Time, Clocks, and the Ordering of Events in a Distributed System

Motivating example: a distributed compilation service

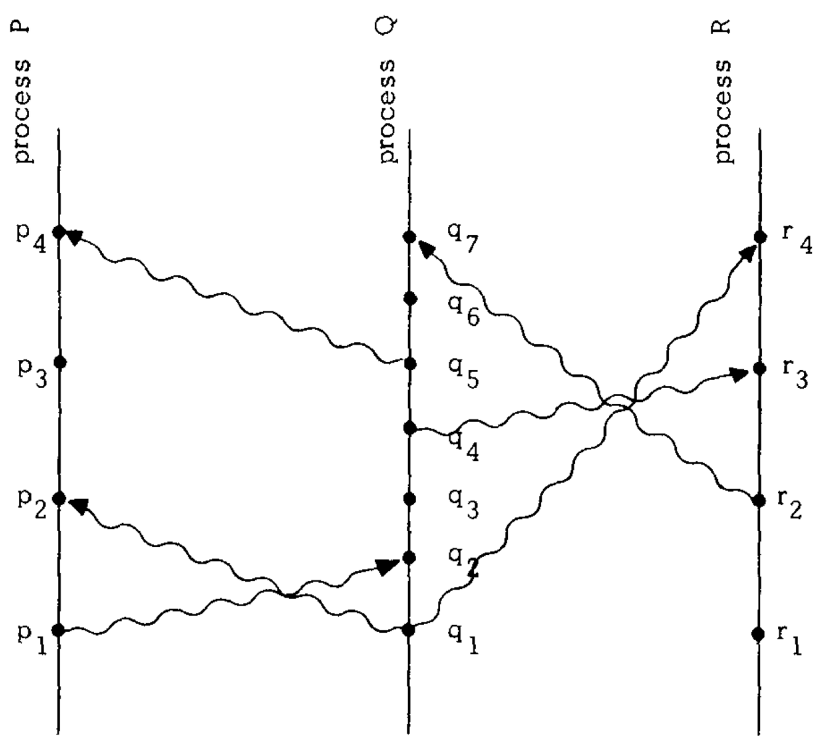
- FTP server storing source files, object files, executable file
- stored files have timestamps, set by client and preserved by server

- basic procedure to depcheck(A)
  - consider file A that depends on file B
    - if timestamp(A) < timestamp(B)
      - compile B
  - compile:
    - depcheck of B
    - fetch file
    - compile file
    - store result

- does this work?
  - need client clocks to be tightly synchronized
    - offset must be less than time to fetch/compile a file

- alternative is to use logical clocks, obviously

<u>Basic idea behind causal ordering</u>



- Three concepts we have to pin down: process, events, and messages
    - what is a process?
        - threads on a multiprocessor? Processes on OS? Etc.
    - three kinds of events in a distributed system
        - local computation
        - send(M)
        - receive(M)
    - what is a message?
        - shared memory communication?

<u>Lamport's happens before ("→") relation</u>

- within a process, if P1 comes before P2, then P1 → P2
    - why?
    - can we have P1, P2 concurrent with each other?

- across processes: message has two events, a = send(m), b = receive(m)
    - a → b
    - why?
        - in shared memory, aren't a,b at the same time? (No!)
- transitivity
    - if a → b and b → c, then a → c
    - why?
    - interpretation of happens before as "could have influenced", i.e., causality

- Physical interpretation: a → b if you can move from a to b in the diagram by following time within a process or message lines across processes

- two different events a, b are concurrent if neither a → b nor b → a
    - interpretation as "could not have influenced"

<u>Abstract logical clock</u>

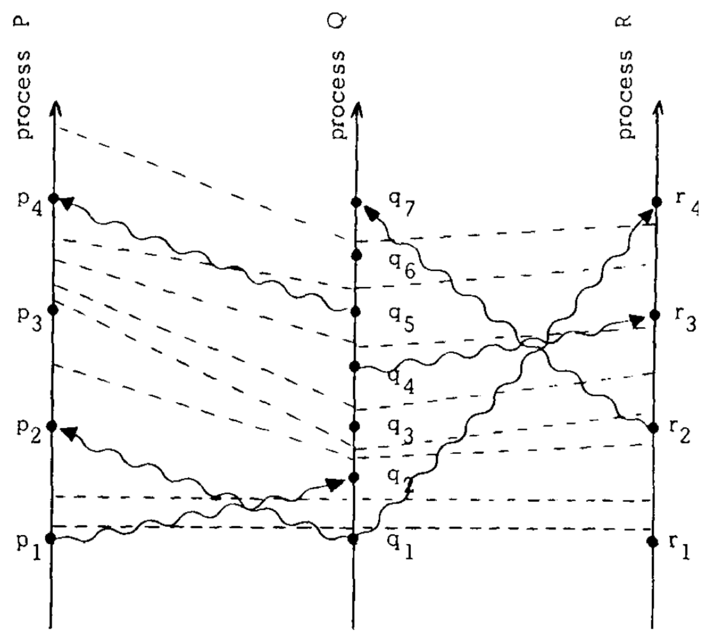
We want to build a system of clocks that respect causality

- each process Pi has a local clock Ci
- time of an event "a" at Pi is Ci(a)
- we want to logically synchronize the clocks, so that there is a global notion of time C(a) = Ci(A)
    - for this to be meaningful, the global clock C must respect lamport's "clock condition"
        - for any events a, b: if a → b then C(a) < C(b)
        - so, an event that happens before is earlier in global logical time
    - there are two subconditions that, if they are respected, imply the clock condition

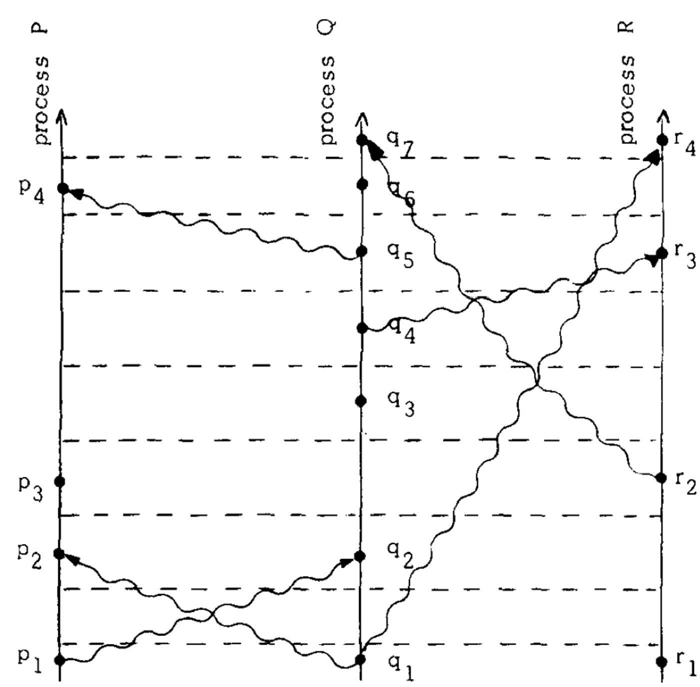
- C1: if a, b are events in Pi and a is before b, then Ci(a) < Ci(b)
- C2: if a = send(m) and b = receive(m), then Ci(a) < Cj(b)

Imposes a series of tickpoints on the diagram

- C1: at least one tick between any two events on a process line
- C2: at least one tick between the send and receive of a message



and then straighten the lines:



Implementing logical clocks

There are many different implementations of logical clocks that are consistent with Lamport's clock conditions. He gives one:

- Each process Pi maintains a local counter Ci
- IR1:
  - Each process Pi increments Ci between any two successive events
- IR2:
  - Each process piggybacks timestamp Tm on a message it sents, where Tm is Ci at the time of sending m
    - If a = send(m) by Pi, then m contains Tm = Ci(a)
    - On receiving m, Pj sets Cj to max(Cj, Tm+1)
    - The receipt of m is a separate event that then separately advances Cj

- Properties of this implementation?
  - Respects causality
    - If a → b, then C(a) < C(b)
  - But, converse is not true
    - If C(a) < C(b), don't know that a → b
    - Why? Both cases are possible
      - Could be concurrent
      - Could be causally preceeding

Global ordering

- Use logical clock to set order
- If tie, use process IDs as tiebreaker
- i.e., global order is (Logical timestamp) . (process ID)

Problems with causal ordering

- There could be events outside of the system that have causal influence on the evolution of the system
  - e.g., users telephoning each other. System could choose to order events in way that breaks the telephone causality, since it doesn't know the events are causally related.

  - Is there a way to implement a system that captures all forms of causality?
    - Hypothetically, yes – this is the Einstein relativity and physical clocks

        - Need to keep clocks in tight synchronization with each other, in particular, any pair of clocks' offsets must be less than min transmission time between them
    - Hard question:
        - If all you can do to synchronize clocks is use the messages inherent in the system, can you synchronize tightly enough to meet this bound?
        - Lamport argues yes

- Causal ordering doesn't actually imply influence, just potential influence
    - Causal consistency algorithms tend to overconstrain as a result

Q: how far from physical time can logical time diverge? I.e., if logical time says two events are concurrent, how far apart in time could they actually occur?

- Arbitrarily far, as clocks can run at independent rates until interaction occurs
- Depends on clock synchronization, depends on how long until interaction (or transitive interaction) occurs.

<u>Alternate system of logical clocks: vector timestamps, a.k.a. version vectors</u>

Remember that with Lamport clocks, if a → b, then C(a) < C(b), but the converse is not true.

We can build a logical clock that satisfies the clock condition, but for which the converse is true: a vector clock.

- Each node maintains a vector of counters, one for each node in the system
- IR1:
    - If two events a and b in Pi, and b is after a, then Pi sets VCi[i] = VCi[i]+1
- IR2:
    - If a is "Pi sends m" and b is "Pj receives m", then:
        - Pi increments VCi[i] and copies its full vector clock into m
        - For each k, VCj[k] = max(VCj[k], timestamp[k])

Need to know how to compare vector clocks:

VCi < VCj iff for all k, VCi[k] <= VCj[k] and there is one k s.t. VCi[k] < VCj[k]

It's basically the partial order captured perfectly.

Back to distributed make

- How to fix?
  - Use different ordering: causal ordering
  - Make clocks more strongly synchronized
    - Physical clock ordering is consistent with “happens-before” relationship if and only if length(event + msg transmit) > d
      - Makes sure timestamps cannot go backwards
    - How tight? If clocks | Ci – Cj | < d for all I,j then need length (compilation + msg transmit) > d
      - Not always true, especially as compiles get faster
  - Or, change timestamps at file server!!
    - Why does this work?