Lecture 2: Safety, Liveness, Virtual Clocks (January 31)

Aurojit Panda

1 Plan for Today

1.1 Safety and Liveness

1.1.1 Traces

1.1.2 Partial traces

1.1.3 Program properties

1.2 Time and our problem with time

1.3 Lamport clocks

1.4 Vector clocks

1.5 [If time permits] Consistent snaphots

2 Safety and liveness

2.1 Some reminders from Lecture 1

- Distributed systems are comprised of many processes, each of which is isolated from others.
- Processes are connected by an asynchronous network which can arbitrarily delay or drop messages.
- We assume the network is fair: doesn't look at message content when deciding whether to delay or drop.
- We assume processes can fail.

Most of this semester: going to write protocols that processes can run to achieve some goals in this environment. What we want to make sure is that the algorithms we write are **correct**, but what does that even mean.

One way to decide correctness would be to say "the output is correct", e.g., a sort algorithm is correct if given a vector it produces a sorted version of the vector. But many distributed systems run forever, so what even is an "output"? Going to use a different strategy.

2.2 Traces/Schedules

Traces or schedules are a modeling tool for reasoning about systems in general that we will be using a lot throughout this semester.

A trace is a sequence of steps, where at each step exactly one process gets to do something.

2.2.1 What can a process do?

Depends on what you are modeling and how, but in the class we will normally this of a step being something where a process:

- Receives **one message**.
- Does some computation based on its local state and the message.
- Sends out one or more messages.

This should remind you a bit of how your code looks.

Some implications: A process can only take a step if some other process (or the system) had previously sent a message to the process.

If no process has a message waiting and no external entity ever sends a message then the system has halted.

> This model might explain some of why timers appear the way they do in the
> emulation environment.

2.2.2 Going back to a trace

We can now think of a trace as some function

$$f : \mathbb{Z}^+ \to (P, M)$$

where P is the set of processes and M is the set of messages.

Equivalently a trace is just an infinite array or list:

$$\begin{array}{cccc} 0 & 1 & 2 & \dots \\ p, m_0 & q, m_1 & p, m_2 & \dots \end{array}$$

where p and q are processes, and m_1, m_2 are messages.

The important thing to remember here is that this is an aid to modeling, not really saying that a single process is active at any time.

2.2.3 Prefixes and adding traces

Traces are potentially infinite but you often want to consider a portion of a trace, which we refer to as a prefix. For example the following is a prefix of the trace above:

$$\begin{array}{cc} 0 & 1 \\ p, m_0 & q, m_1 \end{array}$$

Sometimes it is helpful to combine traces, i.e., given a prefix add actions after. This is done by appending to the prefix, for example we can add a trace consisting of q, m_3, p, m_2 to the above trace to get another prefix

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ p, m_0 & q, m_1 & q, m_3 & p, m_2 \end{array}$$

Observe that this prefix is not a prefix of the original trace we began with.

2.2.4 Protocol correctness as trace properties

Given a trace we can now define a protocols correctness as a trace property. There are two types (and their combinations) that we will consider:

1. Safety This is a requirement that something never happen in a trace. For example we could require that for correctness process p should never receive message m_2 after process q has received m_3. In this case the prefix above shows that there is a problem. We do not need to consider the entire trace, reasoning about a prefix is sufficient in this case.

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ p, m_0 & q, m_1 & q, m_3 & p, m_2 \end{array}$$

2. Liveness This is the requirement that eventually something good happens. For example, we could require that eventually process p should receive message m_2, and all prefixes of the trace above allow for this.

(a) Uniform and Absolute Liveness Uniform liveness says there is some trace suffix (a part of trace) that can be added to any prefix to achieve the liveness property. Why might this be useful?

Absolute liveness says that combining a suffix with any prefix ensures liveness. This is much more powerful than either liveness or uniform liveness. Why might it be useful in practice?

3. General properties One of the more interesting results here is that any program property P can be split into a liveness property L and a safety property S such that $P = L \bigcup S$. You should try and work through the proof of this provided in Alpern and Schneider'85.

3 Time and its problems

Being able to order events in distributed systems is useful for a number of reasons:

- One, it is a way to produce a trace, which we can then use to reason about correctness.
- Second it is often a requirement for applications: example, an application that gives out 10,000 tickets and wants to ensure first-come first-serve.

However, ordering events in a distributed system is complicated: it is hard to get it so **all** processes agree on the order of events. Why? Fundamentally, this is because all information travels at finite speed. In vacuum this is c (the speed of light), in our networks it is dictated by message delays and latencies. This in turn means that two processes might receive information (messages) in different orders, which in turn complicates the problem.

3.1 Causality

One natural case where ordering should not matter is for events that are **causally** linked. We say two events e_0 and e_1 are causally linked iff: (a) e_0 results in the occurrence of e_1. (b) e_0 and e_1 are two events in the same process. (b) e_0 is causally linked to e_2, which is causally linked to e_3 and so on until e_n which is causally linked to e_1. This is the transitive closure of a and b.

Example: e_0 is process P sending a message m to process Q and e_1 is process Q receiving the message.

e_0 is process P sending a message m to process Q, e_2 is Q receiving the message and sending message m' to process R, and e_1 is R receiving message m'.

Why do we care about causality: as we develop clocks we want to make sure that observers do not mutually reorder causally linked events, it would be weird to reason about traces where a message is received before being sent.

4 Lamport Clocks

So now we get to the problem of how to get a trace from observing a distributed system's execution. We start with Lamport clocks.

4.1 Total order

Consider the set of events E. For any two events $e_1, e_2 \in E$, such that $e_1 \neq e_2$, we have either $t(e_1) < t(e_2)$ or $t(e_2) < t(e_1)$.

Think back to the trace: the trace also defines a total order.

4.2 On to Lamport clocks

We want to create a total order among events, while making sure that the order preserves causal order. Why total order? Because it is simple.

How?

- All processes send their current clock value in the message.
- When a process receives a message it sets it clock to the maximum of the processes current clock and the clock carried in the message.
- When sending a message the process increments clock value, and then sends a message with update clock value.

How to reconstruct a trace given this?

5 Vector clocks

One problem with Lamport clocks is that in providing a total order it imposes more constraints than naturally exist. For example, the total order picks an arbitrary order between events that are causally unrelated, and the ordering might significantly disagree from what an external observer might have seen.

Rather than building a total order, build a partial order to capture this structure.

5.1 Partial order

Consider the set of events. For any two events $e_1, e_2 \in E$ we can either have $e_1 < e_2$, $e_2 < e_1$ or e_1 can be incomparable to e_2.

5.2 Vector clocks

- Clocks are represented as a vector.
- Each component corresponds to a single process.
- Compare by element. [Explain how to compare.]
- [Explain update process]

6 Consistent snapshots [Only if time permits]

- What is a consistent snapshot? Want to get a consistent view of the state at all processes. Consistent here implies that we record (a) state at each process, and (b) ensure that the states are recorded in a causally consistent way, i.e., the snapshot should be such that if process p records its state after sending a message m to process q, then q should only record its state as it exists after it has received message m.
- Why useful? Gives us an instantaneous view of the distribtued system.
- How computed? See Chandy Lamport 1985.