing

```
Upon requesting another service: Inject information of current trace and
  span (IDs or timing information) into the request header.
Upon receiving request from another service: Extract trace and span infor-
  mation from the request header and create new span as child span.
```

Remarks:

- All tracing information is collected and has to be sent to some tracing backend which stores the traces and usually provides a frontend to understand what is going on.
- Opentracing implementations are available for the most commonly used programming frameworks and can therefore be used for heterogeneous collections of microservices.

13.5 Mutual Exclusion

When multiple nodes compete for exclusive access to a shared resource, we need a protocol which coordinates the order in which the resource gets assigned to the nodes. The most obvious algorithm is letting a leader node organize everything:

Algorithm 13.39 Centralized Mutual Exclusion Algorithm

```
To access shared resource: Send request message to leader and wait for
  permission.
To release shared resource: Send release message to leader.
```

Remarks:

- An advantage of Algorithm 13.39 is its simplicity and low message overhead with 3 messages per access.
- An obvious disadvantage is that the leader is single point of failure and performance bottleneck. Assuming an asynchronous system, this protocol also does not achieve first come first serve fairness.
- To eliminate the single bottleneck we pass an access token from node to node. This token contains the time t of the earliest known outstanding request.
- We assume a ring of nodes, i.e., there is an order of the nodes given such that every node knows its successor and predecessor.

Algorithm 13.40 Token-Based Mutual Exclusion Algorithm

```
To access shared resource at time $T_R$: Wait for token containing time $t$ of
  earliest known outstanding request.
Upon receiving token:
if $T_R = t$ then
  Hold token and access shared resource.
else if $T_R > t$ then
  Pass on token to next node.
else if $t = null$ or $T_R < t$ then
  Set $t = T_R$ and pass on token.
end if
To release access: Set $t = null$ and pass on token.
```

Remarks:

- Algorithm 13.40 achieves in-order fairness if all nodes stick to the rules.
- One issue is the breakdown if one node does not manage to pass on the token. In this case some new token has to be created and assigned to one of the remaining nodes.
- We can get rid of the token, if access to the token gets decided on a first come first serve basis with respect to logical clocks. This leads to Algorithm 13.41.

Algorithm 13.41 Distributed Mutual Exclusion Algorithm

```
To access shared resource: Send message to all nodes containing the node ID and the current timestamp.
Upon received request message: If access to the same resource is needed and the own timestamp is lower than timestamp in received message, **defer** response. Otherwise send back a response.
Upon responses **from all nodes** received: enter critical section. Afterwards send deferred responses.
```

Remarks:

- The algorithm guarantees mutual exclusion without deadlocks or starvation of a requesting process.
- The number of messages per entry is $2(n-1)$, where n is the number of nodes in the system: $(n-1)$ requests and $(n-1)$ responses.
- There is no single point of failure. Yet, whenever a node crashes, it will not reply with a response and the requesting node waits forever. Eeven worse, the requesting process cannot determine if the silence is due to the other process currently accessing the shared resource or crashing. Can we fix this? Indeed: Change step 2 in Algorithm 13.41 such that upon receiving request there will always be an answer, either Denied or OK. This way crashes will be detected.

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 some time later up with vector clocks [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.

Our definitions for distributed tracing follow the OpenTracing API [1]. The opentracing API only gives high-level definitions of how a tracing system is supposed to work. Only the implementation specifies how it works internally.There are several systems that implement these generic definitions, like Uber's open source tracer called *Jaeger*, or *Zipkin*, which was first developed by Twitter. This technology is relevant for the growing number of companies that embrace a microservice architecture. Netflix for example has a growing number of over 1,000 microservices.

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.

[1] http://opentracing.io/documentation/