Distributed Solutions using Timestamps

Shankar

April 25, 2014

Event ordering in a distributed system

- 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 without acks

- 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

Timestamp mechanism (with acks)

- Each system j has
  - integer clock clk, initially 0
  - rts_k, $k \neq j$, initially 0 // last ts rcvd from k

- let αRts: $\min([rts_k, k] : k \neq j)$ // 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 $[\text{ack}, clk, j]$ to k

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

Properties

- Define auxiliary quantities
  - $\overline{hst}$: seq of all ets's in ets-order // initially $[\,[0,0]\,]$
  - $j.\overline{hst}$: seq of ets's seen by j in ets-order // initially $[\,[0,0]\,]$
  - $j.\alpha\overline{hst}$: prefix of $j.\overline{hst}$ of ets's $\leq j.\alpha Rts$
- Safety properties
  - Inv $j.\overline{hst}$ subsequence-of $\overline{hst}$
  - Inv $j.\alpha\overline{hst}$ prefix-of $\overline{hst}$
- Progress properties (assuming no system stops rcving)
  - $j.\alpha Rts = z < \overline{hst}.last$ $\;\mathit{leads\text{-}to}\;$ $j.\alpha Rts > z$
  - $j.\alpha\overline{hst}.size = z < \overline{hst}.size$ $\;\mathit{leads\text{-}to}\;$ $j.\alpha\overline{hst}.size > z$

Outline

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 req: set of "ongoing" req-ets tuples
- Upon local request x: assign ts, add [x,ts,j] to req
- Upon rcving [y,t,k]: process ts, add [y,t,k] to req
- Serve [x,t,j] in req when:
  - [t,j] < αRts and
  - [t,j] < [u,k] for every conflicting [y,u,k] in req
- After serving [x,t,j]: remove it from req, bcast [REL,x,t,j]
- Upon rcving [REL,y,t,k]: remove [y,t,k] from 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]: $k \neq j$) // 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 isues request x
  - clk++
  - send [REQ,x,clk,j] to every system
  - add [x,clk,j] to req
- Receive [REQ,x,t,k]:
  - clk $\leftarrow$ max(clk, t+1)
  - rts[k] $\leftarrow$ 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] $\leq \alpha$Rts
  - for every [y,s,k] in req st x conflicts with y: [t,j] $\leq$ [s,k]

Solution: system j rules – 2

- Finish serving request [x,t,j]:
  - remove [x,t,j] from req;
  - send [REL,x,t,j] to every other system.
- Receive [ACK,t,k]:
  - clk $\leftarrow$ max(clk, t); rts[k] $\leftarrow$ t
- Receive [REL,x,t,k]:
  - remove [x,t,k] from req
- atomicity assumption: rules are atomic
- progress assumption: weak fairness

Outline

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`$_k\}$ // as in req-ordering
  - `req` $\leftarrow$ `[]` // map indexed by address
    // entry `[j,t]` $\leftrightarrow$ ets `[t,j]`

- System `j` functions
  - α`Rts` // min ets in $\{$`rts`$_k\}$
  - α`Req`: `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 – 1

- Become hungry only if thinking
  - `clk++`
  - send `[REQ,clk,j]` to every system
  - `req[j]` $\leftarrow$ `clk`
- Become eating only if hungry and $[\mathrm{req}_{\mathtt{j}}, \mathtt{j}] = \alpha\mathrm{Req} \leq \alpha\mathrm{Rts}$
- 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

Analysis: safety

Goal: Inv at most one system is eating

- Inv A_1–A_4 holds, where

  A_1 : (([j,s] in k.req) and j $\neq$ k) $\Rightarrow$
  ([j,s] in j.req) or ([REL,j] in transit to k)

  A_2 : (j eating) $\Rightarrow$ [j.req[j], j] $=$ j.αReq $\leq$ j.αRts

  A_3 : ((j eating) and (k eating)) $\Rightarrow$ j $=$ k

  A_4 : ((j hungry) and [j.req[j], j] $=$ j.αReq $\leq$ j.αRts)
  $\Rightarrow$ (no one eating)

- Inv A_3 implies desired property

Analysis: progress

Goal: (wfair, bounded eating, ongoing rx, channel progress)
$\Rightarrow$ j hungry *leads-to* j eating

- Define
  - $\overline{\text{hst}}$: seq of all ets's in ets-order // initially `[[0,0]]`
  - $\overline{\text{ne}}$: # requests that have finished eating
- Proof
  - if `[j,s]` is in `j.req`, eventually `[s,j]` $\leq$ `j.`α`Rts` holds
  - after this point
    - `[j,s]`'s index in $\overline{\text{hst}}$ is fixed, at say *n*
    - entries in $\overline{\text{hst}}$`[`$\overline{\text{ne}}$`+1..n]` eat in order

  [entry $\overline{\text{ne}}$'s release msg is incoming to entry $\overline{\text{ne}}$+1's system.
  when it arrives, the latter eventually becomes eating]

Outline

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

- program LockTsDist(ADDR)

```
{c_j} ← start( FifoChannel(ADDR) )
for j in ADDR
    v_j ← start( LockTs(ADDR, j, c_j) )
return {v_j}
```

- 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}
      c_j.tx(k, [REQ, clk, j])
  - await ([req_j, j] ≤ αReq and   // a2
    (ADDR.size = 1 or [req_j, j] ≤ αRts))
  - return

- 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}

Analysis

- Map alg-level state to await-program state

alg-level		await-program
j hungry	$\leftrightarrow$	thread in `j.acq.a2`
j eating	$\leftrightarrow$	`[j,.]` in `j.req`, no thread in `j.acq.a2`
j thinking	$\leftrightarrow$	no `[j,.]` in `j.req`

- Show that alg-level properties are preserved ($\star$)
- 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 $\star$

Outline

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.αReq
  2. [t,j] $\leq$ j.αRts
- Impose additional requirement:
  3. j eats only after rcving 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 $\{\mathtt{rts}_k\}$
  - 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
  - ~~α`Rts`~~, α`Req`
- Messages
  - `[REQ,t,k]`, `[ACK,`~~`t,k]`~~, `[REL,k]`

Solution: system j rules – 1

- Become hungry only if thinking
  - `clk++`
  - `req[j]` $\leftarrow$ `clk`
  - send `[REQ,clk,j]` to every system
  - `na` $\leftarrow$ `0`
- Become eating only if hungry and $[\mathsf{req}_j, j] = \alpha\mathsf{Req}$ and `na` $=$ `ADDR.size` $-$ `1`
- 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)
  - 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:
  - $\overline{\texttt{hst}}_{\texttt{j}}\texttt{.ts}$ to mean $\overline{\texttt{hst}}\texttt{[j][0]}$
  - $\texttt{N}$ to mean $\texttt{ADDR.size}$

- Define
  - $\overline{\texttt{hst}}$: seq of all ets's, initially `[[0,0]]` // as before
  - $\overline{\texttt{ne}}$: # releases globally, initially 0 // as before
  - $\texttt{j.ne}$: # releases seen by j // may lag $\overline{\texttt{ne}}$
  - $\overline{\texttt{tse}}$: ts of last request to release // $\overline{\texttt{hst}}_{\overline{\texttt{ne}}}\texttt{.ts}$
  - $\texttt{j.tse}$: ts of last request released at j // $\overline{\texttt{hst}}_{\texttt{j.ne}}\texttt{.ts}$

Analysis: overview

- Prove: any ts in transit is in $\overline{\texttt{tse}}\,..\,\overline{\texttt{tse}} + 2\,\texttt{N}$
- Prove: `j.tse` is in $\overline{\texttt{tse}} - 2\,\texttt{N}\,..\,\overline{\texttt{tse}}$
- Hence: any ts in transit is in $\texttt{j.tse}\,..\,\texttt{j.tse} + 4\,\texttt{N}$
- Hence can use modulo-M ts, for $\texttt{M} \geq 4\,\texttt{N}$
- Modify system `j` to use cyclic ts
  - add variable `tse`, initially `0`
  - when a request msg is sent, set its ts to `mod(clk, M)`
  - when a request msg `[REQ, ct, j]` is rcvd,
    treat ct as unbounded ts `tse + mod(ct−tse, M)`
  - when `[k,t]` is removed from req, set `tse` to `t`

Analysis: bound ts wrt $\overline{\texttt{tse}}$

- Following are invariant

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

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

  C_3 : $\overline{\texttt{hst}}_{\texttt{p}}\texttt{.ts} \leq \overline{\texttt{hst}}_{\texttt{p+1}}\texttt{.ts} \leq \overline{\texttt{hst}}_{\texttt{p}}\texttt{.ts+2}$

  C_4 : $(\texttt{[REQ,t,j] rcvable}) \ \Rightarrow$

  $\overline{\texttt{hst}}_{\overline{\texttt{ne}}}\texttt{.ts} \leq \texttt{t} \leq \overline{\texttt{hst}}_{\overline{\texttt{ne}}}\texttt{.ts} + 2 \times \texttt{ADDR.size}$

- $Inv\ C_1$: `[REQ,t,j]` in transit implies req `[t,j]` hungry
- $Inv\ C_2$: C_2 satisfies invariance rule
- $Inv\ C_3$: C_3 satisfies invariance rule assuming $Inv\ C_2$
- $Inv\ C_4$: follows from $Inv\ C_1, C_3$

Analysis: bound ts wrt `j.tse`

- Following are invariant

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

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

  C_7 : $\overline{\text{hst}}_{\overline{\text{ne}}}.\text{ts} - 2\times\text{ADDR.size} \leq \overline{\text{hst}}_{\text{j.ne}} \leq \overline{\text{hst}}_{\overline{\text{ne}}}.\text{ts}$

  C_8 : ([REQ,t,j] rcvable at i) $\Rightarrow$

  $$\overline{\text{hst}}_{\text{j.ne}}.\text{ts} \leq \text{t} \leq \overline{\text{hst}}_{\text{i.ne}}.\text{ts} + 4\times\text{ADDR.size}$$

- *Inv* C_5: [t,k] is acked only after k's previous REL msgs are rcvd
- *Inv* C_6: C_6 satisfies invariance rule
- *Inv* C_7: C_7 implied by C_6, C_5, C_3
- *Inv* C_8: C_8 implied by C_7, C_4