Let $x$ and $y$ be two statement executions (aka events)

Define $x$ causally precedes $y$ if

- $x$ and $y$ happened in that order in the same system, or
- $x$ sent a message that $y$ received, or
- transitive closure of above

Causal precedence is a partial order

- if $x$ and $y$ not causally related, no system can determine which happened first (without other interaction or real-time clocks)

Timestamp mechanism extends partial order to total order for a specified set of events
Outline

- Timestamp mechanism
  - Distributed ordering of conflicting requests
  - Distributed lock program: algorithm level
  - Distributed lock program: overview
  - Cyclic timestamps
Each system $j$ has
- integer “clock” $clk$, initially 0

When $j$ does an event $x$ to be ordered:
- increment $clk$, broadcast $[x, clk, j]$

- $clk$ value: timestamp (ts) of $x$
- $[clk$ value, id]: extended timestamp (ets) of $x$

When $j$ receives msg $[y, t, k]$:
- $clk \leftarrow \max(t, clk) + 1$

- $x.ets < y.ets$: ($x.ts < y.ts$) or ($x.ts = y.ts$ and $x.id < y.id$)
- Event $x$ ordered before event $y$ if $x.ets < y.ets$

For most applications, need acks to timestamps
Each system $j$ has
- integer clock $clk$, initially 0
- $rts_k$, $k \neq j$, initially 0 // last ts rcvd from $k$

let $\alpha Rts$: $\min([rts_k, k] : k \neq j)$ // no new ets $< \alpha Rts$

When $j$ does an event $x$ to be ordered:
- increment $clk$, broadcast $[x, clk, j]$

When $j$ receives msg $[y, t, k]$:
- $clk \leftarrow \max(t, clk) + 1$
- $rts_k \leftarrow t$, send $[ack, clk, j]$ to $k$

When $j$ receives msg $[ack, t, k]$:
- $clk \leftarrow \max(t, clk)$, $rts_k \leftarrow t$
Prop erties

- Define auxiliary quantities
  - $\text{hst}$: seq of all ets's in ets-order  
    // initially $[[0, 0]]$
  - $j.\text{hst}$: seq of ets's seen by $j$ in ets-order  
    // initially $[[0, 0]]$
  - $j.\alpha\text{hst}$: prefix of $j.\text{hst}$ of ets’s $\leq j.\alpha\text{Rts}$

- Safety properties
  - $\text{Inv } j.\text{hst}$ subsequence-of $\text{hst}$
  - $\text{Inv } j.\alpha\text{hst}$ prefix-of $\text{hst}$

- Progress properties (assuming no system stops rcving)
  - $j.\alpha\text{Rts} = z < \text{hst.last}$ leads-to $j.\alpha\text{Rts} > z$
  - $j.\alpha\text{hst.size} = z < \text{hst.size}$ leads-to $j.\alpha\text{hst.size} > z$
Outline

- Timestamp mechanism
- Distributed ordering of conflicting requests
- Distributed lock program: algorithm level
- Distributed lock program: overview
- Cyclic timestamps
Request ordering problem

- Collection of systems attached to a fifo channel
- Users issue conflictable requests to the systems
- Each system should serve its requests so that conflicting requests are not served simultaneously (even by different systems)

Some special cases of the problem
- distributed lock
  - every two requests conflict
- distributed readers-writers lock
  - classify requests into reads and writes
  - write conflicts with every other request
Solution overview

- System \( j \) augments the \( ts \) mechanism as follows

- Maintain variable \( \text{req} \): set of “ongoing” req-ets tuples

- Upon local request \( x \): assign ts, add \([x, ts, j]\) to \( \text{req} \)

- Upon rcving \([y, t, k]\): process ts, add \([y, t, k]\) to \( \text{req} \)

- Serve \([x, t, j]\) in \( \text{req} \) when:
  - \([t, j] < \alpha Rts \) and
  - \([t, j] < [u, k]\) for every conflicting \([y, u, k]\) in \( \text{req} \)

- After serving \([x, t, j]\): remove it from \( \text{req} \), bcast \([\text{REL, } x, t, j]\)

- Upon rcving \([\text{REL, } y, t, k]\): remove \([y, t, k]\) from \( \text{req} \)
Solution: variables and messages

- **System j variables**
  - clk: initially 0 // clock
  - rts_k: initially 0 // highest ts rcvd from k
  - αRts: min(rts_k, k]: k ≠ j) // min ets induced by rts
  - req: initially empty // set of outstanding requests-ets

- **Messages**
  - [REQ,x,t,k] // request msg
  - [ACK,t,k] // ack msg
  - [REL,x,t,k] // release msg
Solution: system j rules – 1

- User issues request x
  - clk++
  - send [REQ,x,clk,j] to every system
  - add [x,clk,j] to req

- Receive [REQ,x,t,k]:
  - clk ← max(clk, t+1)
  - rts[k] ← t
  - send [ACK,clk,j] to k // omit if ets > [t,k] already sent to k
  - add [x,t,k] to req

- Start serving request [x,t,j] in req when
  - [t,j] ≤ αRts
  - for every [y,s,k] in req st x conflicts with y: [t,j] ≤ [s,k]
Solution: system j rules – 2

- Finish serving request \([x,t,j]\):
  - remove \([x,t,j]\) from req;
  - send \([\text{REL},x,t,j]\) to every other system.

- Receive \([\text{ACK},t,k]\):
  - \(\text{clk} \leftarrow \max(\text{clk}, t); \text{rts}[k] \leftarrow t\)

- Receive \([\text{REL},x,t,k]\):
  - remove \([x,t,k]\) from req

- atomicity assumption: rules are atomic

- progress assumption: weak fairness
Outline

Timestamp mechanism
Distributed ordering of conflicting requests
Distributed lock program: algorithm level
Distributed lock program: overview
Cyclic timestamps
Distributed lock: algorithm level

- Distributed program that implements a distributed lock
- Collection of systems attached to a fifo channel
- Specialize the request-ordering solution for a lock
- At most one ongoing request per system
  - so each system is thinking, hungry, or eating
  - no need for ts in release msg
- Later, refine to await program implementing dist lock service
Solution: variables, functions, messages

- **System j variables**
  - `clk`, `{rts_k}`
  - `req ← []`  // as in req-ordering
  // map indexed by address
  // entry [j,t] ↔ ets [t,j]

- **System j functions**
  - `αRts`  // min ets in `{rts_k}`
  - `αReq: min([req[k], k]: k in req.keys)`  // min ets in req

- **Messages**
  - `[REQ,t,k], [ACK,t,k], [REL,k]`  // as in req-ordering
  // note: no “x” field
Solution: system j rules

- Become hungry only if thinking
  - clk++
  - send [REQ, clk, j] to every system
  - req[j] ← clk

- Become eating only if hungry and [req_j, j] = αReq ≤ αRts

- Become thinking only if eating:
  - remove entry for j from req
  - send [REL, j] to every system
Solution: system j rules — 2

- Receive [REQ,t,k]:
  - clk ← max(clk, t+1)
  - rts[k] ← t
  - req[k] ← t
  - send [ACK,clk,j] to k

- Receive [ACK,t,k]:
  - clk ← max(clk, t); rts[k] ← t

- Receive [REL,k]:
  - remove entry for k from req

- atomicity assumption: rules are atomic

- progress assumption: weak fairness
Goal: \( \text{Inv} \) at most one system is eating

- \( \text{Inv} A_1-A_4 \) holds, where
  
  \( A_1 : \) \((\llbracket[j,s]\in k.\text{req}\) and \(j \neq k\) \(\Rightarrow\)
  
  \((\llbracket[j,s]\in j.\text{req}\) or \(\llbracket\text{REL},j\in\text{transit to} k\))))

  \( A_2 : \) \((j \text{ eating}) \Rightarrow \llbracket j.\text{req}[j],j\rrbracket = j.\alpha\text{Req} \leq j.\alpha\text{Rts}\)

  \( A_3 : \) \((j \text{ eating}) \) and \((k \text{ eating})\) \(\Rightarrow\) \(j = k\)

  \( A_4 : \) \((j \text{ hungry}) \) and \(\llbracket j.\text{req}[j],j\rrbracket = j.\alpha\text{Req} \leq j.\alpha\text{Rts}\)
  \(\Rightarrow\) (no one eating)

- \( \text{Inv} A_3 \) implies desired property
Analysis: progress

Goal: (wfair, bounded eating, ongoing rx, channel progress)

⇒ j hungry  \textit{leads-to}  j eating

Define

- \( \text{hst} \): seq of all ets’s in ets-order // initially \([0,0]\)
- \( \text{ne} \): # requests that have finished eating

Proof

if \([j,s]\) is in j.req, eventually \([s,j]\) ≤ j.\(\alpha\)Rts holds

after this point

- \([j,s]\)’s index in \(\text{hst}\) is fixed, at say \(n\)
- entries in \(\text{hst}[\text{ne}+1..n]\) eat in order

[entry \(\text{ne}\)’s release msg is incoming to entry \(\text{ne}+1\)’s system. when it arrives, the latter eventually becomes eating]
Outline

Timestamp mechanism
Distributed ordering of conflicting requests
Distributed lock program: algorithm level
Distributed lock program: overview
Cyclic timestamps
Distributed lock: await-based

Distributed program: implements distributed lock service
- starts a fifo channel
- starts a LockTs system at each address

LockTs: await program, refines algorithm-level system
- input functions acq and rel // called by lock users
- output calls to tx and rx of channel access system
- one local thread to execute rx
- multiple acq calls can be ongoing
  but only one participates in ts mechanism
program LockTsDist(ADDR)
    \{c_j\} \leftarrow \text{start}(\text{FifoChannel}(ADDR))
    \text{for } j \text{ in } ADDR
        v_j \leftarrow \text{start}(\text{LockTs}(ADDR, j, c_j))
    \text{return } \{v_j\}
Program LockTs (ADDR, j, cj) = 1

dist lock: await program

Main

- clk ← 0
- rts_k ← 0, k in ADDR-{j}
- req
- startThread (doRx())

input mysid.acq()

- await (not (j in req.keys))  // a1
  - clk++ ; req_j ← clk
  - for k in ADDR-{j}
    - cj.tx(k, [REQ, clk, j])
- await ([req_j, j] ≤ αReq and  // a2
           (ADDR.size=1 or [req_j, j] ≤ αRts))
- return
Program LockTs (ADDR, j, cj) = 2

- input mysid.rel()
- await (true)
  - req.remove(j)
  - for k in ADDR−{j}
    c.j.tx(k, [REL, j])

- function doRx() // executed by a local thread
  while true
  - msg ← c.j.rx()
    ia {msg is [REQ, t, k], [ACK, t, k], or [REL, k]}
    await true
    do appropriate rx-msg action

- atomicity assumption {awaits}
- progress assumption {wfair threads, sfair await a1}
Map alg-level state to await-program state

<table>
<thead>
<tr>
<th>alg-level</th>
<th>await-program</th>
</tr>
</thead>
<tbody>
<tr>
<td>j hungry</td>
<td>thread in j.acq.a2</td>
</tr>
<tr>
<td>j eating</td>
<td>[j,..] in j.req, no thread in j.acq.a2</td>
</tr>
<tr>
<td>j thinking</td>
<td>no [j,..] in j.req</td>
</tr>
</tbody>
</table>

Show that alg-level properties are preserved (⋆)

Prove: LockTsDist(ADDR) implements DistLockService(ADDR)
  - define program of implementation and service inverse
  - identify effective atomicity breakpoints
  - obtain assertions
  - prove program satisfies assertions // easy given ⋆
Outline

- Timestamp mechanism
- Distributed ordering of conflicting requests
- Distributed lock program: algorithm level
- Distributed lock program: overview
- Cyclic timestamps
Using cyclic timestamps

- Goal: cyclic timestamps in the distributed lock solution
- Easily achieved by modifying solution slightly

- Existing solution: request $[t,j]$ eats when
  1. $[t,j] = j.α_{Req}$
  2. $[t,j] \leq j.α_{Rts}$

- Impose additional requirement:
  3. $j$ eats only after receiving ack from every system

- Resulting simplification
  - ack’s ts always higher than request’s ts
    - so no need for ack’s ts
    - no need for $\{rts_k\}$
  - sufficient to track # acks rcvd
  - no need for ack’s sender id
Solution: variables, functions, messages

- **System j variables**
  - clk, \{rts_k\}, req
  - na // # acks due

- **System j functions**
  - αRts, αReq

- **Messages**
  - [REQ,t,k], [ACK,t,k], [REL,k]
Solution: system $j$ rules

- Become hungry only if thinking
  - $\text{clk}++$
  - $\text{req}[j] \leftarrow \text{clk}$
  - send $[\text{REQ}, \text{clk}, j]$ to every system
  - $\text{na} \leftarrow 0$

- Become eating only if hungry and
  $[\text{req}_j, j] = \alpha \text{Req}$ and $\text{na} = \text{ADDR.size} - 1$

- Become thinking only if eating:
  - remove entry for $j$ from req
  - send $[\text{REL}, j]$ to every system
Solution: system j rules – 2

- Receive [REQ,t,k]:
  - clk ← max(clk, t+1)
  - req[k] ← t
  - send [ACK] to k

- Receive [ACK]:
  - na++

- Receive [REL,k]:
  - remove entry for k from req

atomicity assumption: rules are atomic
progress assumption: weak fairness
Analysis: conventions

Abbreviations for readability:

- \texttt{hst}.ts to mean \texttt{hst[j][0]}
- \texttt{N} to mean ADDR.size

Define

- \texttt{hst}: seq of all ets’s, initially \texttt{[[0,0]]} // as before
- \texttt{ne}: # releases globally, initially 0 // as before
- \texttt{j.ne}: # releases seen by \texttt{j} // may lag \texttt{ne}
- \texttt{tse}: ts of last request to release // \texttt{hst.ne.ts}
- \texttt{j.tse}: ts of last request released at \texttt{j} // \texttt{hst_n.e.ts}
Analysis: overview

- Prove: any ts in transit is in $t_se .. t_se + 2N$
- Prove: $j.t_se$ is in $t_se - 2N .. t_se$
- Hence: any ts in transit is in $j.t_se .. j.t_se + 4N$
- Hence can use modulo-$M$ ts, for $M \geq 4N$

- Modify system $j$ to use cyclic ts
  - add variable $t_se$, initially 0
  - when a request msg is sent, set its ts to $\text{mod}(clk, M)$
  - when a request msg $[\text{REQ}, ct, j]$ is rcvd, treat $ct$ as unbounded ts $t_se + \text{mod}(ct - t_se, M)$
  - when $[k, t]$ is removed from req, set $t_se$ to $t$
Analysis: bound ts wrt tse

cyclic timestamps

■ Following are invariant

\( C_1 : ([\text{REQ}, t, j] \text{ rcvable}) \Rightarrow \overline{\text{hst}}_{\text{ne}}.ts \leq t \leq \overline{\text{hst}}.\text{last}.ts \)

\( C_2 : \text{forsome}(x \in \overline{\text{hst}}: \ x.ts \leq i.\text{clk} \leq x.ts+1) \)

\( C_3 : \overline{\text{hstp}}.ts \leq \overline{\text{hstp}}_1.ts \leq \overline{\text{hstp}}.ts+2 \)

\( C_4 : ([\text{REQ}, t, j] \text{ rcvable}) \Rightarrow \overline{\text{hst}}_{\text{ne}}.ts \leq t \leq \overline{\text{hst}}_{\text{ne}}.ts + 2 \times \text{ADDR}.\text{size} \)

■ Inv \( C_1 : [\text{REQ}, t, j] \text{ in transit implies req } [t, j] \text{ hungry} \)

■ Inv \( C_2 : C_2 \text{ satisfies invariance rule} \)

■ Inv \( C_3 : C_3 \text{ satisfies invariance rule assuming Inv } C_2 \)

■ Inv \( C_4 : \text{follows from Inv } C_1, C_3 \)
Following are invariant

\( C_5 \): (\# REL msgs incoming to j) < ADDR.size

\( C_6 \): \( \overline{ne} - j.ne = (\# REL msgs incoming to j) \)

\( C_7 \): \( \overline{hst_{\overline{ne}}.ts} - 2 \times ADDR.size \leq \overline{hst_{j.ne}} \leq \overline{hst_{\overline{ne}}.ts} \)

\( C_8 \): ([REQ,t,j] rcvable at i) \( \Rightarrow \)
\[ \overline{hst_{j.ne}.ts} \leq t \leq \overline{hst_{i.ne}.ts} + 4 \times ADDR.size \]

**Inv C5**: \([t,k]\) is acked only after k’s previous REL msgs are rcvd

**Inv C6**: \( C_6 \) satisfies invariance rule

**Inv C7**: \( C_7 \) implied by \( C_6, C_5, C_3 \)

**Inv C8**: \( C_8 \) implied by \( C_7, C_4 \)