TIME AND SYNCHRONIZATION

In previous lectures, we discussed some important concepts and goals of distributed systems. One important concept is remote procedure calls, where we saw how failures creep up into semantics and challenge communication. We now look at another important concept – time – and show how network delays creep up into semantics and challenge coordination.

I. Physical Clock Synchronization: Motivation and Challenges

Why is time an important concept in DS? It's required for synchronization/coordination. E.g.: we're all in class right now because we all agreed to be here at 1:10pm every Friday, and our notion of "time" is (roughly) the same across all of us. Similarly, in distributed systems, machines need to coordinate, and often to synchronize their actions with one another, and having a notion of time simplifies coordination/synchronization. As one example, a machine may want to write at a particular location within a file, print on a network printer, or access some other kind of resource. It needs to make sure that it's the only one doing that (mutual exclusion). As another example, machines often need to agree on a time to start executing something (barrier). In both of these examples, having a global notion of time would be very valuable, e.g., it could be used to order access to a shared resource or establish a rendez-vous point among the components of a distributed system.

As a third example, consider distributed debugging. Different machines log their events to their own local logs. When a failure occurs, a system admin needs to merge all logs together, creating one "complete log" that reflects all events. Ideally, the order of events in the complete log reflects the order of events in real life, so the administrator can debug the crash by looking at what happened before the crash. Having a unified notion of time across all machines in the distributed system would make it easy to create this coherent, complete log.

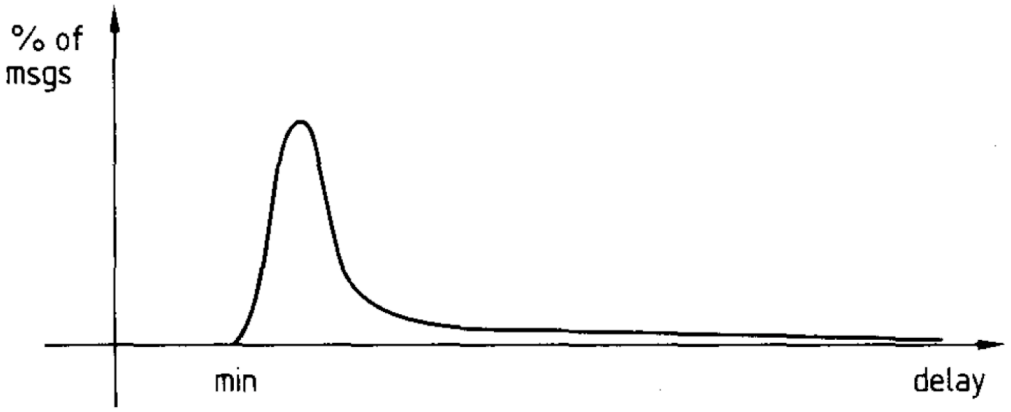
In general, many distributed systems problems can be addressed easily if one can assume a unified notion of time (a.k.a., global time). The trouble is, there isn't one.

Why is time synchronization hard? Two core reasons. First, machines have different physical clocks, which are never identical from a physical/chemical perspective (e.g., for quartz clocks, the crystals differ inside the clocks; surrounding electro-magnetic field, even surrounding temperature, affect oscillators). So they drift apart at varying pace. To address this problem, clocks are often *synchronized*, either amongst each other or with a reference time (e.g., GMT).

Second, synchronizing clocks involves network protocols, and that makes the problem hard. In realistic settings, we have to model the network as "*asynchronous*." This means that messages have:

- a lower bound "min" on propagation delay, dictated by the speed of light
    - if unknown, assume min = 0 (which hurts estimates the most)
- a "modus operandi," or a most likely propagation delay (peak of the delay distribution)
- BUT no hard upper bound on propagation delay

- some algorithms assume a known max, but this is problematic in practice. These algorithms are said to assume a "synchronous" network model. But this is not realistic, even in the best datacenter LANs (e.g., buggy router, queuing, attacks).



The best-known time synchronization protocol is NTP (Network Time Protocol), which is employed by all of our devices to synchronize their clocks with a set of reference clocks. How close do you think our clocks are within that reference? Over WAN, within tens of milliseconds. That's great synchronization for us (which is why all (most) of us can be on time for class at 1:10pm), but for machines, which operate at incredibly fast speeds, that's not sufficient synchronization. Within some tens of milliseconds, a regular computer can execute hundreds of millions of instructions! (That's a lot to order wrong in the distributed log example!)

II. Physical Clock Synchronization Protocols

Let's look at a few simple physical clock synchronization protocols to see where the bounds come from.

Context:

- master clock that is assumed to keep perfect time (RT)
  - keeps time t
- slave clocks Ci that we want to synchronize to master
  - each keeps local time Ci(t)
  - assume that Ci is "correct" if it drifts at a rate p
    - i.e., $(1-p)\Delta <= Ci(t+\Delta) - Ci(t) <= (1+p)\Delta$
- want two properties from clock synchronization
  - Clock consistency (internal): $| Ci(t) - Cj(t) | < d1$ for all i, j
  - Clock accuracy (external): $| Ci(t) - t | < d2$ for all i
  - If you have external synchronization, you get internal synchronization for free

Protocol 1: Simple broadcast-based time synchronization

Clock broadcasts time to all slaves

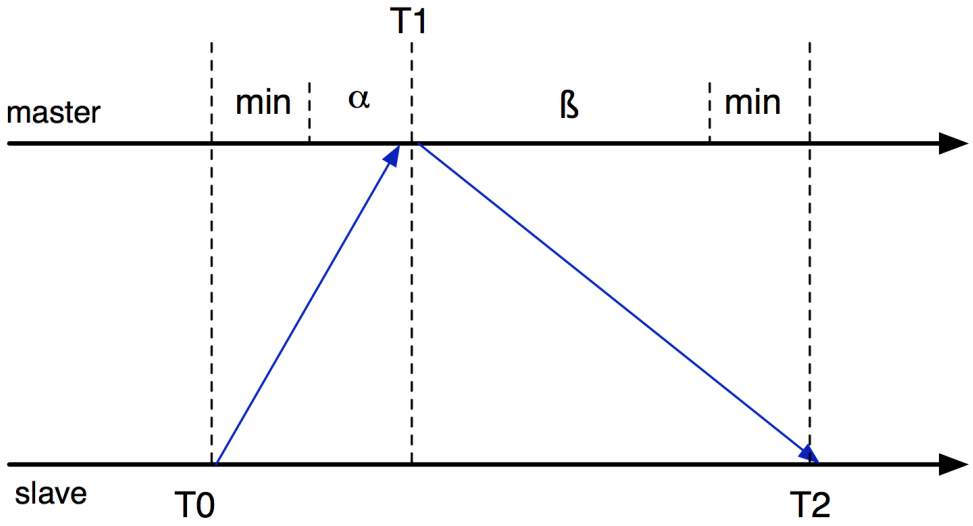
- broadcast message contains t
- slaves set clock to (t + min) when they receive broadcast

What is the accuracy of the clock?

- depends on where in the distribution the message delay is

- if assume "max" delay, then error could fall anywhere in the range (max – min)
- provable that this is the tightest error bound with probability 100%
    - therefore tightest consistency / accuracy

Protocol 2: Interrogation-based time synchronization



Goal:

- figure out what the master's clock says when the slave's clock says T2
    - it depends on alpha and beta, obviously
        - bounded by two cases: alpha = 0, and beta = 0
    - if alpha = 0, then beta = (T2-T0) – 2*min
        - Cmaster(T2) = T1 + min + beta
        - Cmaster(T2) = T1 + (T2-T0) – min
    - If beta = 0 , then:
        - Cmaster(T2) = T1 + min
- least possible error is to pick the midpoint
    - Cmaster(T2) = T1 + ((T2 – T0) / 2)
    - Max error = ((T2 – T0) / 2) – min

That was ignoring clock skew p. If you factor in clock skew, then the equations get a little more complicated. Least possible error is to pick turns out to be:

- Cmaster(T2) = T + ((T2 – T0)/2)(1 + 2p) – min p
- Max error = ((T2 – T0)/2)(1 + 2p) – min

Many implications to this:

- max error grows as clock skew climbs
- if you don't know "min", you have to set min = 0, and max error is basically proportional to the round-trip time (RTT)
- error diminishes as the measurement trial RTT approaches 2*min
    o is a probabilistic tradeoff

- can require measurements to be close to RTT to "accept" them and achieve rapport – increase number of trials necessary, but get tight error bounds
- can be sloppy and take any measurement – decreases number of trials, but get worse error bounds

Question: Does GPS change any of this fundamentally?

- can get a pretty tight bound on "min"
- alpha, beta are low
- get very good synchronization error bounds as a result
- but still not zero error, so perfect physical clock synchronization remains a challenge.

III. Logical Clocks

All of the above gives you a sense why it's hard to synchronize physical clocks perfectly. Fortunately, it turns out that coordination in distributed systems (e.g., for mutual exclusion, barriers, complete event log) **doesn't require** real time (as given by a physical clocks)! Leslie Lamport (Turing award winner and "father" of much of the theoretical foundation of distributed systems) was the one who realized that most coordination in distributed systems only needs a notion of **order of discrete events**. Going back to the distributed debugging example, in that case you only need order between **dependent** events that could possibly have **caused** the failure.

Lamport thus proposed the concept of **logical clocks**, plus a protocol for how to maintain them in a distributed system. Logical clocks (a.k.a., Lamport clocks), plus the protocol for how to synchronize them, are currently the underpinning of most distributed protocols and systems (we'll see examples later in the course).

Lamport defined two requirements for logical clocks:

- They must preserve *program order* (i.e., the order of events in one process needs to be preserved by the logical clock)
- They must preserve *message order* (i.e., a message sent event always needs to precede that message's receipt event in the logical clock)

He observed that these two orders are the minimum that capture the **"causal" relationship** between events in a distributed system. It turns out that capturing this relationship (which can be defined formally, but we won't do it here) is sufficient to address many coordination problems in distributed systems. As an example, reflect on why these two are critical requirements for the distributed debugging example.

The key idea behind logical clocks is the following: Each process in a distributed protocol maintains its own version of the logical clock. They each update their clocks as the protocol advances (i.e., as events happen). Whenever two processes communicate with one another, the process whose clock is behind (lower value) advances its clock to the other's clock. This is how processes "catch up" with the events that have happened in the distributed protocol beyond their limited local view. A detailed description of the logical clock synchronization protocol follows.

IV. Logical Clock Synchronization Protocol

Setup:

- Process = individual node in a distributed system
- Processes communicate by messages (e.g., RPCs)
- Events can be messages or system-specific events (e.g., write to file, read from file, whatever makes sense for the specific distributed system).
- View each process in the distributed system as a **state machine**: has some initial state, events cause it to move from one state to another.

Logical clock protocol:

- Each process P_i maintains a local counter, C_i
- Each process P_i increments C_i between any two successive events
- Each process piggybacks timestamp T_m on a message it sends out, where T_m is C_i at the time of sending m.
- On receiving m at process P_j:
    - P_j sets its counter C_j to max(C_j, T_m+1)
    - The receipt of m is a separate event that then separately advances C_j (i.e., C_j++)
- The box to the side describes the specific state machine at process P_i.

Node P_i's state machine:
On local event:
- C_i++

On message send:
- Piggyback C_i to msg.
- C_i++

On message receive:
- C_i = max(C_i, T_m+1)
- C_i++

Global ordering:

- The preceding protocol gives only partial ordering of events, where only events that may have a causal relationship are ordered, but there can be ties. But it's easy to get a total ordering of events. E.g.:
    - Use logical clock to set order
    - If tie, use process IDs as tie breaker
    - I.e., global order is (Logical timestamp).(process ID)

Example 1: Follow this protocol with the debugging example. Take a concrete set of events that each machine might have logged and follow the chain of events leading to the cause of the failure; see how logical clocks give us all the ordering that's needed to do the debugging.

Example 2: Distributed mutual exclusion with priority queues. Illustrates the "magic" of Lamport clocks. Uses slides (https://columbia.github.io/ds1-class/lectures/03-clocks-mutex-example-ppt.pdf, slides 2 and 13-23).

Pluses and minuses of Lamport clocks:

+ Respect causality, which can address many coordination problems in distributed systems.
- Capturing causality is sometimes insufficient, as there can be events outside the system that have causal influence on the evolution of the system. The ordering doesn't capture these relationships.
- Causal ordering doesn't actually imply influence, just potential influence. Hence, causal order can be too much order, affecting performance/scalability.

Key Papers

For more details on NTP and logical clocks, please consult the original papers that introduced them:

- [Mills-1991] David Mills. *Internet Time Synchronization: the Network Time Protocol*. In *IEEE Transactions on Communications, 1991.*
- [Lamport-1978] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. In *Communications of the ACM*, 1978.

Acknowledgements

The preceding notes contain portions adapted from the University of Washington distributed systems course, Steve Gribble's edition:
https://courses.cs.washington.edu/courses/csep552/13sp/lectures/2/physical_clocksync.pdf.