Chapter 6

Logical clocks

Logical clocks assign a timestamp to all events in an asynchronous message-passing system that simulates real time, thereby allowing timing-based algorithms to run despite asynchrony. In general, they don't have anything to do with clock synchronization or wall-clock time; instead, they provide numerical values that increase over time and are consistent with the observable behavior of the system. This means that local events on a single process have increasing times, and messages are never delivered before they are sent, when time is measured using the logical clock.

6.1 Causal ordering

The underlying notion of a logical clock is **causal ordering**, a partial order on events that describes when one event e provably occurs before some other event e'.

For the purpose of defining casual ordering and logical clocks, we will assume that a schedule consists of **send events** and **receive events**, which correspond to some process sending a single message or receiving a single message, respectively.

Given two schedules S and S', call S and S' **similar** if $S|p = S'|p$ for all processes p; in other words, S and S' are similar if they are indistinguishable by all participants. We can define a causal ordering on the events of some schedule S implicitly by considering all schedules S' similar to S, and declare that $e < e'$ if e precedes e' in all such S. But it is usually more useful to make this ordering explicit.

Following [AW04, §6.1.1] (and ultimately [Lam78]), define the **happens-before** relation $\underset{S}{\Rightarrow}$ on a schedule S to consist of:

1. All pairs (e, e') where e precedes e' in S and e and e' are events of the same process.

2. All pairs (e, e') where e is a send event and e' is the receive event for the same message.

3. All pairs (e, e') where there exists a third event e'' such that $e \underset{S}{\Rightarrow} e''$ and $e'' \underset{S}{\Rightarrow} e'$. (In other words, we take the **transitive closure** of the relation defined by the previous two cases.)

It is not terribly hard to show that this gives a partial order; the main observation is that if $e \underset{S}{\Rightarrow} e'$, then e precedes e' in S. So $\underset{S}{\Rightarrow}$ is a subset of the total order $<_S$ given by the order of events in S.

A **causal shuffle** S' of a schedule S is a permutation of S that is consistent with the happens-before relation on S; that is, if e happens-before e' in S, then e precedes e' in S'. The importance of the happens-before relation follows from the following lemma, which says that the causal shuffles of S are precisely the schedules S' that are similar to S.

Lemma 6.1.1. *Let S' be a permutation of the events in S. Then the following two statements are equivalent:*

1. *S' is a causal shuffle of S.*

2. *S' is the schedule of an execution fragment of a message-passing system with $S|p = S'|p$ for all S'.*

Proof. $(1 \Rightarrow 2)$. We need to show both similarity and that S' corresponds to some execution fragment. We'll show similarity first. Pick some p; then every event at p in S also occurs in S', and they must occur in the same order by the first case of the definition of the happens-before relation. This gets us halfway to showing S' is the schedule of some execution fragment, since it says that any events initiated by p are consistent with p's programming. To get the rest of the way, observe that any other events are receive events. For each receive event e' in S, there must be some matching send event e also in S; thus e and e' are both in S' and occur in the right order by the second case of the definition of happens-before.

$(2 \Rightarrow 1)$. First observe that since every event e in S' occurs at some process p, if $S'|p = S|p$ for all p, then there is a one-to-one correspondence between events in S' and S, and thus S' is a permutation of S. Now we need to show that S' is consistent with $\underset{S}{\Rightarrow}$. Let $e \underset{S}{\Rightarrow} e'$. There are three cases.

1. e and e' are events of the same process p and $e <_S e'$. But then $e <_{S'} e'$ because $S|p = S'|p$.

2. e is a send event and e' is the corresponding receive event. Then $e <_{S'} e'$ because S' is the schedule of an execution fragment.

3. $e \underset{S}{\Rightarrow} e'$ by transitivity. Then each step in the chain connecting e to e' uses one of the previous cases, and $e <_{S'} e'$ by transitivity of $<_{S'}$.

□

There are two main applications for causal shuffles:

1. We can prove upper bounds by using a causal shuffle to turn some arbitrary S into a nice S', and argue that the niceness of S' means that S at least looks nice to the processes. An example of this can be found in Lemma 7.1.1.

2. We can prove lower bounds by using a causal shuffle to turn some specific S into a nasty S', and argue that the existence of S' tells us that there exist bad schedules for some particular problem. An example of this can be found in §7.4.2. This works particularly well because $\underset{S}{\Rightarrow}$ includes enough information to determine the latest possible time of any event in either S or S', so rearranging schedules like this doesn't change the worst-case time.

In both cases, we are using the fact that if I tell you $\underset{S}{\Rightarrow}$, then you know everything there is to know about the order of events in S that you can deduce from reports from each process together with the fact that messages don't travel back in time.

In the case that we want to use this information *inside* an algorithm, we run into the issue that $\underset{S}{\Rightarrow}$ is a pretty big relation ($\Theta(|S|^2)$ bits with a naive encoding), and seems to require global knowledge of $<_S$ to compute. So we can ask if there is some simpler, easily computable description that works almost as well. This is where logical clocks come in.

6.2 Implementations

The basic idea of a logical clock is to compute a **timestamp** for each event, so that comparing timestamps gives information about $\underset{S}{\Rightarrow}$. Note that these timestamps need not be totally ordered. In general, we will have a relation

$<_L$ between timestamps such that $e \underset{S}{\Rightarrow} e'$ implies $e <_L e'$, but it may be that there are some pairs of events that are ordered by the logical clock despite being incomparable in the happens-before relation.

Examples of logical clocks that use small timestamps but add extra ordering are Lamport clocks [Lam78], discussed in §6.2.1; and Neiger-Toueg-Welch clocks [NT87, Wel87], discussed in §6.2.2. These both assign integer timestamps to events and may order events that are not causally related. The main difference between them is that Lamport clocks do not alter the underlying execution, but may allow arbitrarily large jumps in the logical clock values; while Neiger-Toueg-Welch clocks guarantee small increments at the cost of possibly delaying parts of the system.[1]

More informative are **vector clocks** [Fid91, Mat93], discussed in §6.2.3. These use n-dimensional vectors of integers to capture $\underset{S}{\Rightarrow}$ exactly, at the cost of much higher overhead.

6.2.1 Lamport clock

Lamport's **logical clock** [Lam78] runs on top of any other message-passing protocol, adding additional state at each process and additional content to the messages (which is invisible to the underlying protocol). Every process maintains a local variable clock. When a process sends a message or executes an internal step, it sets $\mathsf{clock} \leftarrow \mathsf{clock} + 1$ and assigns the resulting value as the clock value of the event. If it sends a message, it piggybacks the resulting clock value on the message. When a process receives a message with timestamp t, it sets $\mathsf{clock} \leftarrow \max(\mathsf{clock}, t) + 1$; the resulting clock value is taken as the time of receipt of the message. (To make life easier, we assume messages are received one at a time.)

Theorem 6.2.1. *If we order all events by clock value, we get an execution of the underlying protocol that is locally indistinguishable from the original execution.*

Proof. Let $e <_L e'$ if e has a lower clock value than e'. If e and e' are two events of the same process, then $e <_L e'$. If e and e' are send and receive events of the same message, then again $e <_L e'$. So for *any* events e, e', if $e \underset{S}{\Rightarrow} e'$, then $e <_L e'$. Now apply Lemma 6.1.1. □

[1] This makes them similar to **synchronizers**, which we will discuss in Chapter 7.

6.2.2 Neiger-Toueg-Welch clock

Lamport's clock has the advantage of requiring no changes in the behavior of the underlying protocol, but has the disadvantage that clocks are entirely under the control of the logical-clock protocol and may as a result make huge jumps when a message is received. If this is unacceptable—perhaps the protocol needs to do some unskippable maintenance task every 1000 clock ticks—then an alternative approach due to Neiger and Toueg [NT87] and Welch [Wel87] can be used.

Method: Each process maintains its own variable clock, which it increments whenever it feels like it. To break ties, the process extends the clock value to $\langle \mathsf{clock}, \mathsf{id}, \mathsf{eventCount} \rangle$ where eventCount is a count of send and receive events (and possibly local computation steps). As in Lamport's clock, each message in the underlying protocol is timestamped with the current extended clock value. Because the protocol can't change the clock values on its own, when a message is received with a timestamp later than the current extended clock value, its delivery is delayed until clock exceeds the message timestamp, at which point the receive event is assigned the extended clock value of the time of delivery.

Theorem 6.2.2. *If we order all events by clock value, we get an execution of the underlying protocol that is locally indistinguishable from the original execution.*

Proof. Again, we have that (a) all events at the same process occur in increasing order (since the event count rises even if the clock value doesn't, and we assume that the clock value doesn't drop) and (b) all receive events occur later than the corresponding send event (since we force them to). So Lemma 6.1.1 applies. □

The advantage of the Neiger-Toueg-Welch clock is that it doesn't impose any assumptions on the clock values, so it is possible to make clock be a real-time clock at each process and nonetheless have a causally-consistent ordering of timestamps even if the local clocks are not perfectly synchronized. If some process's clock is too far off, it will have trouble getting its messages delivered quickly (if its clock is ahead) or receiving messages (if its clock is behind)—the net effect is to add a round-trip delay to that process equal to the difference between its clock and the clock of its correspondent. But the protocol works well when the processes' clocks are closely synchronized, which has become a plausible assumption in the last 10-15 years thanks to

the Network Time Protocol, cheap GPS receivers, and clock synchronization mechanisms built into most cellular phone networks.[2]

6.2.3 Vector clocks

Logical clocks give a *superset* of the happens-before relation: if $e \underset{S}{\Rightarrow} e'$, then $e <_L e'$ (or conversely, if $e \not<_L e'$, then it is not the case that $e \underset{S}{\Rightarrow} e'$). This is good enough for most applications, but what if we want to compute $\underset{S}{\Rightarrow}$ exactly?

Here we can use a **vector clock**, invented independently by Fidge [Fid91] and Mattern [Mat93]. Instead of a single clock value, each event is stamped with a vector of values, one for each process. When a process executes a local event or a send event, it increments only its own component x_p of the vector. When it receives a message, it increments x_p and sets each x_q to the max of its previous value and the value of x_q piggybacked on the message. We define $\mathsf{VC}(e) \leq \mathsf{VC}(e')$, where $\mathsf{VC}(e)$ is the value of the vector clock for e, if $\mathsf{VC}(e)_i \leq \mathsf{VC}(e')_i$ for all i.

Theorem 6.2.3. *Fix a schedule S; then for any e, e', $VC(e) < VC(e')$ if and only if $e \underset{S}{\Rightarrow} e'$.*

Proof. The if part follows immediately from the update rules for the vector clock. For the only if part, suppose e does not happen-before e'. Then e and e' are events of distinct processes p and p'. For $\mathsf{VC}(e) < \mathsf{VC}(e')$ to hold, we must have $\mathsf{VC}(e)_p < \mathsf{VC}(e')_p$; but this can occur only if the value of $\mathsf{VC}(e)_p$ is propagated to p' by some sequence of messages starting at p and ending at p' at or before e' occurs. In this case we have $e \underset{S}{\Rightarrow} e'$. □

6.3 Consistent snapshots

A **consistent snapshot** of a message-passing computation is a description of the states of the processes (and possibly messages in transit, but we can reduce this down to just states by keeping logs of messages sent and received) that gives the global configuration at some instant of a schedule that is a consistent reordering of the real schedule (a **consistent cut** in the terminology of [AW04, §6.1.2]. Without shutting down the protocol

[2] As I write this, my computer reports that its clock is an estimated 289 microseconds off from the timeserver it is synchronized to, which is less than a tenth of the round-trip delay to machines on the same local-area network and a tiny fraction of the round-trip delay to machines elsewhere, including the timeserver machine.

before taking a snapshot this is the about the best we can hope for in a message-passing system.

Logical clocks can be used to obtain consistent snapshots: pick some logical clock time and have each process record its state at this time (i.e., immediately after its last step before the time or immediately before its first step after the time). We have already argued that the logical clock gives a consistent reordering of the original schedule, so the set of values recorded is just the configuration at the end of an appropriate prefix of this reordering. In other words, it's a consistent snapshot.

If we aren't building logical clocks anyway, there is a simpler consistent snapshot algorithm due to Chandy and Lamport [CL85]. Here some central initiator broadcasts a snap message, and each process records its state and immediately forwards the snap message to all neighbors when it first receives a snap message. To show that the resulting configuration is a configuration of some consistent reordering, observe that (with FIFO channels) no process receives a message before receiving snap that was sent after the sender sent snap: thus causality is not violated by lining up all the pre-snap operations before all the post-snap ones.

The full Chandy-Lamport algorithm adds a second marker message that is used to sweep messages in transit out of the communications channels, which avoids the need to keep logs if we want to reconstruct what messages are in transit (this can also be done with the logical clock version). The idea is that when a process records its state after receiving the snap message, it issues a marker message on each outgoing channel. For incoming channels, the process all records all messages received between the snapshot and receiving a marker message on that channel (or nothing if it receives marker before receiving snap). A process only reports its value when it has received a marker on each channel. The marker and snap messages can also be combined if the broadcast algorithm for snap resends it on all channels anyway, and a further optimization is often to piggyback both on messages of the underlying protocol if the underlying protocol is chatty enough.

Note that Chandy-Lamport is equivalent to the logical-time snapshot using Lamport clocks, if the snap message is treated as a message with a very large timestamp. For Neiger-Toueg-Welch clocks, we get an algorithm where processes spontaneously decide to take snapshots (since Neiger-Toueg-Welch clocks aren't under the control of the snapshot algorithm) and delay post-snapshot messages until the local snapshot has been taken. This can be implemented as in Chandy-Lamport by separating pre-snapshot messages from post-snapshot messages with a marker message, and essentially turns into Chandy-Lamport if we insist that a process advance its clock to the

snapshot time when it receives a marker.

6.3.1 Property testing

Consistent snapshots are in principle useful for debugging (since one can gather a consistent state of the system without being able to talk to every process simultaneously), and in practice are mostly used for detecting **stable properties** of the system. Here a stable property is some predicate on global configurations that remains true in any successor to a configuration in which it is true, or (bending the notion of properties a bit) functions on configurations whose values don't change as the protocol runs. Typical examples are quiescence and its evil twin, deadlock. More exotic examples include total money supply in a banking system that cannot create or destroy money, or the fact that every process has cast an irrevocable vote in favor of some proposal or advanced its Neiger-Toueg-Welch-style clock past some threshold.

The reason we can test such properties using consistent snapshot is that when the snapshot terminates with value C in some configuration C', even though C may never have occurred during the actual execution of the protocol, there *is* an execution which leads from C to C'. So if P holds in C, stability means that it holds in C'.

Naturally, if P doesn't hold in C, we can't say much. So in this case we re-run the snapshot protocol and hope we win next time. If P eventually holds, we will eventually start the snapshot protocol after it holds and obtain a configuration (which again may not correspond to any global configuration that actually occurs) in which P holds.