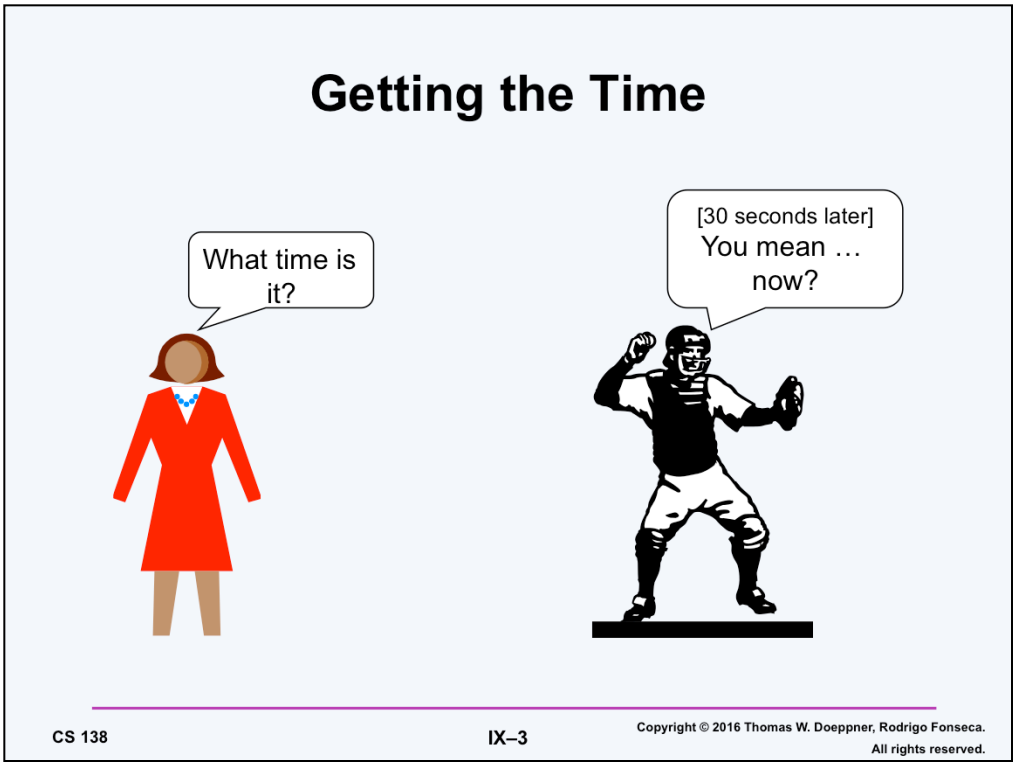
CS 138: Time

This material is partially covered in Chapter 14 of Coulouris, Dollimore, Kindberg, and Blair.

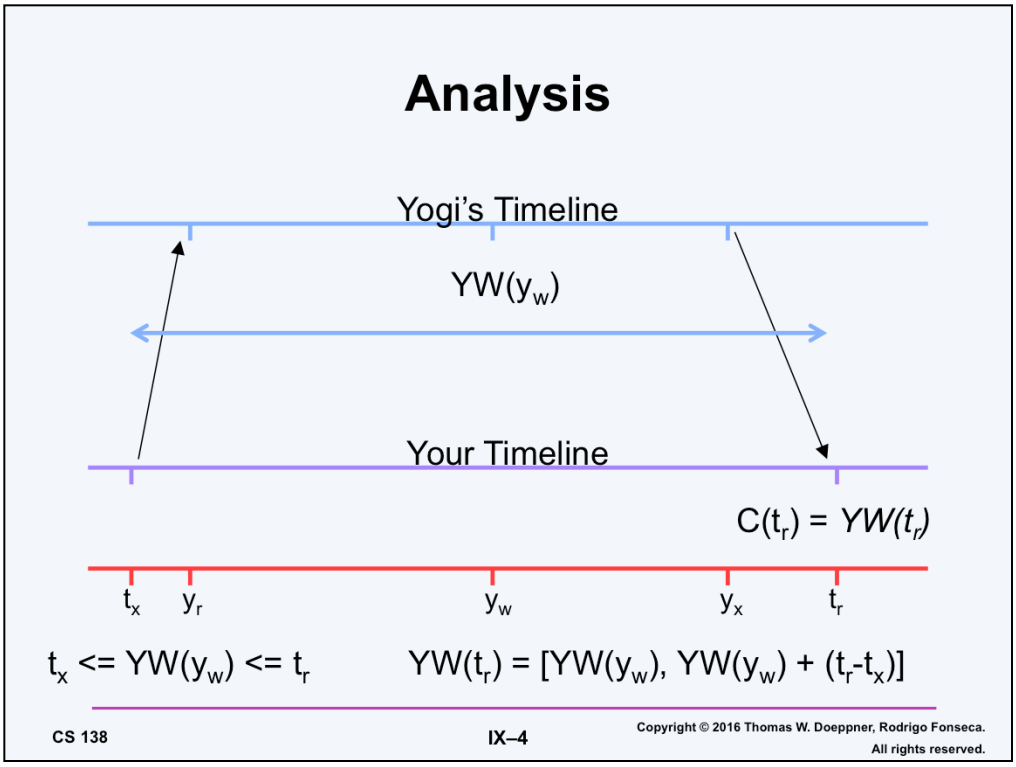
Topics

- Clock Synchronization
- Logical Clocks
- Causality
- Vector Clocks



It is reported of Yogi Berra, the old-time baseball player, that he was once asked, “What time is it?” After thinking for a bit, he responded, “You mean … now?” In this spirit, we talk about how nodes within a network synchronize their clocks. We assume that there are one or more time servers on the network whose clocks are presumed to be accurate. Thus any client can ask such a server for the correct time, but unfortunately a delay is involved both in propagating the request to the server and in getting the response from the server. Thus the server indeed gives you the correct time; the problem is that you don’t really know when this was the correct time.

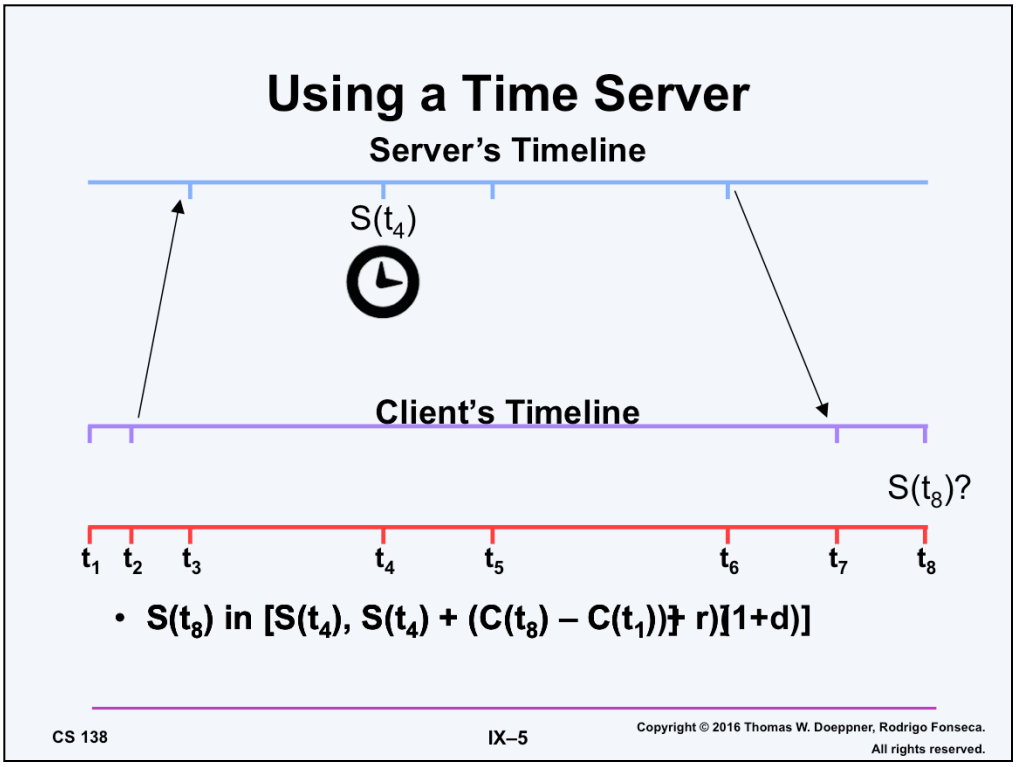
(Other Yogi Berra’isms include: “if you come to a fork in the road, take it”, “you can observe a lot by watching”, “it was déjà vu all over again”, “you should always go to other people’s funerals, otherwise they won’t come to yours”, “I really didn’t say everything I said”.)



Suppose, at a better moment, we asked Yogi for the correct time, and after thinking about it for some time he eventually responded, "3:30." Assuming that Yogi did indeed consult his watch to determine his answer and that his watch was more or less correct, how can we make use of his response?

In the picture, let t_x represent the time at which you uttered your request to Yogi. Some time later, y_r , the request arrived at Yogi's ears. He thought about it for a bit and then, at time y_w , glanced at his watch. He thought some more and, at time y_x, uttered the time he had recently read from his watch. At time t_r his words reached your ears. So you know what Yogi's watch read at time y_w, you just aren't certain how far in the past y_w was.

Assuming that you have a watch and the purpose of your query was merely to set it, you can measure the amount of time that elapsed from t_x to t_r. Let YW(x) represent what Yogi's watch reads at time x; thus you know YW(y_w) and what you would like to know is YW(t_r). If we assume that y_w could have appeared any place within the interval [t_x, t_r], then your estimate of YW(t_r) is somewhere within the interval [YW(y_w), YW(y_w) + (t_r–t_x)].



Consider the issues involved in keeping computer clocks in sync by means of a time-server computer. For example, suppose a client wishes to update its clock by obtaining the correct time from the server.

At time t_1 the client application makes the request to query the server for the time of day.
At time t_2 this request is actually transmitted.
At time t_3 the request arrives at the server, and
at time t_4 the request is passed to the server routine that deals with time and the clock is read.
At t_5 this routine requests that the time be transferred to the client,
at time t_6 the time is actually transferred,
at time t_7 it arrives at the client, and
at time t_8 it is finally received by the original application.

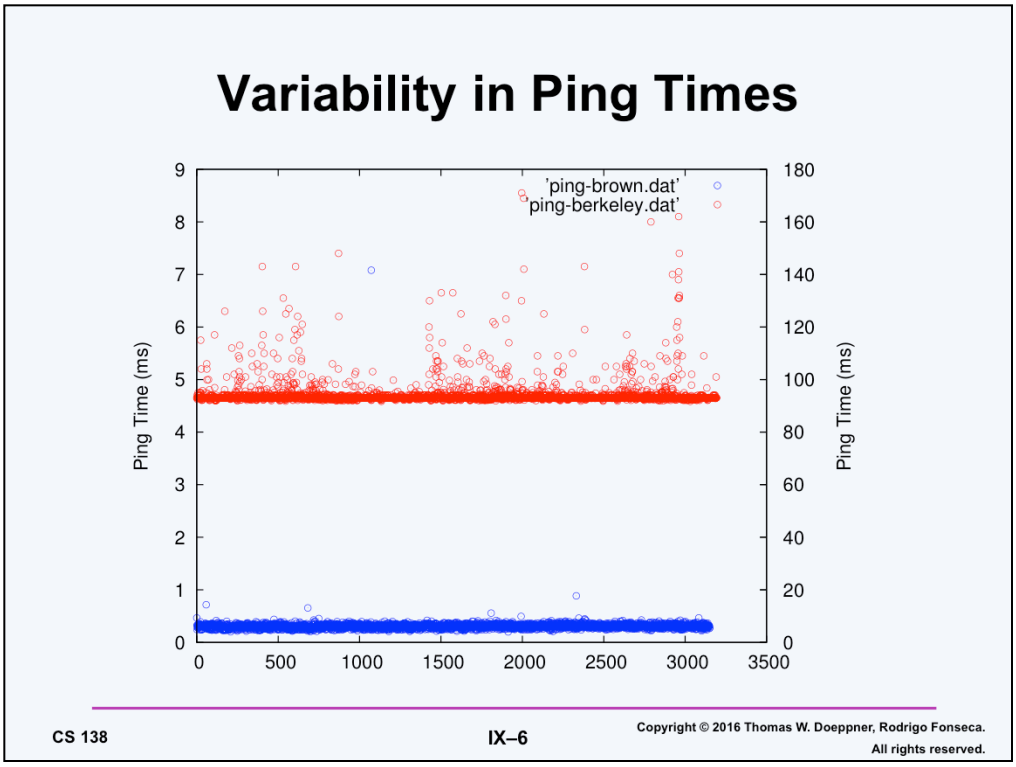
Let $S(t_4)$ be the value of the server's clock at time t_4. The client is, of course, interested in $S(t_8)$. Let $C(x)$ be the value of the client's clock at time x. The time that elapsed between when the client made its request and when it received the response is $C(t_8) - C(t_1)$.

Thus what the client learns from this transaction is that $S(t_8)$ is in the range $[S(t_4), S(t_4) + (C(t_8)-C(t_1))]$.

There are at least two problems with the above computation. First of all, the client's (and the server's) clock is not continuous; it "ticks" in discrete units. Let ρ be the length of a clock tick. If t_1 occurred just after a tick and t_8 occurred just before a tick, then the actual value of $t_8 - t_1$ could be as great as $C(t_8) - C(t_1) + \rho$. A further problem is that the client's clock may drift over time. Let us assume that this drift rate is bounded by a constant δ. Thus an upper bound on $t_8 - t_1$ is $(C(t_8)-C(t_1) + \rho)(1 + \delta)$.

Our final estimate of $S(t_8)$ is $[S(t_4), S(t_4) + (C(t_8)-C(t_1) + \rho)(1 + \delta)]$.

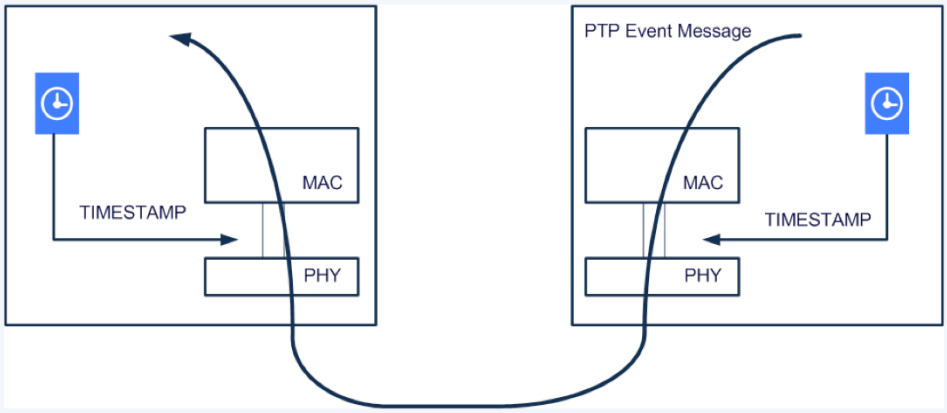
What are some typical numbers? ρ might be ten milliseconds, the actual communication delay might be 20 milliseconds, and δ might be .0004.



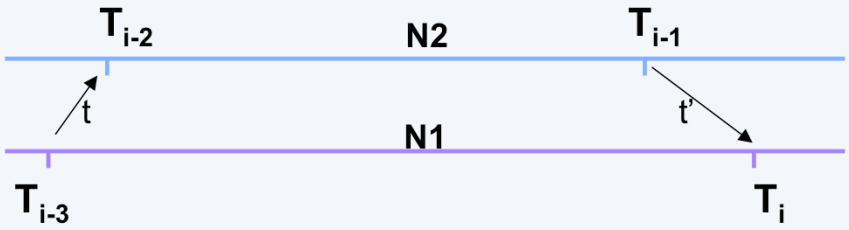
How does this affect the previous computation?

How do we minimize the effect of this variability? (Take multiple measurements and take the MIN of the total delta)

Precision using HW time-stamping (PTP Protocol)



Example: NTP



Goal: calculate offset O

$a = T_{i-2} - T_{i-3}$, $b = T_{i-1} - T_i$

$d_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) = a - b$;

$o_i = (a+b)/2 = (T_{i-2} + T_{i-i})/2 - (T_i + T_{i-3})/2$

$t = T_{i-2} - O - T_{i-3} = a - O >= 0 \rightarrow a >= O$

$t' = T_i - (T_{i-1} - O) = O - b >= 0 \rightarrow b <= O$

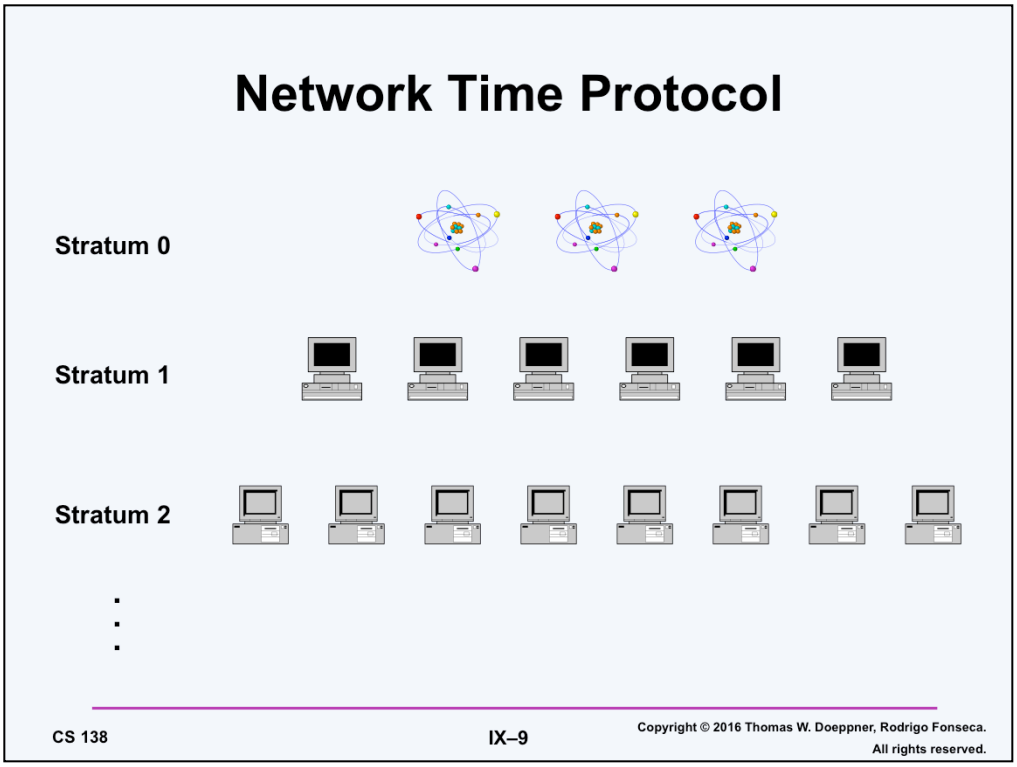
$b = (a+b)/2 - (a-b)/2 <= O <= (a+b)/2 + (a-b)/2 = a$

$o_i - d_i/2 <= O <= o_i + d_i/2$

di is the difference between the intervals in the two servers, and thus corresponds to the time spent in the network

oi is the offset between the average of the two intervals

We use a and b here just to make the intermediate steps shorter, but the gist of the algorithm is that the interval in N2 can't start before the interval in N1, and can't end after the end of N1.



The Internet's Network Time Protocol (NTP) also uses the Marzullo approach, though things are organized a bit differently. See http://www.eecis.udel.edu/~mills/ntp/html/index.html#docs for details. Time servers are organized into strata, depending on their "distance" from the ultimate time source. Stratum 0 is the collection of time sources themselves, stratum 1 is the collection of servers connected directly to time sources, etc. Servers of one stratum connect to a number of servers/devices at the next lower stratum, rule out liars, and average the time of apparent truth-sayers. (In NTP terminology, liars are known as "falsetickers" and truthful servers are "truechimers.") Windows uses a simplified version of NTP known as SNTP (simple NTP) — clients contact just one server. Linux and most other Unix implementations use the full-blown NTP protocol, even for clients. Brown University provides a stratum-2 time server at ntp.brown.edu.

Nobody's Perfect

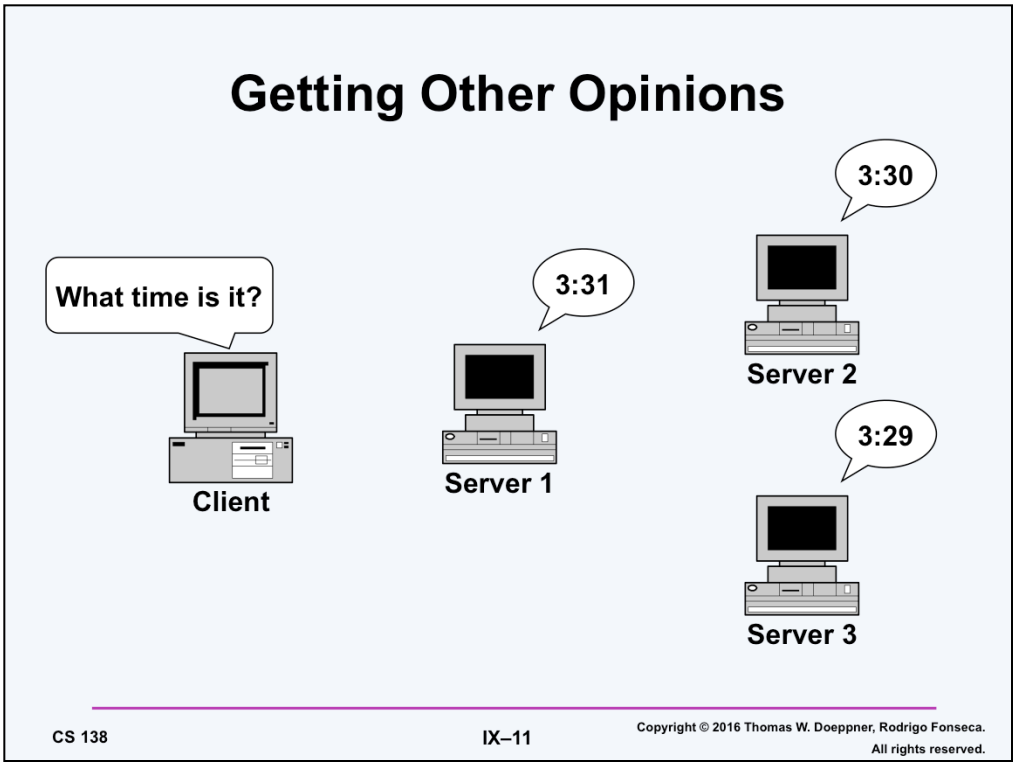
- **Inaccuracy:** $T(t) - I(t) <= t <= T(t) + I(t)$
- **Drift:** $I(t_2) <= d \cdot (t_2 - t_1) + I(t_1)$ (a bound on I)
- **Resolution: each tick represents *r* seconds**
- **Clocks are assumed always to make forward progress**

$t_2 >= t_1 + r/(1-d) \rightarrow T(t_2) >= T(t_1) + r$

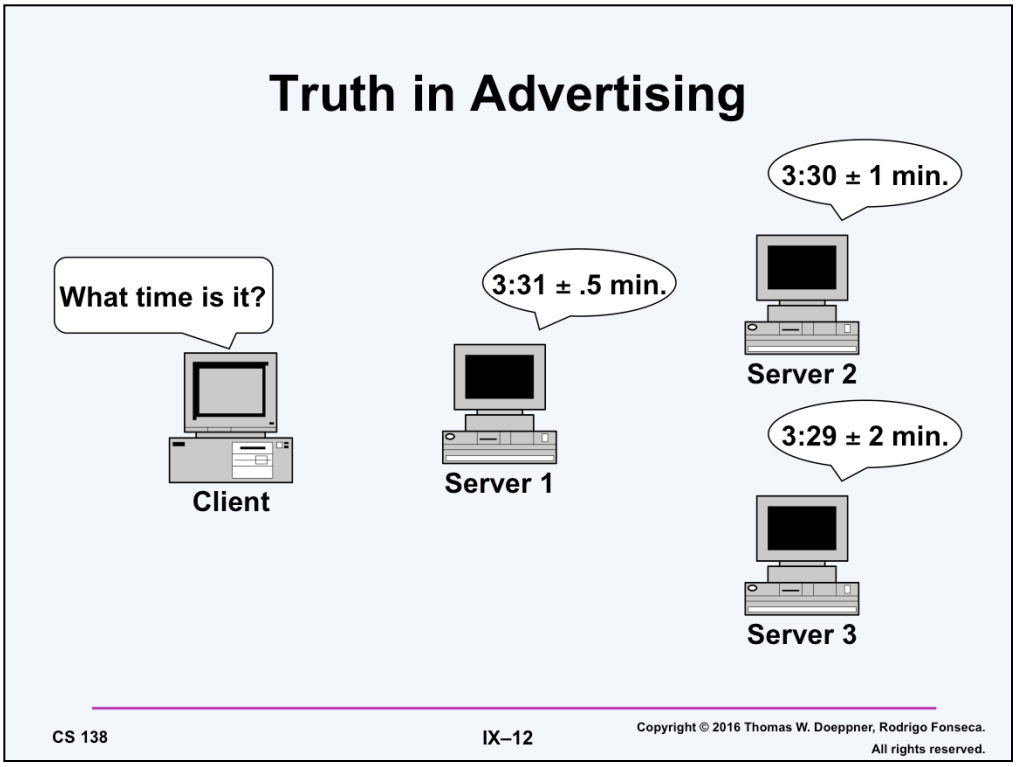
Here we introduce some terminology so that we can be precise about how imperfect things are. The *inaccuracy* of a clock is a bound on the error of a clock as a function of time; thus a clock does not tell us precisely what time it is, but rather that the correct time lies somewhere within an interval.

Unfortunately the inaccuracy of a clock changes with time. We don't know precisely what this change is, but we can place **a bound** on the rate of change. This bound, which is assumed to be constant, is known as the drift. Clocks on digital computers tend to be counters, with each tick indicating that a certain amount of time has passed. The length of time corresponding to each tick is known as the *resolution*.

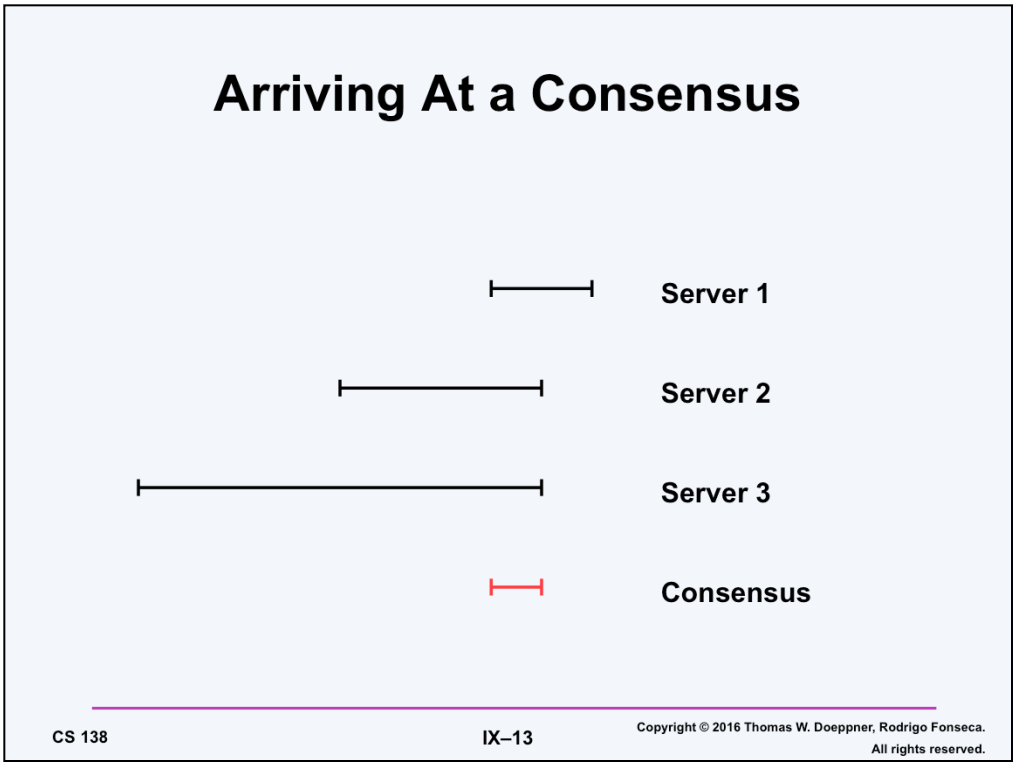
We assume that clocks always make forward progress. What this means is that the clock will tick if we wait long enough, and that it never ticks "backwards." How long do we have to wait for a clock to tick? We know that each tick represents the passage of ρ seconds, but this is ρ seconds as measured by the clock. How much time does this correspond to in reality? Let t_1 and t_2 represent two instants in "real time." From our definition of inaccuracy, we know that $t_2 - t_1$ is no greater than $T(t_2) + I(t_2) - (T(t_1) - I(t_1))$. This, in turn, given our definition of drift (δ), is no greater than $T(t_2) + \delta \cdot (t_2 - t_1) + 2 \cdot I(t_1) - T(t_1)$. If we let $T(t_2) - T(t_1)$ be ρ, the clock resolution, and if we assume that $T(t_1)$ is the correct time, then $t_2 - t_1$ is no greater than $\rho + \delta \cdot (t_2 - t_1)$. Thus the duration of the time interval, $t_2 - t_1$, required to insure that the time measured by the clock is at least one clock tick (ρ), is no greater than $\rho/(1 - \delta)$.



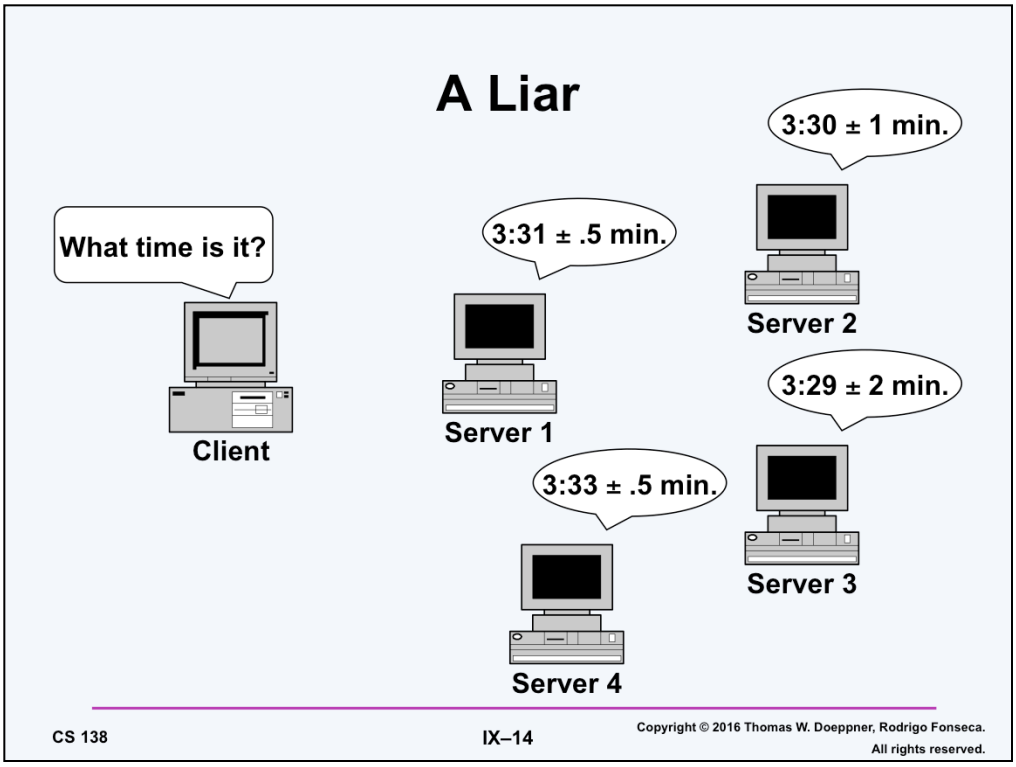
A client might decide to improve its chances of getting the correct time by consulting a number of servers and then somehow "averaging" the results. However, if each server gives it a different result (and each server claims to be correct), there is no basis by which averaging can be applied.



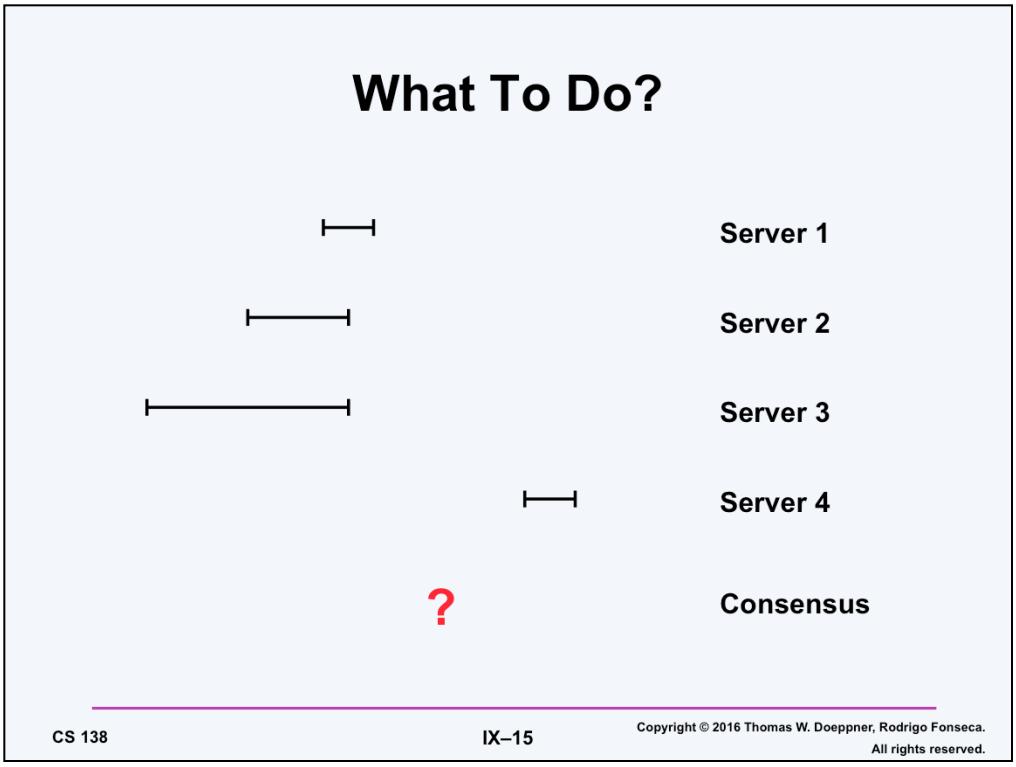
A responsible server will reply not only with what it considers to be the correct time but also with a bound on the accuracy of this time estimate. Thus the client obtains a set of intervals from which it can determine a current time interval that is tighter than any interval obtained from individual servers.



If each server has supplied a correct interval representing the current time, then the current time must lie in the intersection of all of the servers' intervals. Though we don't know exactly where this point is, the client can use the intersection interval as its current time interval (since the intersection interval contains the correct time).

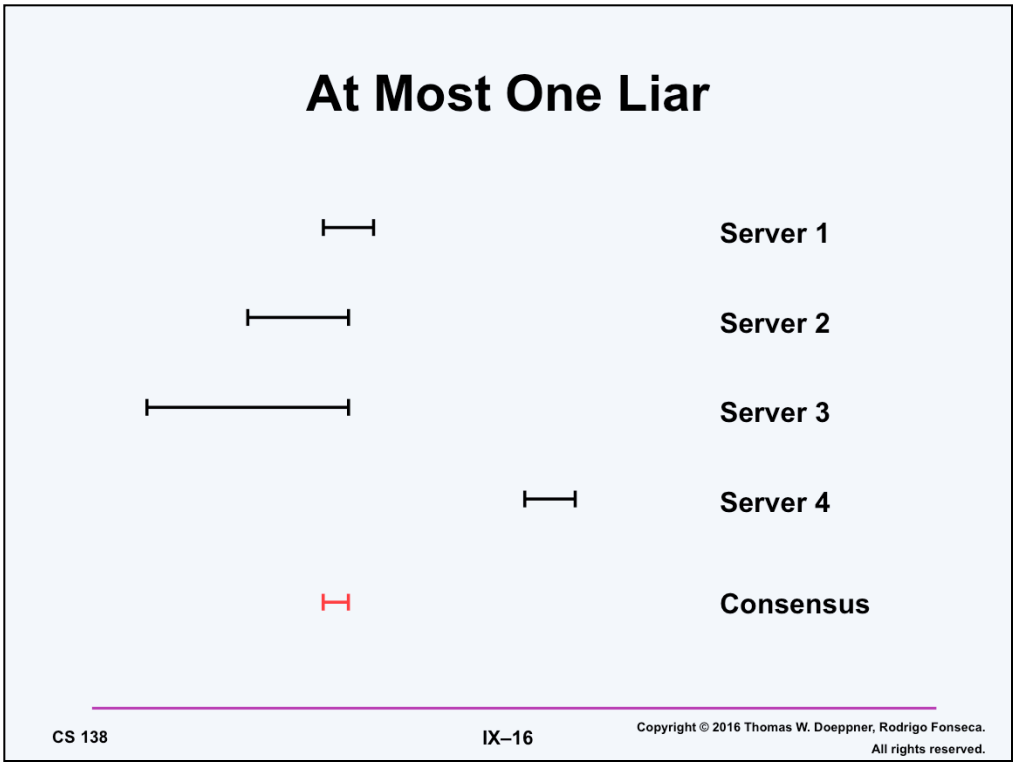


One or more of the servers might be totally wrong about the time of day.

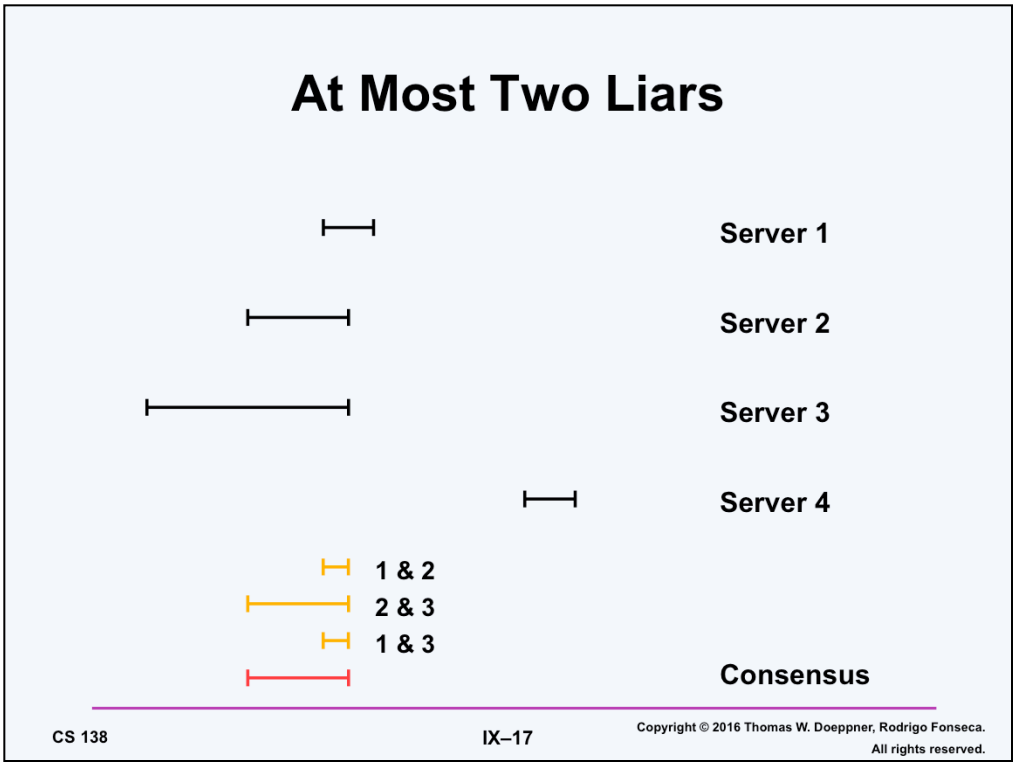


The intersection of these intervals is null. If we assume that there is at most one faulty server, then any point contained in at least three of the intervals can represent the correct time.

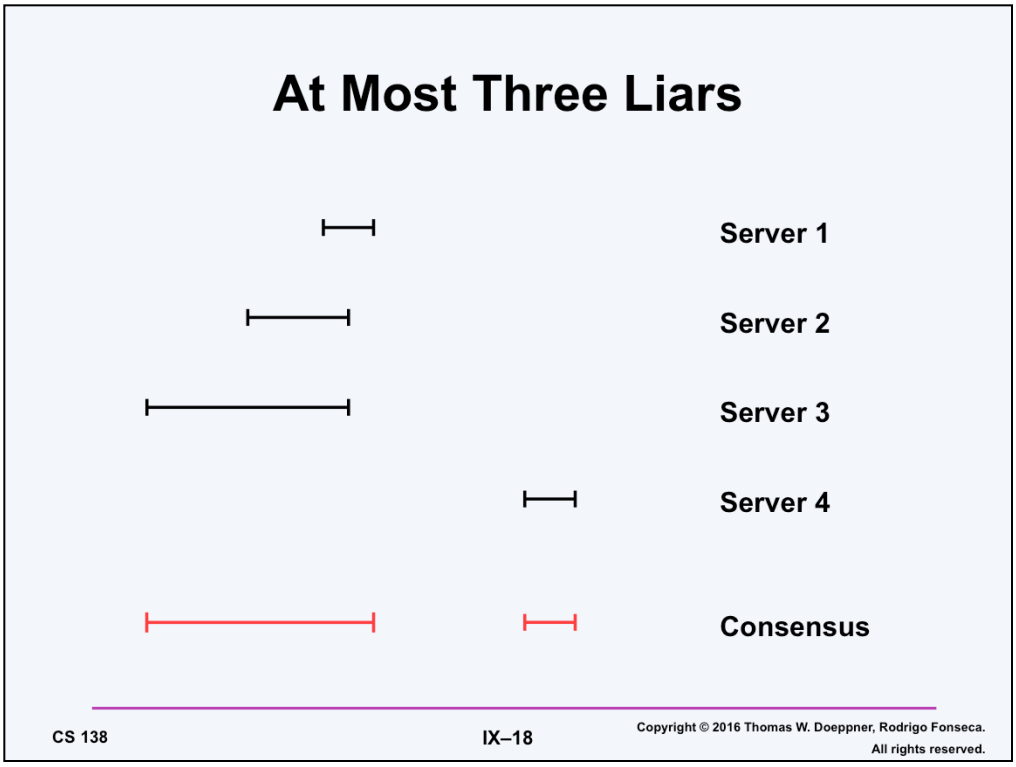
In general, if we have m servers and at most f of them are faulty, then any point contained in at least $m - f$ of the time intervals can be correct.



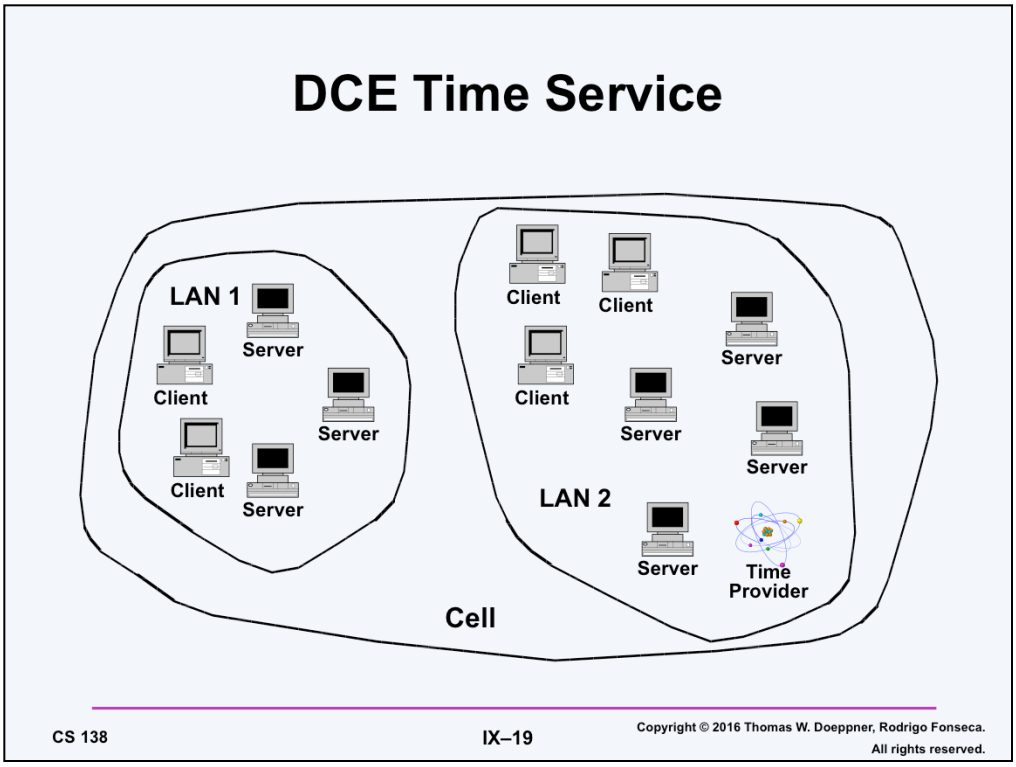
If there is at most one liar, then at least three of the servers must be telling the truth. In this case, the truth-sayers must be the first three servers and the consensus is as before.



However, if all we can determine is that there are at most two liars, then we must look at possible pairs of truth-sayers. Server 4 is still ruled out, but the two truth-sayers might be 1 and 2, or 2 and 3, or 1 and 3. Thus the correct time lies in **the union of their intersections.**



If all we can say is that there are at most three liars, then any one of the servers might be telling the truth. In this case, the correct time lies in the union of all four intervals, resulting in a disjoint interval.



The basic approach to clock synchronization that we just discussed is due to Keith Marzullo: K. A. Marzullo. *Maintaining the Time in a Distributed System: An Example of a Loosely-Coupled Distributed Service*. Ph.D. dissertation, Stanford University, Department of Electrical Engineering, February 1984.

The DCE Distributed Time Service (DTS), based on the Digital Time Synchronization Service (DTSS) specification, is provided by a collection of clerks and time servers. Clerks reside on individual computers and are responsible for maintaining the time for their local computers. They contact the time servers, acting as clients, to obtain the information necessary to keep their clocks reasonably accurate.

Time servers fall into two categories: local time servers and global time servers. A collection of local time servers is responsible for keeping the time on an individual LAN. (LAN, in this context, probably means local area network, but really means a collection of computers which are relatively close to one another (in terms of communication delays).) These servers periodically synchronize their clocks with one another's. On a more frequent basis, clerks synchronize with (at least) a subset of the local time servers' clocks and, possibly, with some global servers.

Global time servers are time servers in other LANs (within the cell) that clerks and local time servers may contact if additional sources of time information are needed. The idea is that local time servers are, with respect to communication time, near one another and the clerks, while global time servers are somewhat farther away.

The ultimate source of time is the time provider (TP). This provides the time along with a (presumably tiny) inaccuracy. It might be an atomic clock, or some other time service such as NTP (Network Time Protocol) of the Internet. A time server synchronizes with a TP if one is available (otherwise it synchronizes with other time servers).

What's Time?

- GMT
  - determined by astronomical observations at Greenwich
  - improved versions, accounting for irregularities of earth's rotation and orbit, are UT0, UT1, UT2
- International Atomic time (TAI)
  - based on transitions of energy levels of cesium atom
  - synchronized with earth time in 1958
  - earth has been running slow since then
    - UTC: leap seconds added as necessary to adjust

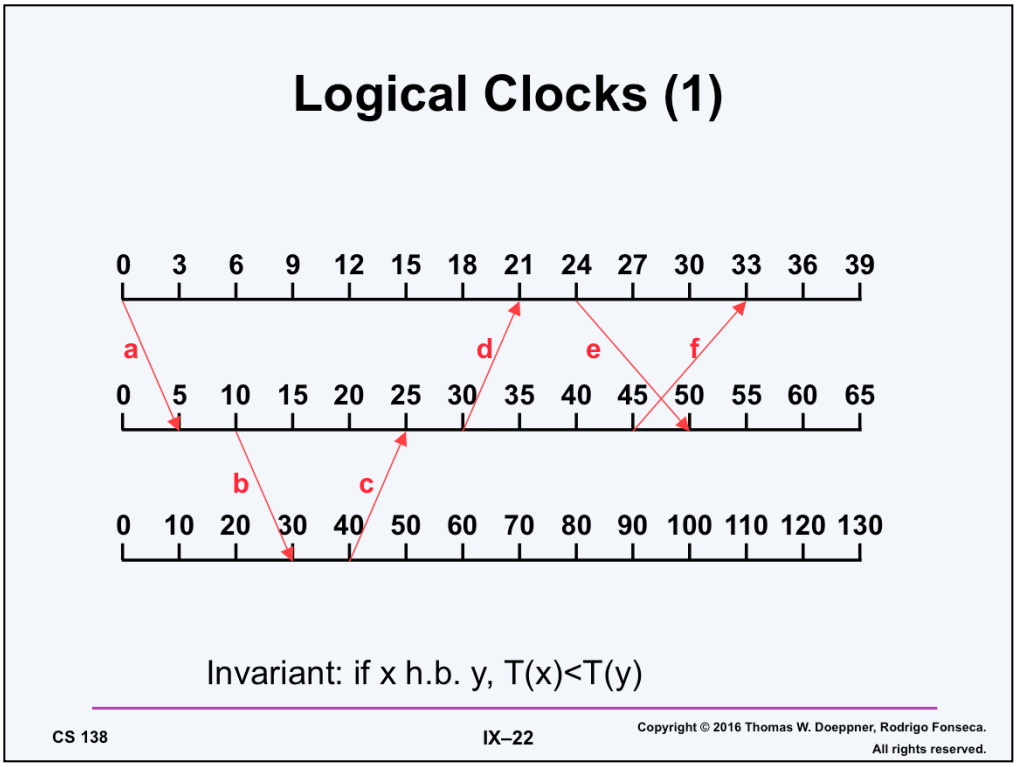
UTC is pronounced *coordinated universal time* in English and *temps universel coordonne* in French. "UTC" was chosen apparently so as not to show favoritism to either language. To find the official US time, go to http://www.time.gov/timezone.cgi?Eastern/d/-5/java.

What's Time? (Really …)

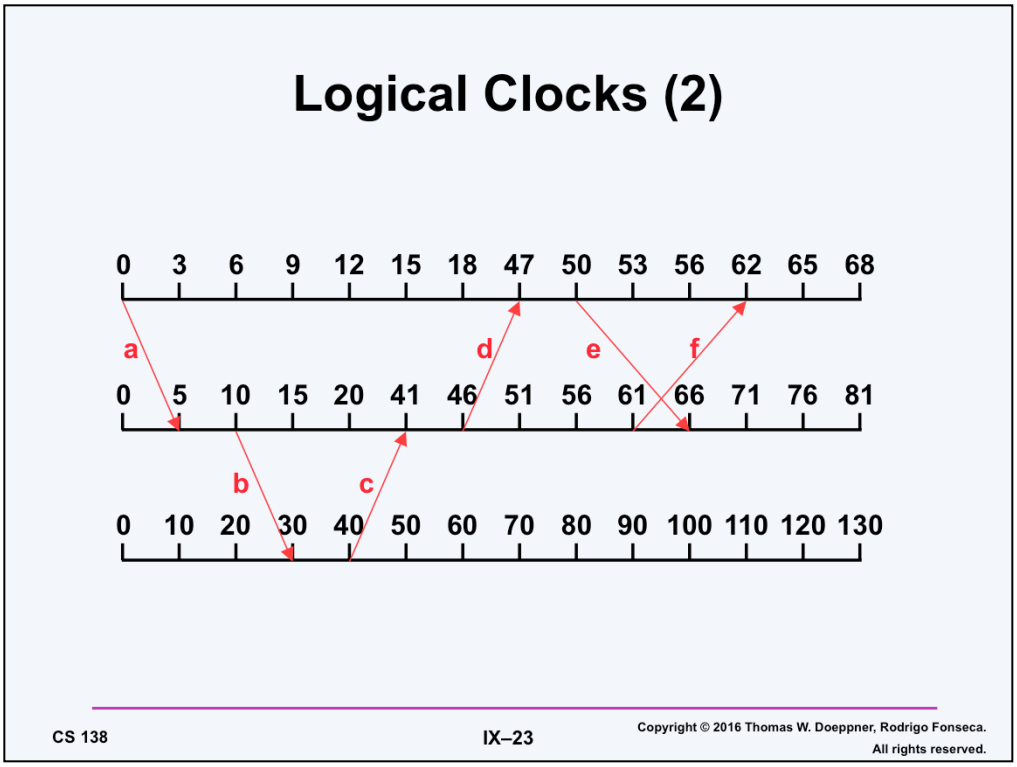
- "Time is God's way of keeping everything from happening at once."

  –various sources
- Timestamp events to determine order of their occurrence
- Problems
  - doesn't work if clocks either aren't perfectly in sync or don't have sufficiently fine resolution
  - an order might not be meaningful if two events have no causal relationship
- A → B iff A could possibly have had an effect on B (pronounced A *happened before* B)

According to Wikipedia, the quote is "variously attributed to Woody Allen, Albert Einstein, John Archibald Wheeler, and Anonymous."



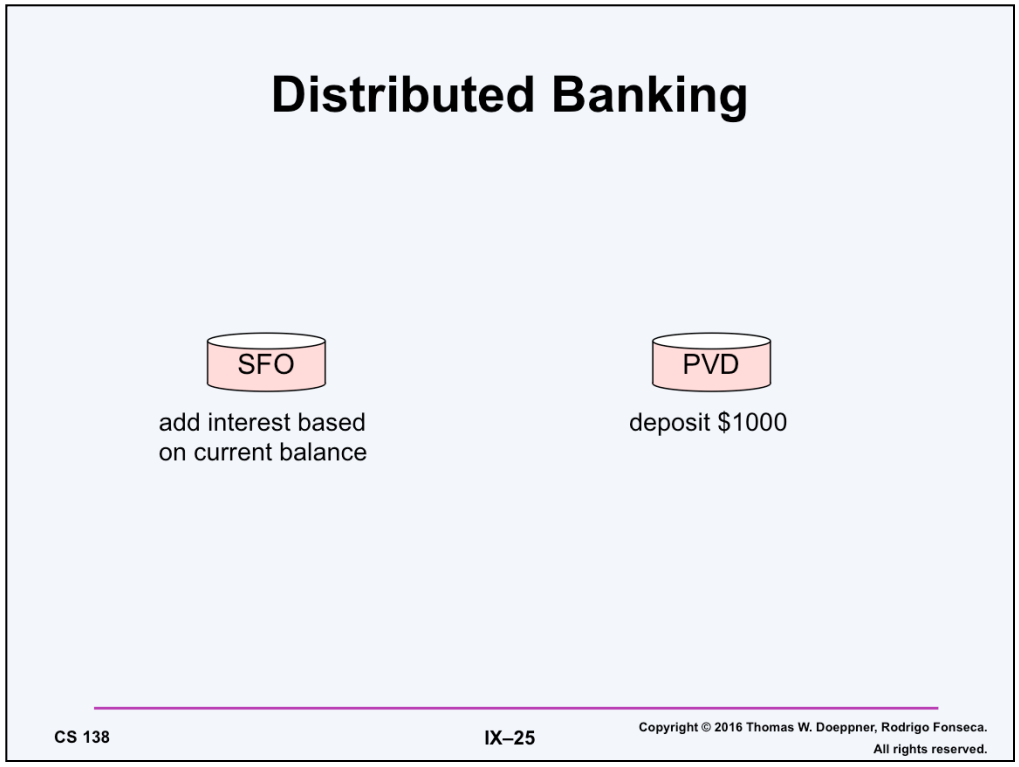
In many situations it's not so important that we keep our clocks in perfect synchrony, but that we be able to consistently determine the order of events. In the slide are time lines for three machines. The arrows represent messages traveling from one machine to another. If a is one such message, then s_a represents the sending of a and r_a represents the receiving of a. T_i is the clock function of machine i. Then $T_i(s_a)$ is the time that a was transmitted according to machine 1's clock. Assuming our clocks are well behaved, $T_1(s_a) < T_1(r_a)$. Since message a was transmitted from machine 1 to machine 2, it makes sense for $T_1(s_a) < T_2(r_a)$. **But look at message *c* in the slide**. We certainly want $T_3(s_c) < T_2(r_c)$, but the clocks of the machines 2 and 3 are a bit out of sync, so this is not the case. Even if we use the clock synchronization techniques we discussed earlier, there still is no guarantee that the clocks will be close enough in sync for the desired inequality to hold. What we need is some way of **guaranteeing that if event *x* on machine *i* has a causal effect on event *y* on machine *j*, then $T_i(x) < T_j(y)$**. Note that it's not important that our clock functions actually give us time—sequence numbers are good enough.



The approach used to solve this problem is due to Leslie Lamport. When one sends a message, one should include the current time along with the message. Then when you receive a message, you should make certain that the current time as reported by your clock is greater than the timestamp supplied with the message**. If not, you should then advance your clock** so that it does report a time greater than the timestamp supplied with the message. This simple procedure ensures us that the simple inequality of the previous slide holds, as long as causality is due only to messages. If it is important that all events have unique timestamps, ties are broken by comparing process IDs.

Using Logical Timestamps

- We can use Lamport Clocks to create a total order of events agreed to by all processes



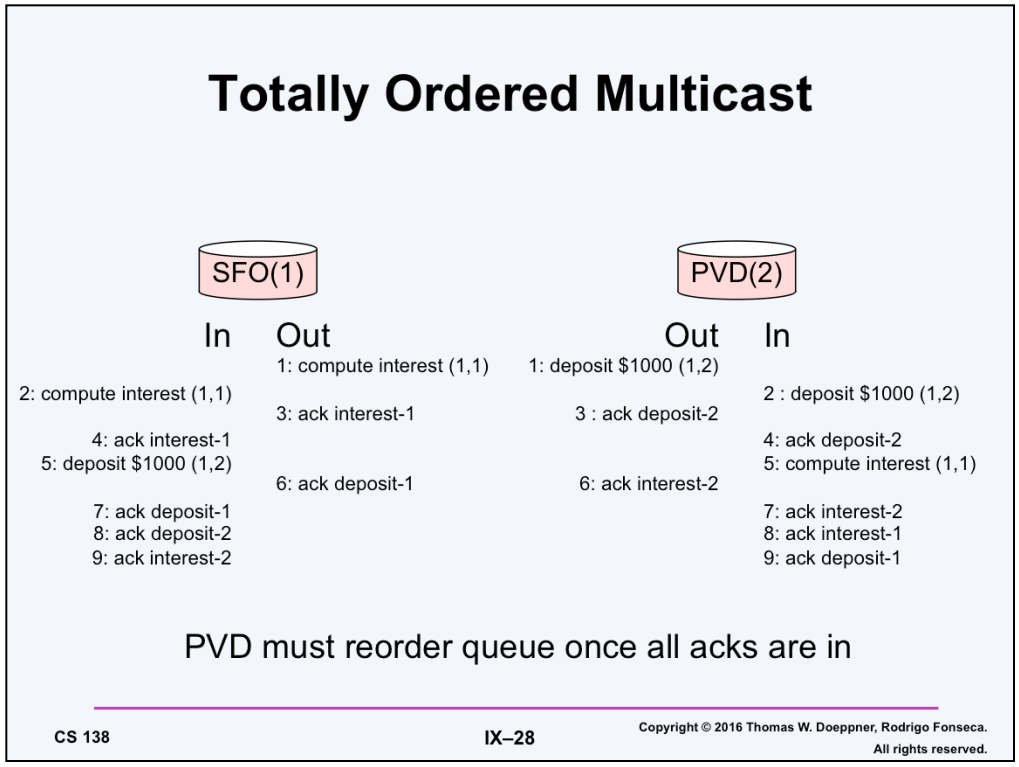
Both the San Francisco and the Providence branches of the bank have a complete copy of the database. The San Francisco office multicasts a request to add to your account the accrued interest based on your current balance. The Providence office multicasts a request to deposit $1000 into your account. It certainly matters to you which action takes place first. It matters to both you and the bank that whatever order is taken, the two sites agree on the order.

Total Order

- Tie-breaking rule
  - what if $T_i(a) = T_h(b)$?
  - *a* comes before *b* iff $i<h$
- Total order for all events in a distributed system

Totally Ordered Multicast

- To send multicast:
  - tag message with sender's timestamp (<time, sender ID>)
    - sender receives own multicast
- On receipt of message
  - queue message in timestamp order
  - multicast an acknowledgement
- On receipt of acknowledgement
  - link to acknowledged message
- Deliver message to application when
  - message is at front of receive queue
  - has been acknowledged by all



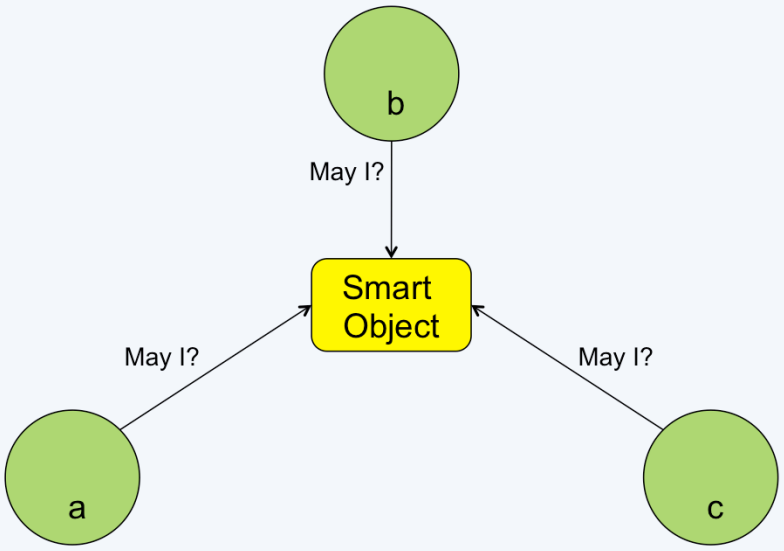
(1,2) means operation 1, id 2.

Note that neither SFO nor PVD can deliver operations that are in their in queues to their applications until they've been ack'd by both parties. PVD must reorder its in queue and deliver the compute-interest request from SFO before its own deposit request.

Mutual Exclusion

- Central server
- Logical clocks

Central-Server Mutual Exclusion

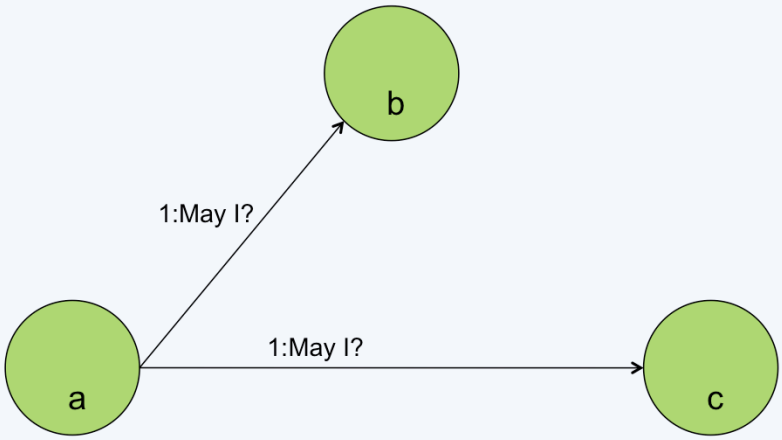


Mutual Exclusion with Logical Clocks

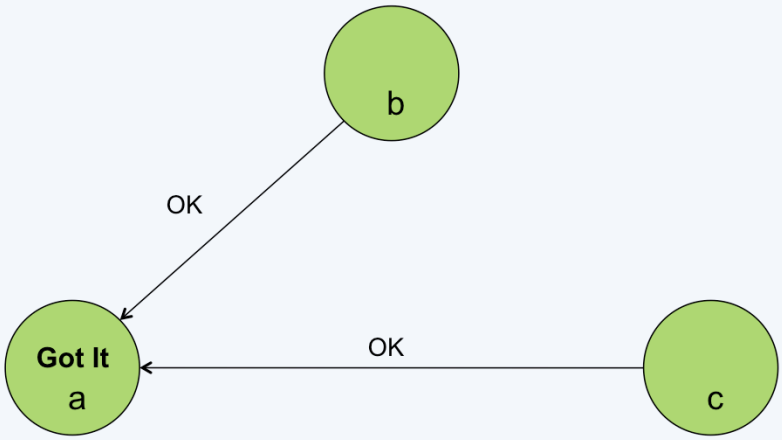
- Requester
  - multicast request with timestamp
  - proceed when all other parties respond OK
- Receiver of request
  - if neither using nor waiting for resource, respond OK
  - if waiting for resource, respond OK if request's *timestamp is lower than own*, otherwise queue request
  - if using resource, queue request
- When finished
  - respond OK to queued requests

This algorithm is due to Ricart and Agrawalla, and is an optimization of the mutual exclusion algorithm presented by Lamport in the 78 paper.

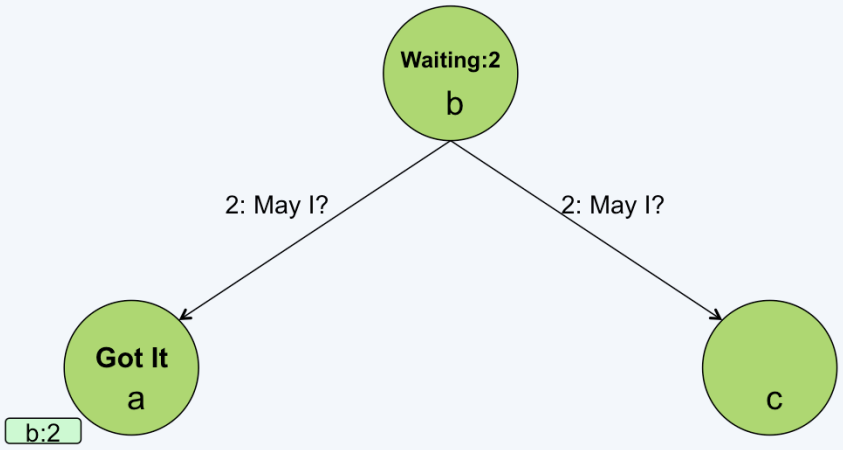
Mutex Exclusion (1)



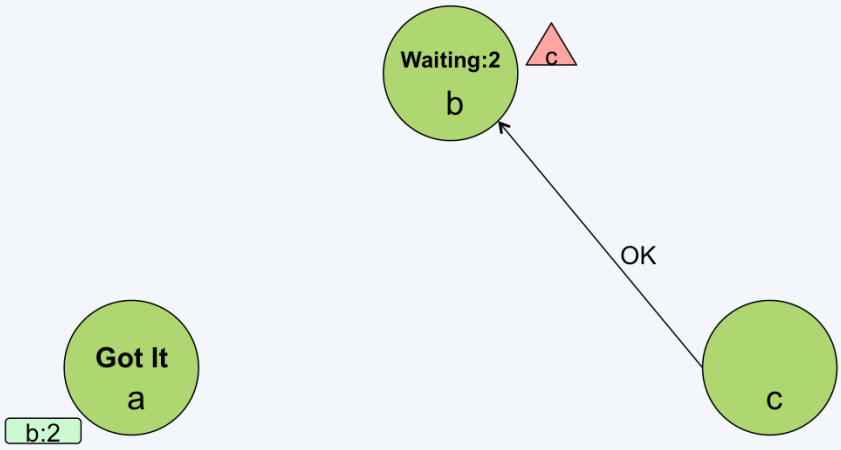
Mutex Exclusion (2)



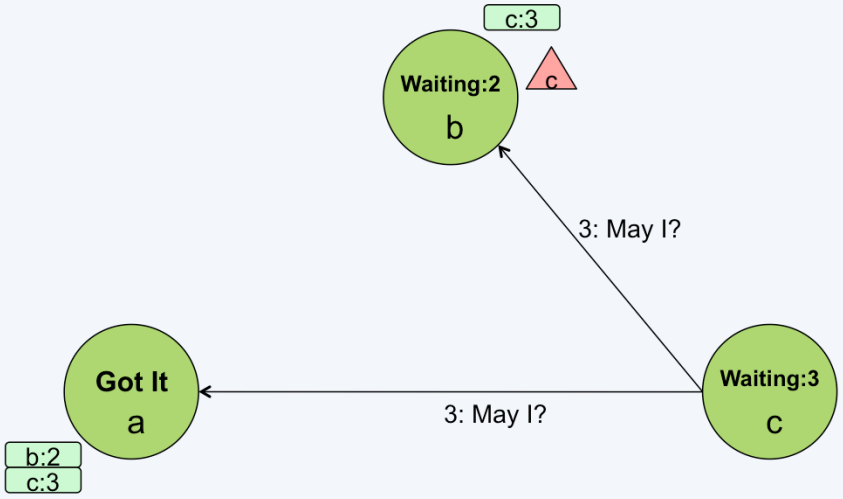
Mutex Exclusion (3)



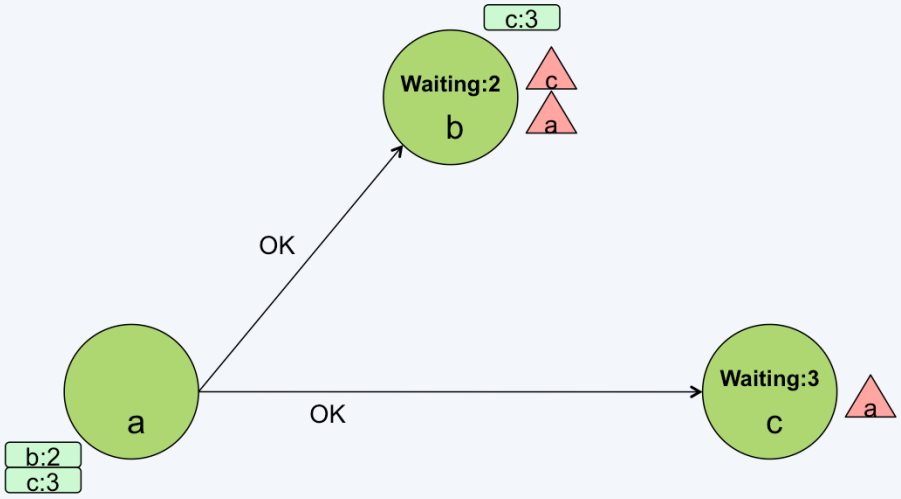
Mutex Exclusion (4)



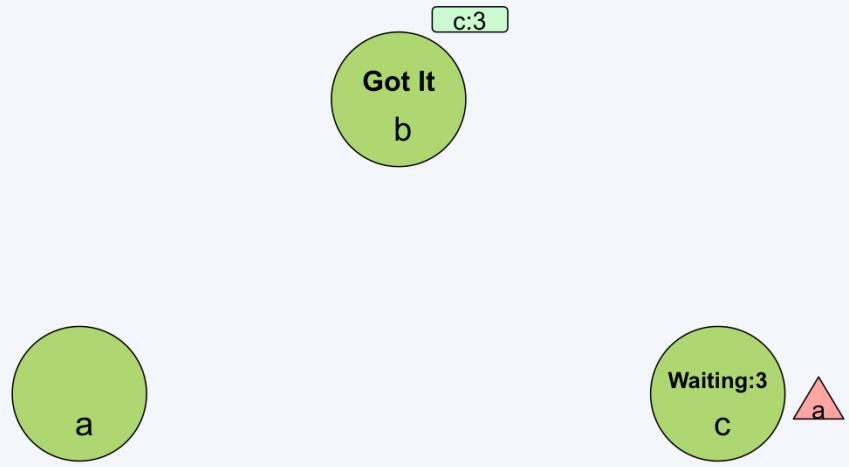
Mutex Exclusion (5)



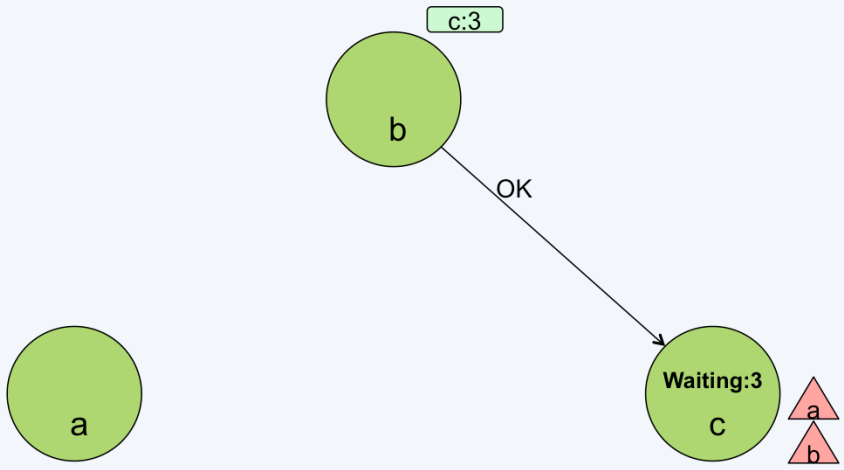
Mutex Exclusion (6)



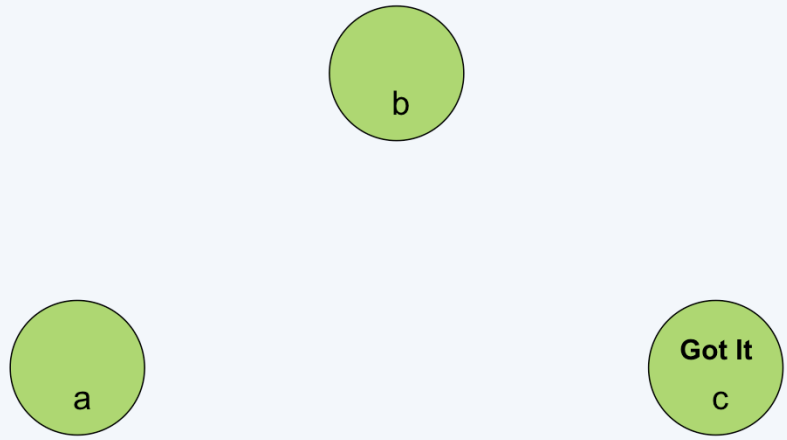
Mutex Exclusion (7)



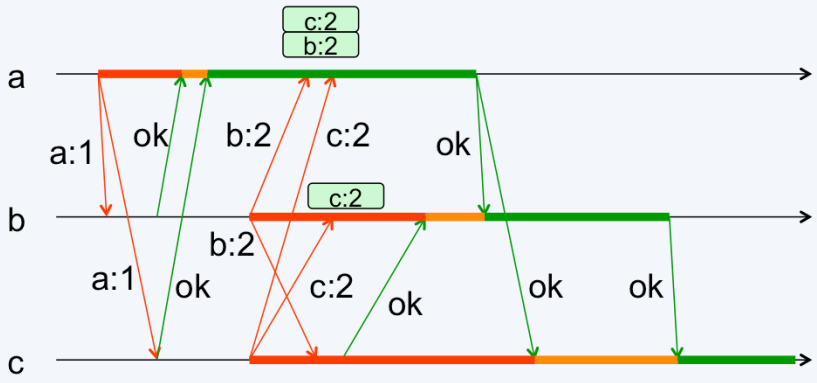
Mutex Exclusion (8)



Mutex Exclusion (9)

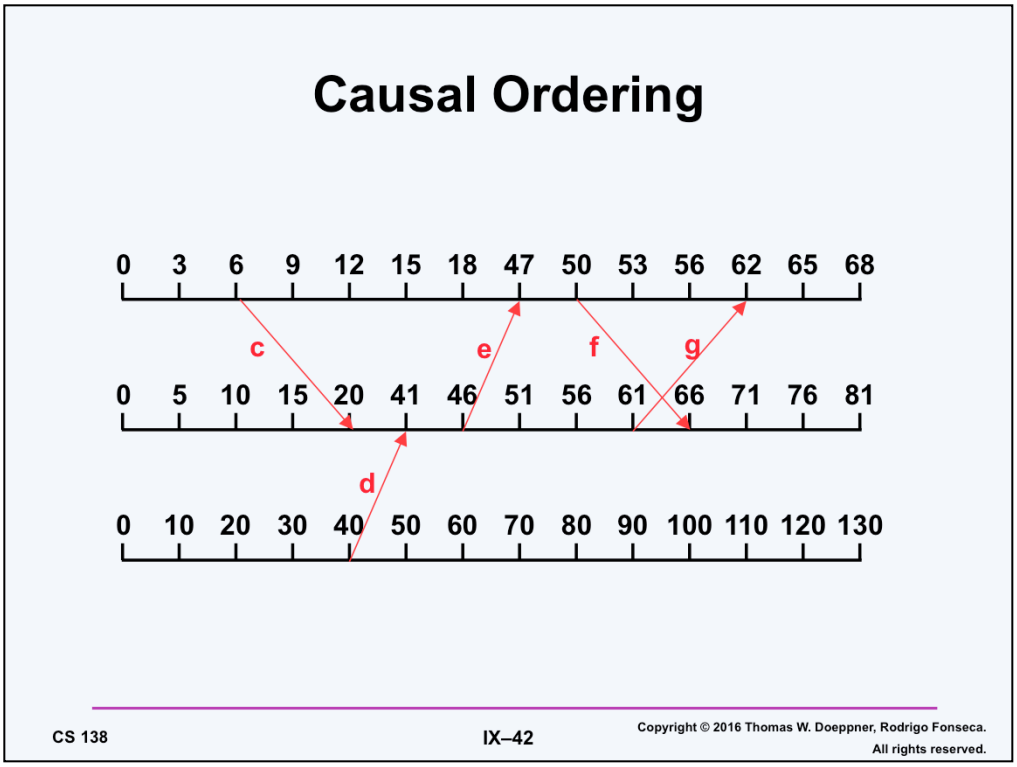


Why Total Order is Important



"if waiting for resource, respond OK if request's timestamp is lower than own, otherwise queue request"

b:2 < c:2



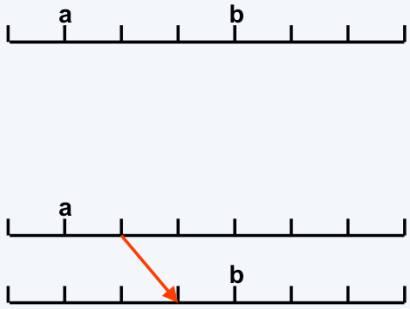
The logical-clocks approach doesn't reflect causality. If m happens before n, then T(m) < T(n).

But the converse is not necessarily true: T(m) < T(n) does not necessarily imply that m happens before n.

In particular, even though $T(c_{receive})<T(d_{send})$, there really is no causality relation between the two events.

Causality

- How can event *a* have a causal effect on *b*?



Concurrent Events

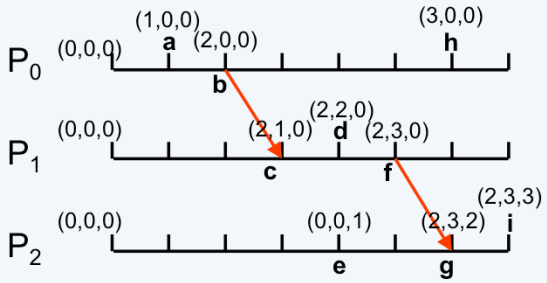
- Events a and b are *concurrent* if neither of the following are true:

  $a \rightarrow b$

  $b \rightarrow a$

Vector Clocks

- $VC_i[i]$
  - number of events so far at process P_i
- $VC_i[h] = k$
  - process P_i is aware of the first k events at P_h



See Coulouris, Dollimore, Kindberg and Blair, page 609, for discussion.

Rules …

1) Initially, $VC_i[j] = 0$, for i, j = 0, … , n-1
2) Just before P_i timestamps an event, it sets $VC_i[i] = VC_i[i]+1$
3) P_i includes the current timestamp in every message it sends
4) When P_i receives a timestamp t in a message, it sets $VC_i[j] = \max(VC_i[j], t[j])$, for j = 0, … , n-1
5) Comparing clocks:
   1) If all components of $VC_i <= VC_j$ and at least one component is smaller ⇔ i happens before j
   2) i and j are concurrent iff neither $VC_i <= VC_j$ and $VC_j <= VC_i$

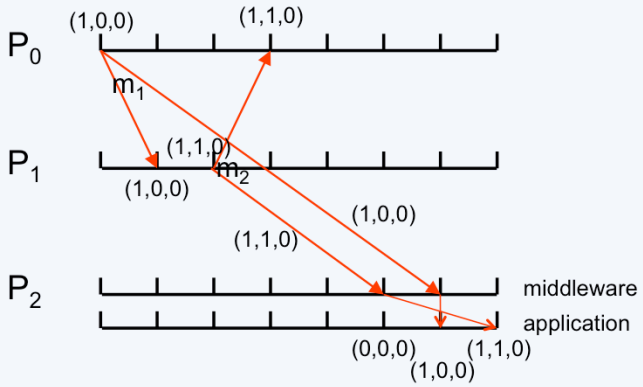
This is from page 609 of the textbook.
Thanks to Alex Kleiman for catching the error on the condition for two events to be concurrent.

Causally Ordered Multicast

- Application of vector clocks
  - the only event is 'sending message'
  - all messages are multicast to all
  - messages carry "timestamp" = sender's VC
- Strategy
  - P_h receives multicast message *m* from P_i
  - deliver *m* to application when:
    - timestamp(*m*)[i] = VC_h[i] + 1
      - next expected message from P_i
    - timestamp(*m*)[k] ≤ VC_h[k], for all k≠i
      - P_h has seen all events P_i had seen when it sent the message

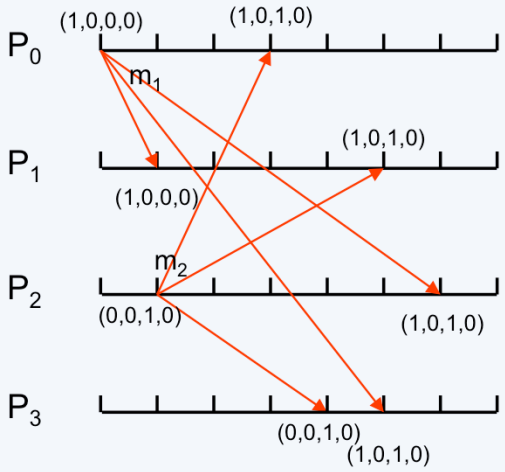
Note that *m*'s timestamp is what P_i's vector clock was when the message was sent.

Causally Ordered Multicast (1)



Here P_0 sends multicast message m_1 to P_1 and P_2. P_1 receives the message first, then sends its own multicast m_2, which arrives at P_2 before m_1. Middleware must delay giving m_2 to the application until it has received m_1.

Causally Ordered Multicast (2)



Here P_0 sends multicast message m_1 to P_1, P_2, and P_3. P_2 independently sends multicast message m_2 to P_0, P_1, and P_3. Since the messages are not causally related, they may be delivered in different orders to P_1 and P_3. Causal ordering is different from total ordering!

Total ordering imposes one total order that is consistent with the clocks.

Causal ordering enables each process to detect whether two messages are causally related or concurrent. If messages m1 and m2 would apply conflicting modifications to an object, for example, the fact that they are concurrent allows the middleware to detect the conflict and ask the user what to do.

Not covered in class

- These won't be in the exams, but are another example of when you can use timestamps

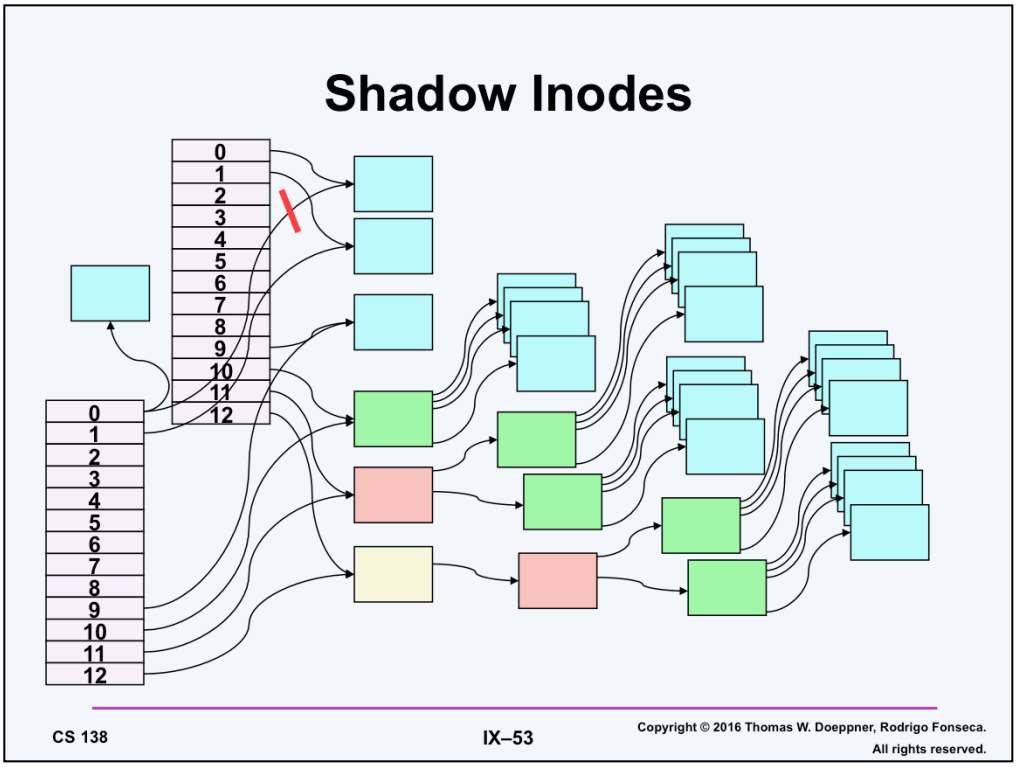
File Transactions

- Open file
- Write first item
  - allocate space
  - update inode
  - write data
- Write second item
  - allocate space
  - update inode
  - write data
- Close file

As an application of logical clocks we look at implementing transactions for a file system. An example is shown on the slide: we are appending a record consisting of two items to the end of a file.

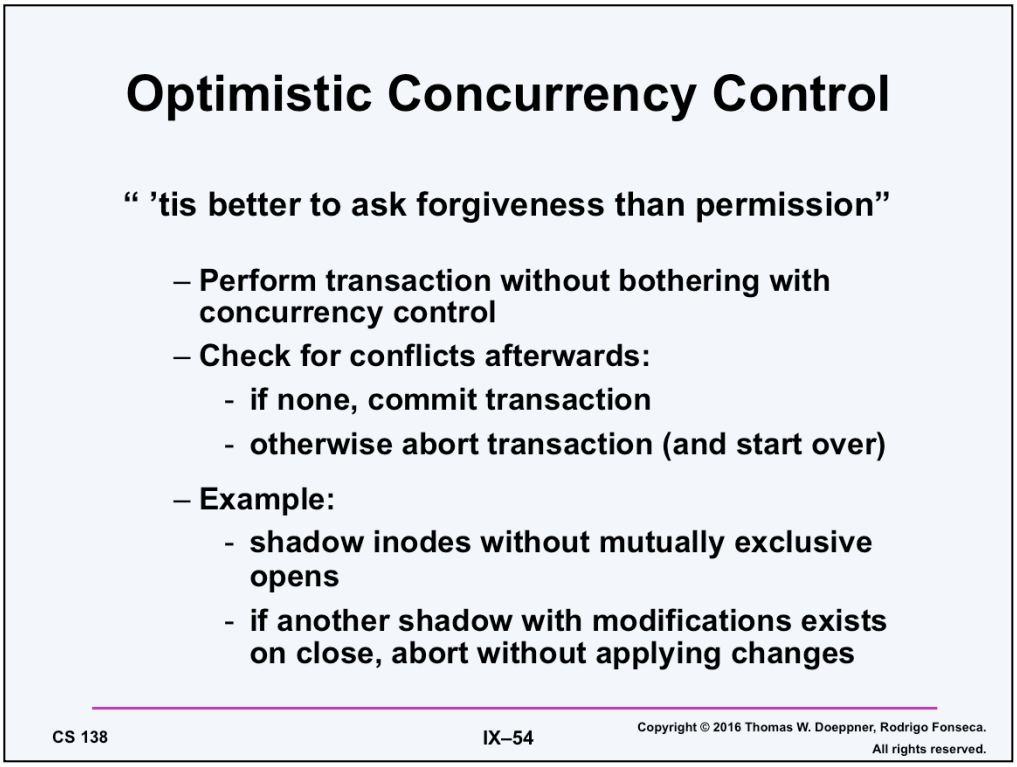
Example Strategies

- Shadow inodes
- Logging
  - Ext3, NTFS
- Optimism



A simple example of the use of atomic transactions with file systems is a technique known as “shadow inodes.” When one opens a file for write (which, necessarily, must be done under mutual exclusion), a copy of the file’s inode is created (in particular, including the disk map), called the *shadow inode*. Operations on the file involve this copy of the inode. If a block of the file is modified in the direct portion of the file, then, much as with copy-on-write techniques, a copy of the original block is made and that copy is modified and linked to the shadow inode. Analogous techniques are used in the indirect portions of the file.

When the file is closed, its inode is replaced with the shadow and all no-longer-referenced blocks are deleted.

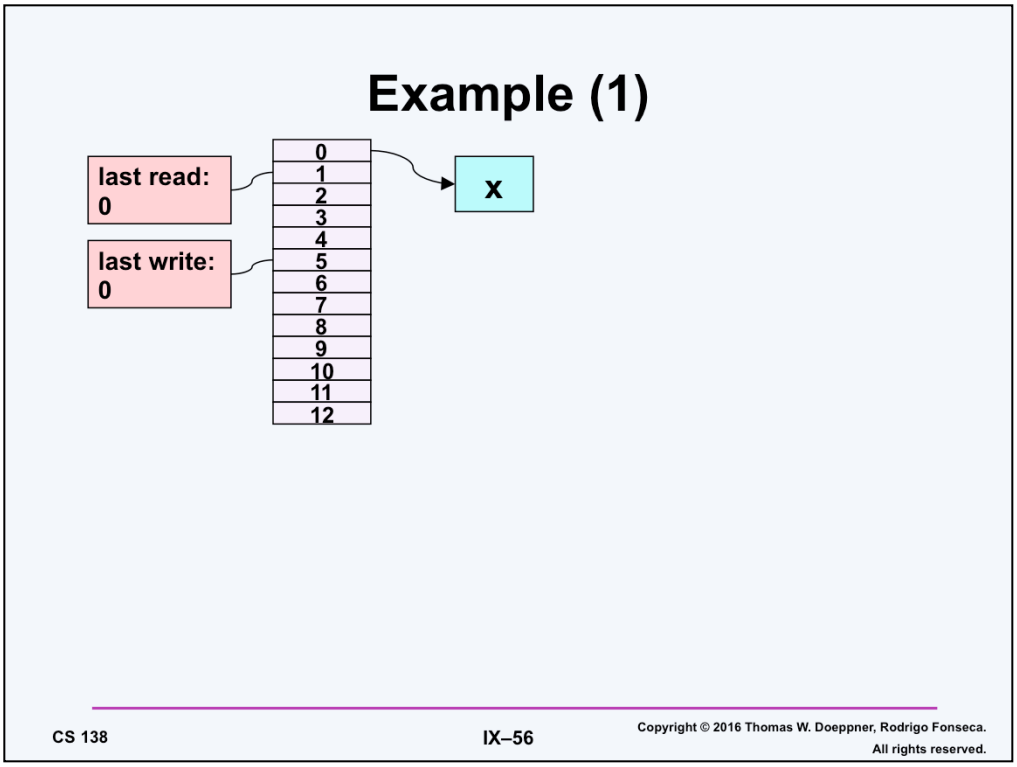


An approach that works quite well for concurrency control when conflicts are rare is *optimistic concurrency control*: forge ahead blindly without worrying about anything until the transaction is about to commit, then worry—at this point check for conflicts and abort if any are found. For example, if this is a transaction involving files, a record is kept of all files that were used in the transaction. When it comes time to commit, a check is made to see if any of these files were involved in other, recently completed transactions. If so, our transaction is aborted, otherwise it commits.

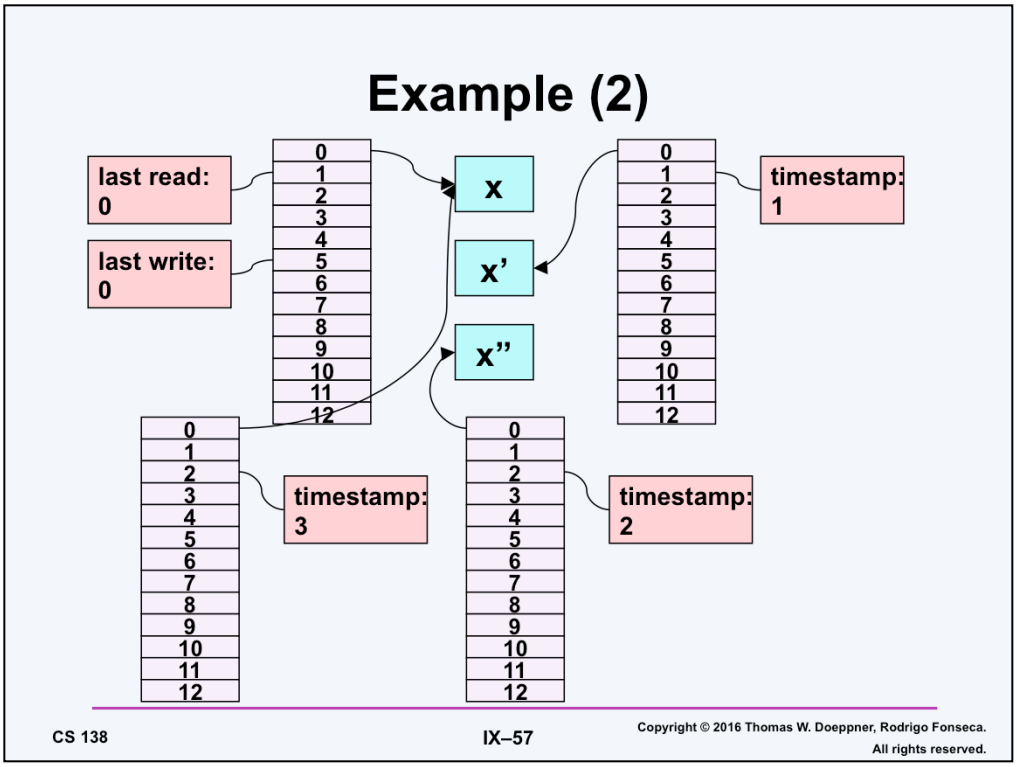
Optimistic Concurrency Control with Timestamps

- Each transaction, when started, is assigned the current timestamp; transactions are ordered by their timestamps
- Each file has last-read and last-write timestamps identifying the last committed transaction that read or wrote it
- If things are correctly ordered, whenever a transaction accesses a file, the file's timestamps will be earlier than the transaction's
- There's a problem if the transaction is writing and the file's read or write timestamp is later than the transaction's, or if the transaction is reading and the file's write timestamp is later than the transaction's
- In case of problem, abort

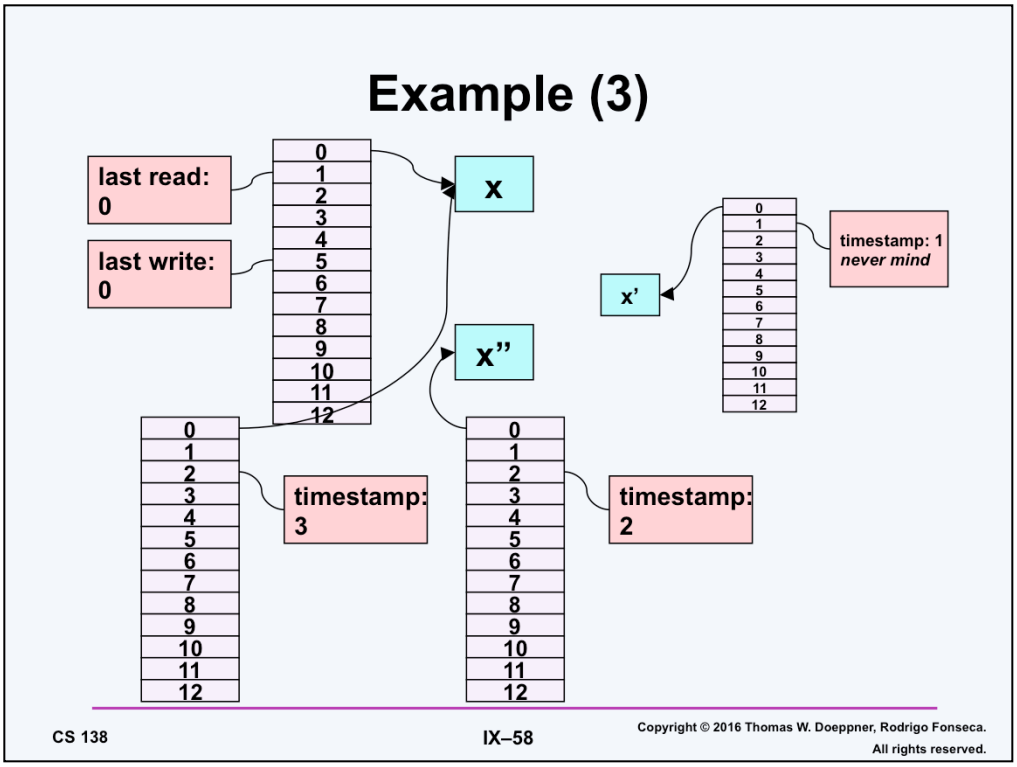
A variation of the optimistic approach takes advantage of logical clocks. This approach has the advantage in that it discovers conflicts somewhat more quickly than that outlined on the previous page.



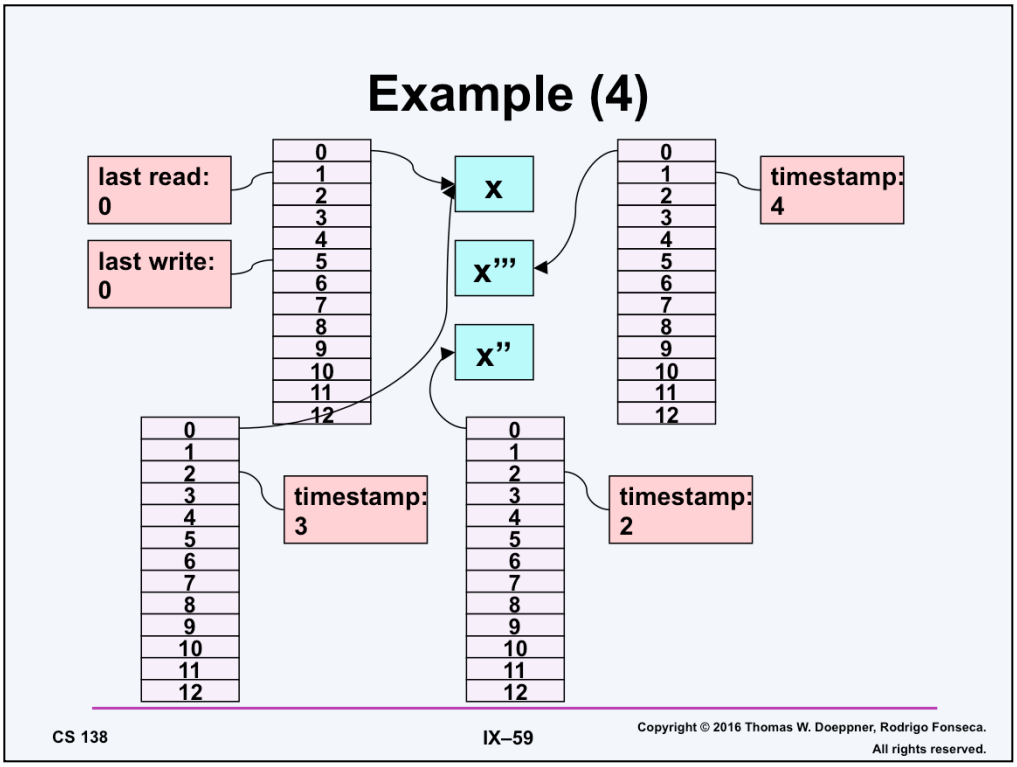
As an example of optimistic concurrency control with timestamps, we again look at the use of shadow inodes. Here we have a file with a single direct block, whose last-read and last-write timestamps are 0.



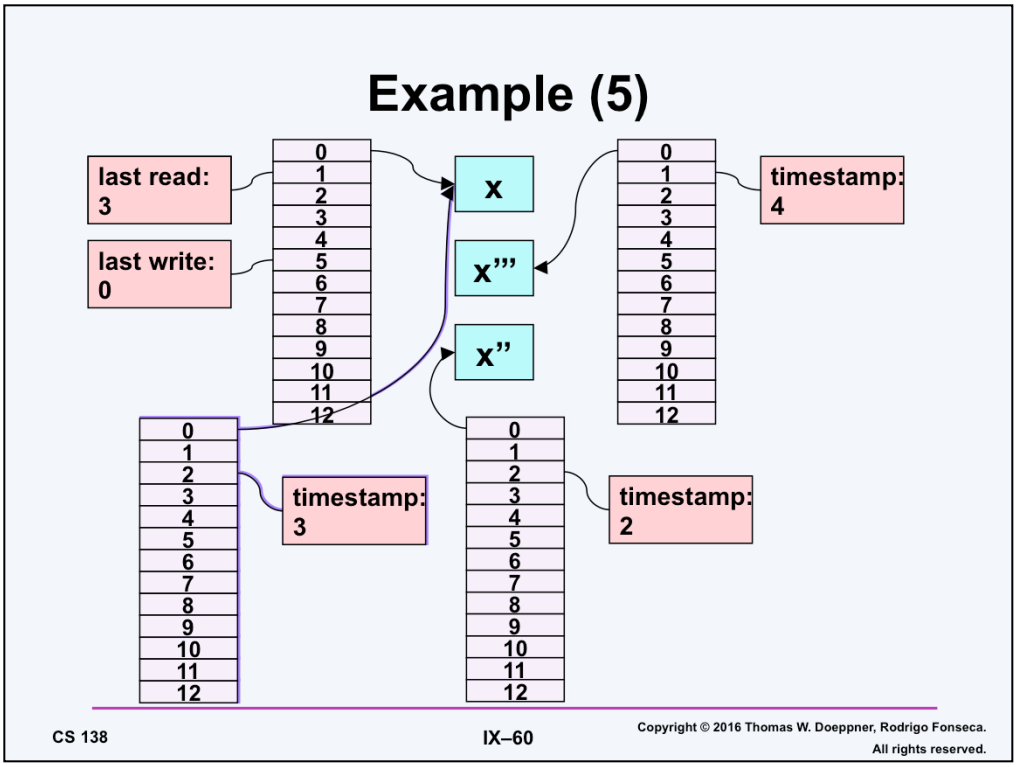
Now two transactions start operating on the file. The first is given a timestamp of 1, the other a timestamp of 2. Both transactions modify the file. Then a third transaction, with timestamp 3 starts on the file, but is merely reading it. Since neither of the write transactions have committed, the read transaction uses data from the timestamp-0 version of the file.



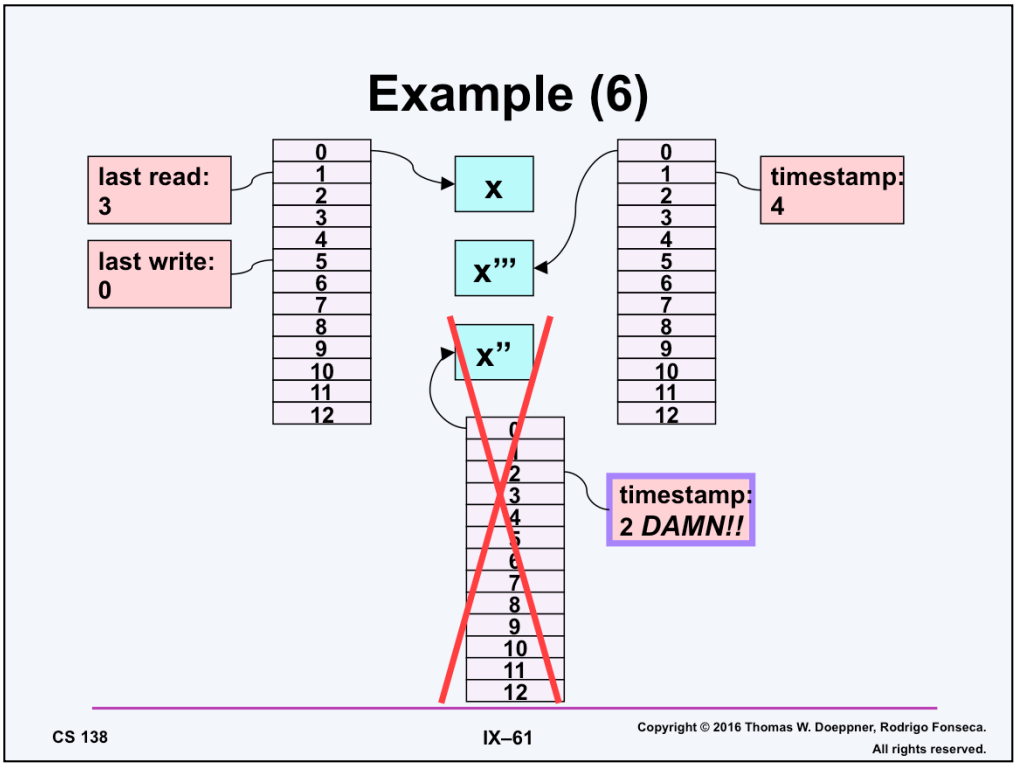
At this point, the first transaction aborts (and thus doesn't commit and doesn't modify the file).



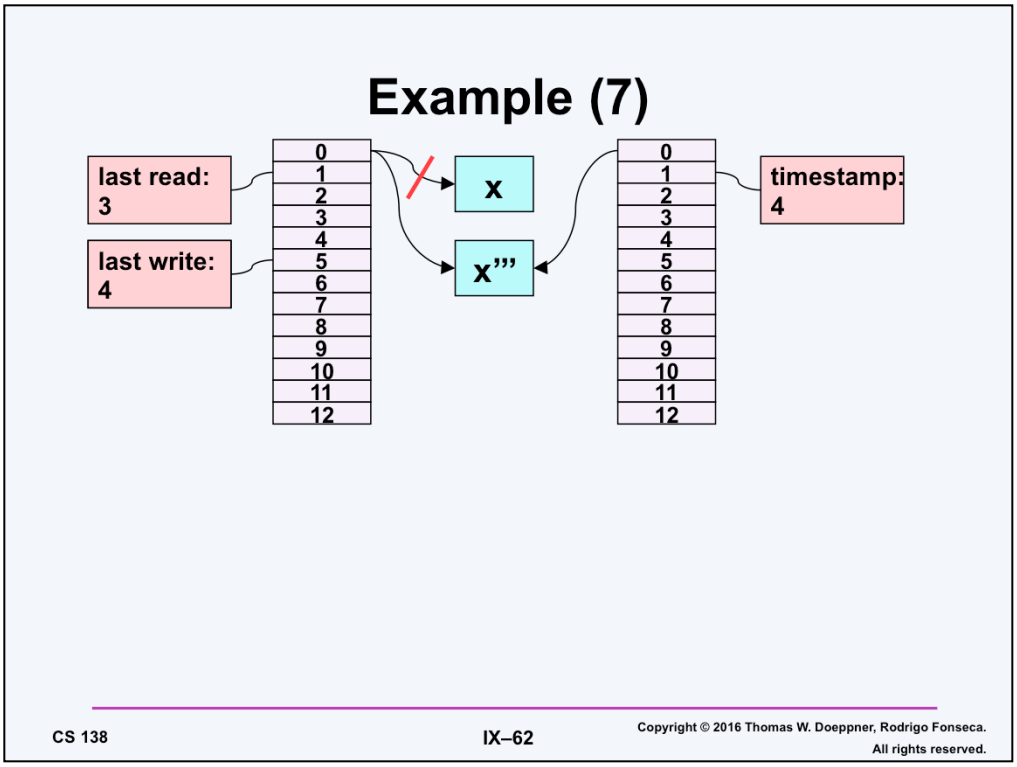
Now another transaction starts up and modifies the file.



Transaction 3 commits and thus sets the last-read timestamp of the file.



Transaction 2 tries to continue but discovers that the file's last-read timestamp is more recent than the transaction, and it thus must abort. If transaction 2 were allowed to commit, then its effect should have happened before transaction 3. However, transaction 3 has read data that was subsequently modified by transaction 2.



Finally, transaction 4 completes. Since its timestamp is more recent than the file's last-read and last-write timestamps, it's allowed to commit.