CS 138: Time
Topics

- Clock Synchronization
- Logical Clocks
- Causality
- Vector Clocks
Internal V. External Synchronized

- External: synchronized with an external server
  - Alice and bob synchronize with the UTC clock
    - $|\text{UTC} \_ \text{time} - \text{Alice} \_ \text{time}| < D$

- Two clock synchronized with each other
  - Alice and bob synchronize with each other
    - $|\text{Alice} \_ \text{time} - \text{Bob} \_ \text{time}| < D$
Getting the Time

What time is it?

[30 seconds later]
You mean … now?
Your Timeline

Yogi’s Timeline

\[ C(t_r) = YW(t_r) \]

\[ t_x \leq YW(y_w) \leq t_r \]

\[ YW(t_r) = [YW(y_w), YW(y_w) + (t_r - t_x)] \]
Using a Time Server

Server’s Timeline

Client’s Timeline

• $S(t_8)$ in $[S(t_4), S(t_4) + (C(t_8) - C(t_1))]$
Variability in Ping Times

![Variability in Ping Times Diagram](image)
Example: NTP

Goal: calculate offset $O$

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

d$= \text{time spent in network} = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) = a - b$;

$o_i = \text{difference in average} = (T_{i-2} + T_{i-1})/2 - (T_i + T_{i-3})/2$

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

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

$b = (a+b)/2 - (a-b)/2 \leq O \leq (a+b)/2 + (a-b)/2 = a$

$o_i - d_i/2 \leq O \leq o_i + d_i/2$
Network Time Protocol

Stratum 0

Stratum 1

Stratum 2
Precision using HW time-stamping
(PTP Protocol)
Nobody’s Perfect

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

\[
t_2 \geq t_1 + \frac{r}{1-d} \quad \Rightarrow \quad T(t_2) \geq T(t_1) + r
\]
Getting Other Opinions

What time is it?

Client

3:31

Server 1

3:30

Server 2

3:29

Server 3
Truth in Advertising

What time is it?

Client

3:31 ± .5 min.

Server 1

3:30 ± 1 min.

Server 2

3:29 ± 2 min.

Server 3
Arriving At a Consensus

Server 1

Server 2

Server 3

Consensus
A Liar

What time is it?

Client

Server 1

3:31 ± .5 min.

Server 2

3:29 ± 2 min.

Server 3

Server 4

3:33 ± .5 min.

3:30 ± 1 min.
What To Do?

Server 1
Server 2
Server 3
Server 4
Consensus
At Most One Liar

Server 1
Server 2
Server 3
Server 4
Consensus
At Most Two Liars

Server 1

Server 2

Server 3

Server 4

1 & 2
2 & 3
1 & 3

Consensus
At Most Three Liars

Server 1

Server 2

Server 3

Server 4

Consensus
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
What’s Time? (Really …)

• 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

• $A \rightarrow B$ iff $A$ could possibly have had an effect on $B$ (pronounced $A$ happened before $B$)
Invariant: if x h.b. y, T(x) < T(y)
Logical Clocks (2)
Using Logical Timestamps

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

- SFO: add interest based on current balance
- PVD: deposit $1000
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
# Totally Ordered Multicast

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: compute interest (1,1)</td>
<td>1: compute interest (1,1)</td>
</tr>
<tr>
<td>4: ack interest-1</td>
<td>3: ack interest-1</td>
</tr>
<tr>
<td>5: deposit $1000 (1,2)</td>
<td>6: ack deposit-1</td>
</tr>
<tr>
<td>7: ack deposit-1</td>
<td>8: ack deposit-2</td>
</tr>
<tr>
<td>8: ack deposit-2</td>
<td>9: ack interest-2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Out</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: deposit $1000 (1,2)</td>
<td>2: deposit $1000 (1,2)</td>
</tr>
<tr>
<td>3: ack deposit-2</td>
<td>4: ack deposit-2</td>
</tr>
<tr>
<td>5: compute interest (1,1)</td>
<td>6: ack interest-2</td>
</tr>
<tr>
<td>7: ack interest-2</td>
<td>8: ack interest-1</td>
</tr>
<tr>
<td>9: ack deposit-1</td>
<td></td>
</tr>
</tbody>
</table>

PVD must reorder queue once all acks are in.
Mutual Exclusion

• Central server
• Logical clocks
Central-Server Mutual Exclusion

Smart Object

May I?

a

b

May I?

Smart Object

May I?

c

May I?
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
Mutex Exclusion (1)

a \rightarrow b \rightarrow c

1: May I?
Mutex Exclusion (2)
Mutex Exclusion (3)

Waiting: 2

2: May I?

Got It
a

b: 2

b

2: May I?

C
Mutex Exclusion (4)
Mutex Exclusion (5)
 Mutex Exclusion (6)
Mutex Exclusion (7)

Got It

b

c:3

Waiting: 3

a

a

Copyright © 2016 Thomas W. Doeppner, Rodrigo Fonseca. All rights reserved.
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
Causal Ordering

c

d

e

f

g
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 $P_h$
Rules …

1) Initially, $VC_i[j] = 0$, for $i, j = 0, \ldots, n-1$

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 $j = 0, \ldots, n-1$

5) Comparing clocks:
   1) If all components of $VC_i \leq VC_j$ and at least one component is smaller $\iff$ $i$ happens before $j$
   2) $i$ and $j$ are concurrent iff neither $VC_i \leq VC_j$ and $VC_j \leq VC_i$
Causally Ordered Multicast

• Application of vector clocks
  – the only event is ‘sending message’
  – all messages are multicast to all

• Strategy
  – $P_h$ receives multicast message $m$ from $P_i$
  – deliver $m$ to application when:
    - $\text{timestamp}(m)[i] = \text{VC}_h[i] + 1$
      • next expected message from $P_i$
    - $\text{timestamp}(m)[k] \leq \text{VC}_h[k]$, for all $k \neq i$
      • $P_h$ has seen all events $P_i$ had seen when it sent the message
Causally Ordered Multicast (1)
Causally Ordered Multicast (2)
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
Example Strategies

- Shadow inodes
- Logging
  - Ext3, NTFS
- Optimism
Shadow Inodes
Optimistic Concurrency Control

“’tis better to ask forgiveness than permission”

– Perform transaction without bothering with concurrency control
– Check for conflicts afterwards:
  - if none, commit transaction
  - otherwise abort transaction (and start over)
– Example:
  - shadow inodes without mutually exclusive opens
  - if another shadow with modifications exists on close, abort without applying changes
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.
Example (1)

last read: 0

last write: 0

0
1
2
3
4
5
6
7
8
9
10
11
12

x
Example (2)

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```

```
last read: 0
last write: 0
```
Example (3)

last read: 0

last write: 0

0
1
2
3
4
5
6
7
8
9
10
11
12

x

x'

x''

0
1
2
3
4
5
6
7
8
9
10
11
12

timestamp: 1
never mind

timestamp: 2

timestamp: 3
Example (4)

last read: 0

last write: 0

timestamp: 3

timestamp: 2

timestamp: 4

x

x’’

x’’’
Example (5)

last read: 3

last write: 0

timestamp: 3

x

x””

x”

timestamp: 4

timestamp: 2

timestamp: 3
Example (6)

last read: 3
last write: 0

x

x''

x'''

0 1 2 3 4 5 6 7 8 9 10 11 12

0 1 2 3 4 5 6 7 8 9 10 11 12
timestamp: 4

0 1 2 3 4 5 6 7 8 9 10 11 12
timestamp: 2
DAMN!!
Example (7)

last read: 3

last write: 4

timestamp: 4

last write: 4