14-736: DISTRIBUTED SYSTEMS

LECTURE 5 * SPRING 2018 * KESDEN

PHYSICAL TIME

- Real-world monolith systems synchronize on clock edges
- Other than some ticklish business at really high rates within the processors, themselves, this works great
- In distributed systems, where the network is our shared communication channel, this doesn't work
  - Too many paths that are too long and of too many different lengths.
- Even once synchronized, timekeeping in commodity systems aren't atomic clocks. The clocks drift – some are fast, some are slow.
- Except in rare cases, physical time isn't good for synchronization

PHYSICAL TIME SYNCHRONIZATION

- Synchronizing physical time can be of use for human reasons
  - File time stamps
  - Login records
  - Session time-outs
  - Etc
- At "Google Scale", they can actually get a bit more mileage out of it

SYNCHRONIZING PHYSICAL TIME

- Generally there need to be one or more authoritative sources of the correct time.
- Then, this time needs to be distributed to clients (typical per client, upon request).
- But, this is tricky – it is off by an unknown amount of time upon receipt
  - It took time to get there over the network
  - And, a different amount of time for each client
  - The best we can do is estimate this

ESTIMATING COMMUNICATION LATENCY

- Without accurate time, we can't compare send and receive timestamps (or we wouldn't need to send the time)
- At best, we can measure the time relatively accurately over short interval from a single perspective
  - Measure round-trip-time
  - Assume it is symmetric (ouch)
  - Neglect processing time (tiny)
  - Ex, If the round trip time from request-to-reply was 10mS, assume the one way time was 5ms and add this to the time in the reply to estimate the current time.

IMPROVING PHYSICAL TIME ESTIMATION

- Assume one-way latency is half of RTT
- Reject outliers
- Keep a rolling weighted average, e.g.
  - $avg\_clock\_error_0 = local\_clock\_error$
  - $avg\_clock\_error_n = (weight * local\_clock\_error) + (1 - weight) * local\_clock\_error_{n-1}$
- Big caution!
  - Don't ever move backwards in time – it really messes up logging.
  - Instead, slow down future movements. "Miss ticks", etc.
  - (It is okay to jump ahead)
- Essentially this is known as "Cristian's Algorithm" and is the basis for NTP servers, etc.

HOW OFTEN TO REQUEST AN UPDATE?

- What is the spec for the maximum drift rate?
- How much drift is acceptable?
- At the maximum drift rate, how long does it take to reach the maximum tolerance?
  - Let this be the guide.

WHAT IF THERE ARE NO TIME SERVERS?

- Averaging clocks in community may produce a better estimate of the real time
- If nothing else, it produces a more robust estimate of the real time than any one clock
- We still don't want to ever set a clock backwards (just slow it down)
- Drift rate is now effectively double – some clocks can drift fast and others drift slow
  - Synchronize twice as often

BERKELEY ALGORITHM

- Server requests time from clients
- As responses come in, it adjusts as before via RTT
- Averages understandings of time
- Sends each client information about how much to speed up or slow down.
- As before, repeat as needed by the maximum drift rate of the clocks and the maximum tolerance for the time
  - But, keep in mind that, across the community, some clocks can be drifting faster and some slower. So, they drift apart from each other by the sum – up to twice as fast as a single clock from the ture time.

GOOGLE TRUETIME

- https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf
- Highly accurate physical time used at Google.
  - In some ways made possible by their scale.

GOOGLE TRUETIME: MASTERS

- GPS-references
- Atomic-clock referenced (paper says not much more expensive than GPS-referenced)
- Masters check time against each other and local clock
  - Evict themselves if drifting too fast.
- Between synchronizations, advertise slowly increasing uncertainty

GOOGLE TRUETIME CLIENTS

- Synchronize against multiple servers
- Reject outliers
- Evict self if they are repeated out of bounds upon synchronizing

GOOGLE TRUE TIME: HOW TIGHT?

- Uncertainty is1 to 7 ms over each poll interval, 4 ms most of the time.
- The daemon's poll interval was 30 seconds at publication.
- The drift rate was set at 200 microseconds/second at publication
- Latency can cause localized spikes in uncertainty, etc.

GOOGLE TRUETIME API

Method	Returns
$TT.now()$	*TTinterval*: $[earliest, latest]$
$TT.after(t)$	true if t has definitely passed
$TT.before(t)$	true if t has definitely not arrived

Table 1: TrueTime API. The argument t is of type *TTstamp*.

LOGICAL TIME

- If we can't synchronize physical time tightly enough to order operations, synchronize things, etc, maybe we can solve specific problems differently
- Logical time stamps are basically counters that advance in well-understood ways
  - Can be scalars, vectors, matricies, etc
  - Comparisons may provide total orderings or allow some concurrency (partial orderings)
    - Sometimes partial orderings ae made into total orderings by arbitrarily bra
  - Concurrency is may be true concurrency, or simply not being able to determine the order.
  - If counters upon certain events enables certain orderings

SEQUENCE NUMBERS

- Simplest logical time
- Order events on a single host, within a single session, etc.
- Think about TCP sequence numbers, etc.
- Meaningful within their scope, but can't be compared across scopes.

LAMPORT LOGICAL TIME CRITICAL DEFINITION

- def'n: *happens-before*
  - When comparing events on the same host, if event a occurs before event b then a happens-before b.
  - If one host receives a message sent by another host, the send happens-before the receive
  - If x occurs on P_1 and y occurs on P_2 and P_1 and P_2 have not exchanged messages then X and Y are said to be *concurrent*. If this is the case, we can't infer anything about the order of event x and event y. Please note, that this is only true if x and y don't exchange messages at all, even indirectly via third (or several) parties.
  - The relationship is transitive: if a happens before b, and b happens before c, then a happens before c.

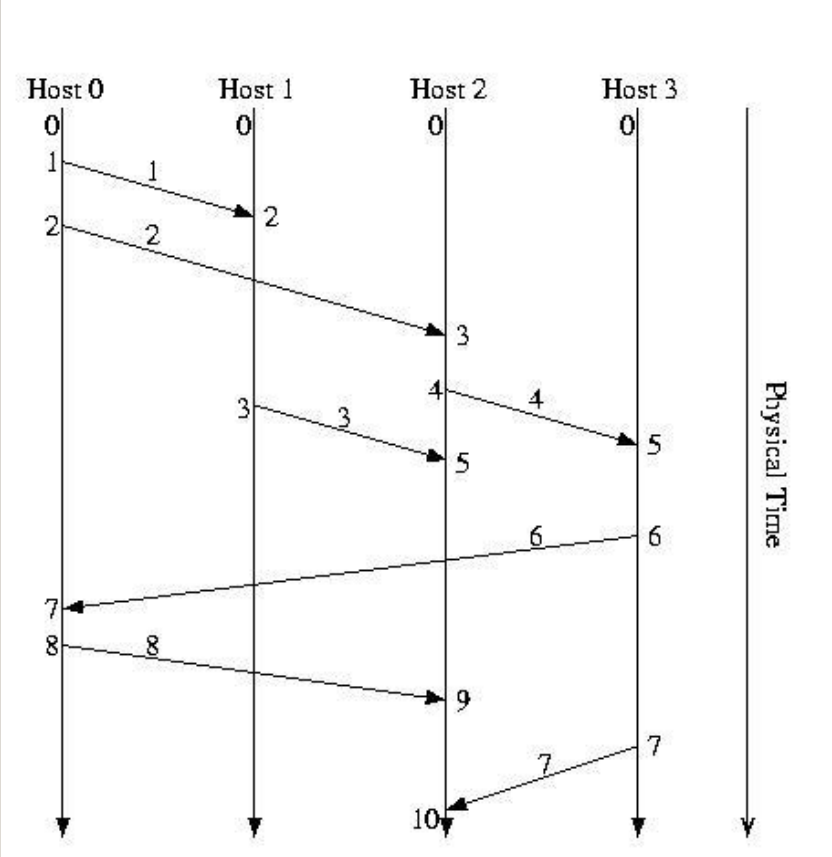
UPDATING LAMPORT TIME

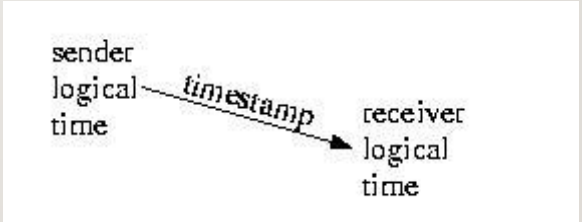
- The counter is incremented before each event.
- In the case of a send, the counter is incremented, and then the message is sent. The message should carry the new (incremented) timestamp.
- In the case of a receive, the proper action depends on the value of the timestamp in the message.
  - If the message has a higher timestamp than the receiver, the receiver's logical clock adopts the value sent with the message. (If not, skip this step and see next bullet)
  - In either case, the receiver's logical clock is incremented and the message is said to have been received at the new (incremented) clock value. This ensures that the messages is received after it was sent and after prior events on the receiving host
- Total ordering can be achieve, for example, by breaking ties with hostID.
  - View timestamp as decimal number: lamportTime.HostID

UNDERSTANDING LAMPORT TIME STAMPS

- If *a* occurs (real-world) before *b* on the same host
  - Lamport_Timestamp(a) < Lamport_Timestamp(b), i.e. "a happened before b"
- Lamport_Timestamp($send_x$ < Lamport_Timestamp($recv_x$)
  - Messages are necessarily received after they are sent
- Lamport_Timestamp(a) != Lamport_Timestamp(b)
  - Can be concurrent, unless a total ordering is imposed.
  - Necessarily the same event, if a total ordering is imposed.

LAMPORT TIME EXAMPLE

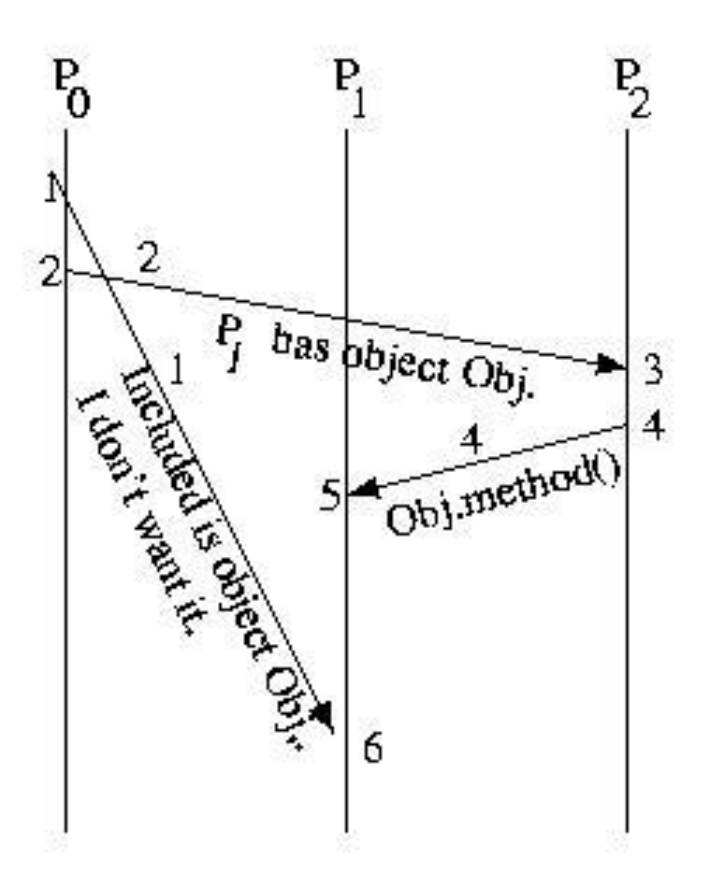




WHAT GOOD IS LAMPORT TIME?

- Causal ordering
  - If some eventA could have influenced some eventB, then eventA has a lower timestamp than B
  - Why? A "Could have influenced" eventB if there was communication, direct and indirect, between the two events where A could have told B something relevant to eventB
    - Consider the whisper game. A tells B to tell C to tell D to tell E to turn off the lights.
      - Since sends must have a higher stamp than receives, the time stamp must go up.
- Additionally, it is the philosophical basis for other types of more powerful timestamps.

WHAT HAPPENS HERE?

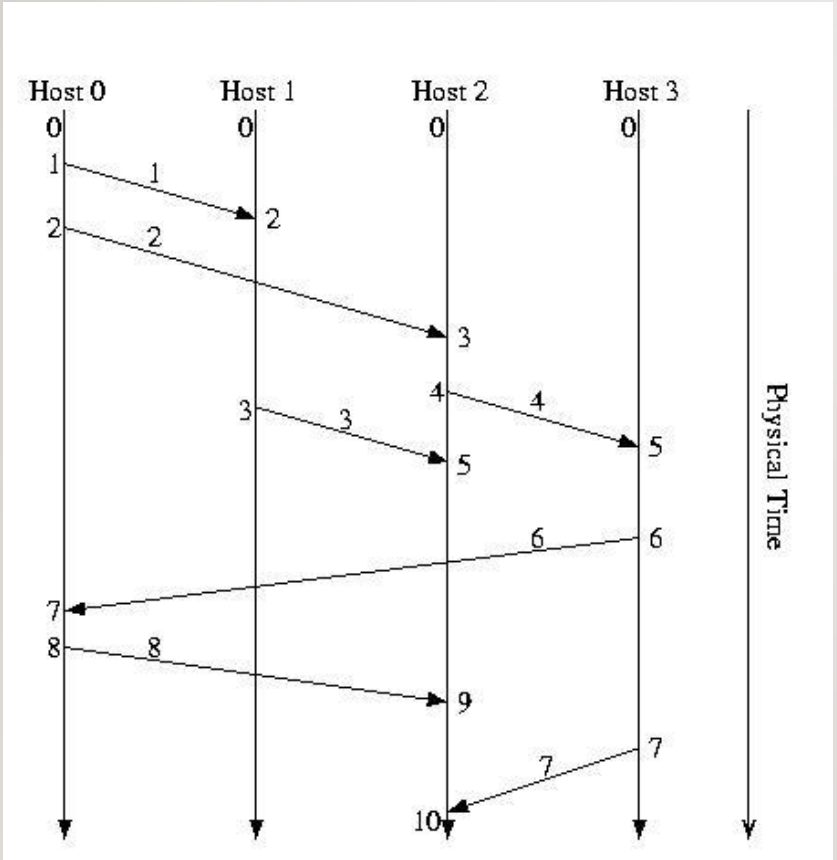


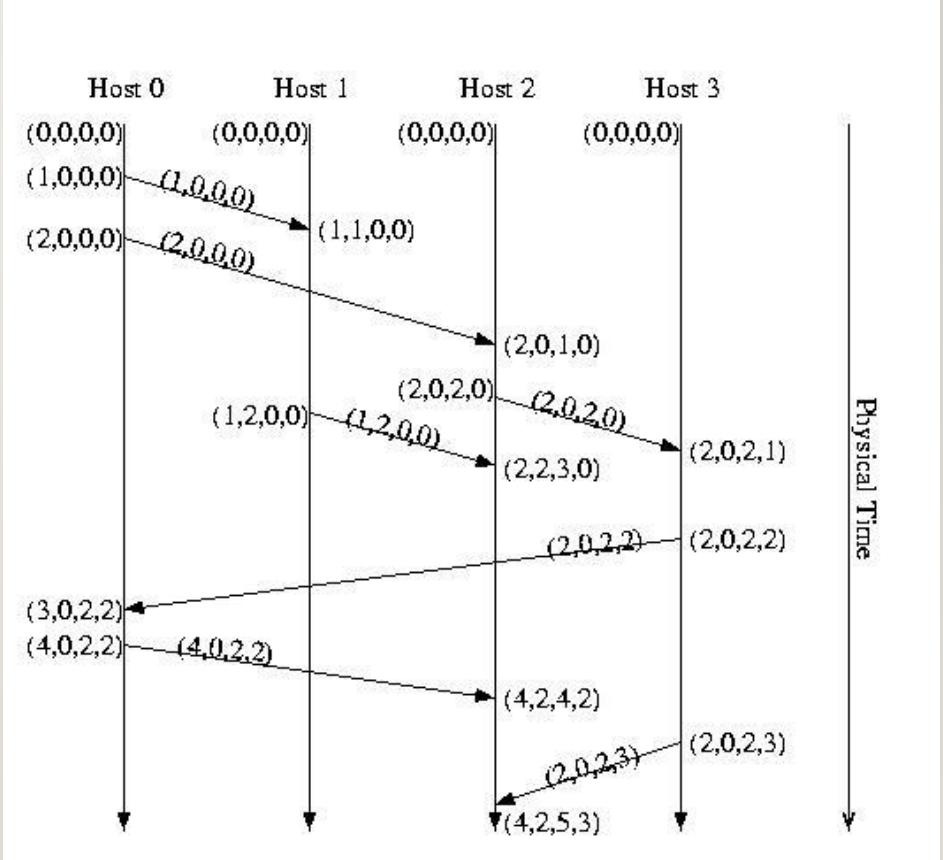
- When hosts don't talk, Lamport can't capture the causality

A MORE POWERFUL VECTOR TIME STAMP

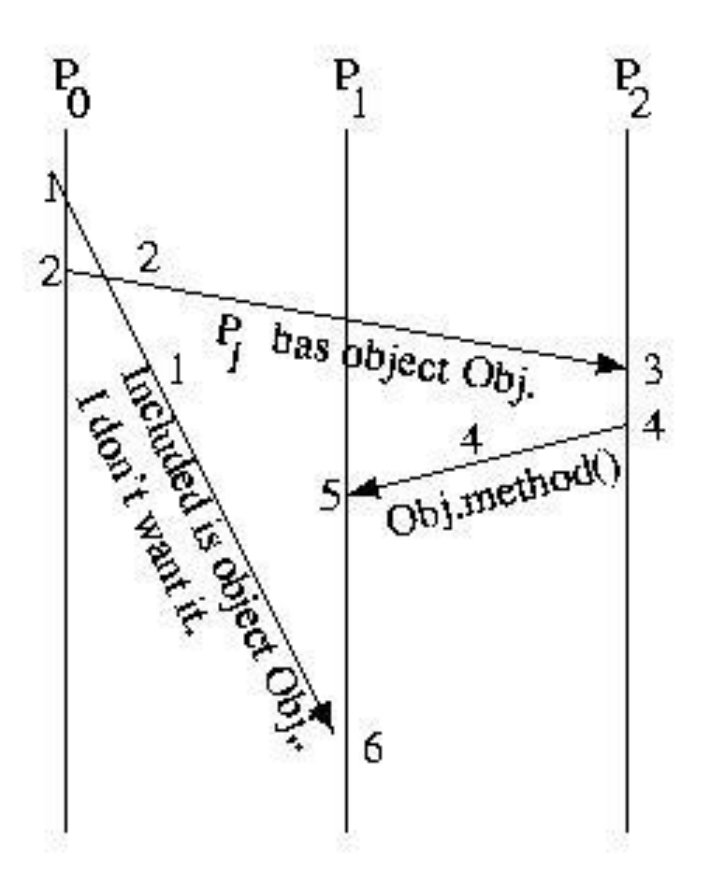
- Instead of just keeping our logical time, we keep a vector, V[], such that V[i] represents what we know of the logical time on processor i.
- V[our_id] is our logical time
- Send V[] vector with each message
- On receive, merge both vectors, selecting the greater of the corresponding elements from each. Then increment the component for self. The event is said to have happened at new (incremented) time.
- On send, increment time component for self. Send the updated timestamp vector with the message. The event is said to have happened at new (incremented) time.

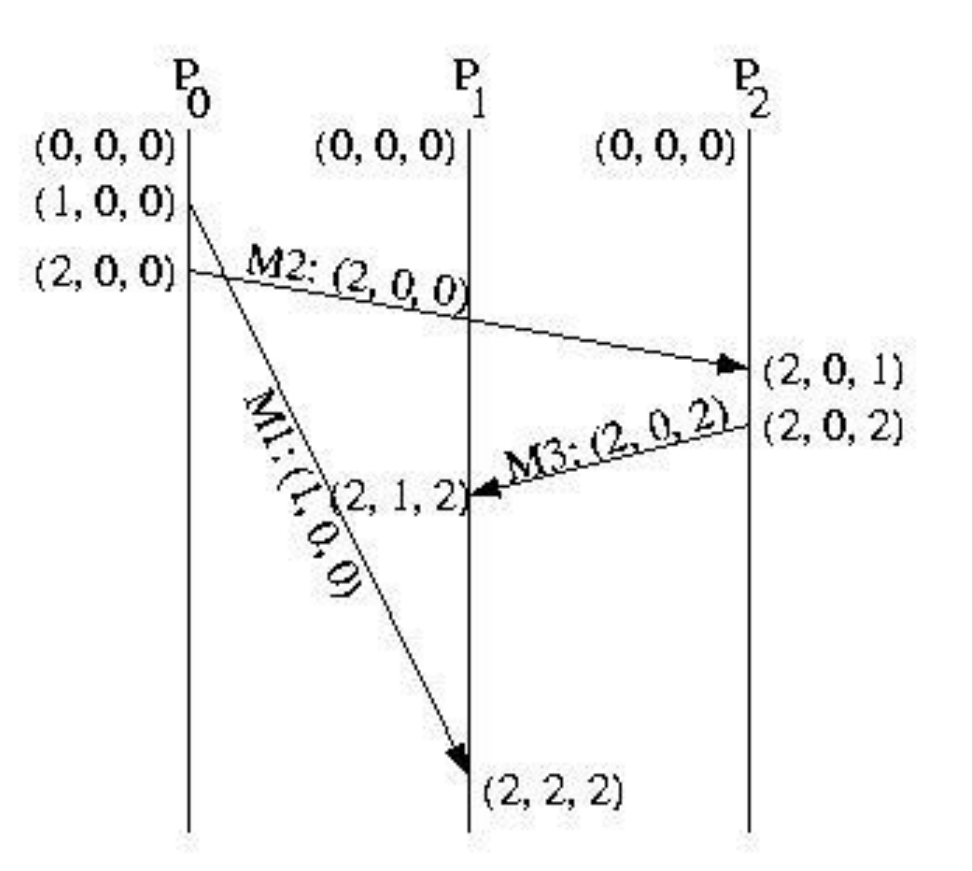
VECTOR TIME, BY EXAMPLE





WHAT HAPPENS HERE? TAKE 2





CODA VERSION VECTORS (CVVS)

- Each CVV contains one entry for each host server. Each entry is the version number of the file on the corresponding server. In the perfect case, the entry for each replica will be identical. But, should an update reach only a portion of the servers, some servers will have newer versions than others.

CODA CVVS AND CLIENT READS

In Coda, the client request a file via a three-step process.

- It asks all replicas for their version number
- It then asks the replica with the greatest version number for the file
- If the servers don't agree about the files version, the client can direct the servers to update a client that is behind, or inform them of a conflict. CVVs are compared just like vector timestamps. A conflict exists if two CVVs are concurrent, because concurrent vectors indicate that each server involved has seen some changes to the file, but not all changes.

CODA CVVS AND CLIENT WRITES

In the perfect case, when the client writes a file, it does it in a multi-step process:

- The client sends the file to all servers, along with the original CVV.
- Each server increments its entry in the file's CVV and ACKS the client.
- The client merges the entries form all of the servers and sends the new CVV back to each server.
- If a conflict is detected, the client can inform the servers, so that it can be resolved automatically, or flagged for mitigation by the user.

CODA CVV EXAMPLE

```
Initial:      <1,1,1,1>
              <1,1,1,1>
              <1,1,1,1>
              <1,1,1,1>

--------- Partition 1/2 and 3/4 ----------
Write 1/2:    <2,2,1,1>
              <2,2,1,1>

Write 3/4:    <1,1,2,2>
              <1,1,2,2>


--------- Partition repaired ----------
Read (ouch!) <1,1,2,2>
              <1,1,2,2>
              <2,2,1,1>
              <2,2,1,1>
```

MATRIX LOGICAL TIME

- We'll take a pass for now until we can see the applications
- But, common application include garbage collection, finding checkpoint recover lines, and more.