Object Transfer using Path Reversal: 
Distributed Path-Reversal Algorithm

Shankar

September 18, 2014
Outline

Path reversal: algorithm
Path reversal: safety analysis
Path reversal: progress analysis
Path reversal: serializability analysis
Path reversal algorithm

- Systems attached to a fifo channel; obj initially at a0

- Messages
  - [REQ, j]: request msg // j is issuer (not forwarder)
  - [OBJ]: object-carrying msg // ignore value for now

- System j: eating (has obj), hungry (wants obj), thinking (o/w)

- System j has a “last” pointer
  - addr in the last req msg rcvd by j after last becoming hungry.
  - nil if no such msg
  - initially nil at a0, and a0 elsewhere

- System j has a “next” pointer
  - nil if j thinking or not rcvd req since non-thinking
  - o/w equals addr in the first req msg rcvd since non-thinking
  - initially nil
System j rules

- become hungry only if thinking:  \[ H(j) \]
  send [REQ,j] to lst
  set lst to nil

- rcv [OBJ]:  \[ E(j) \]
  become eating

- become thinking only if eating and nxt non-nil:  \[ T(j) \]
  send object to nxt
  set nxt to nil

- rcv [REQ,k]:  \[ R(j,k) \]
  if lst not nil
    send [REQ,k] to lst
    set lst to k
  else set nxt and lst to k
**Conventions**

- j–k is a **last edge**: j.lst is not nil and equals k
- j–k is a **next edge**: j.nxt is not nil and equals k
- j–k is a **request edge**: message [REQ, j] is in transit to k

- **digraph**: directed multi-graph
- **LNR**: digraph [addresses; last/next/request edges]
- **L**: digraph [addresses; last edges]
- **LR**: digraph [addresses; last/request edges]

**Drawing conventions**
- last edges: ————
- next edges: - - - - -
- request edges: · · · · ·
Serial evolution: at most 1 hungry at a time

Initially:
1 eating
4 turns hungry, sends \([\text{REQ}, 4]\) to 2
2 forwards \([\text{REQ}, 4]\) to 1
1 \text{rcvs} [\text{REQ}, 4]
1 turns thinking
sends \text{obj} to 4
4 \text{rcvs} \text{object}, turns eating

- \(L\) is an in-tree when no req msg in transit
- each request effects a \textbf{path reversal}
  - \(j\)’s req travels from \(j\) to root
  - all nodes on path now point to \(j\)
- amortized cost of \(\log N\)
Serial evolution: more than 1 hungry at a time

- Initially 1 eating
- 3 turns hungry, sends [REQ,3]
- 1 rcvs [REQ,3]. 1 turns thinking, 3 turns hungry, sends object.
- 1 frwrds [REQ,2].
- 3 rcvs [REQ,2]
- 1 turns thinking sends object. 3 turns eating

- $L$ evolves as before, so amortized cost same
- next ptrs form queue
Non-serial evolution

- \( L \) may never be an in-tree
- can be several next-ptr queues
- progress?
- amortized cost?
Does hungry j eventually eat?

- Possibilities when j’s request msg dies

(a)  
(b)  
(c)  
(d)  
(e)  

Above has implicit assumptions
- L remains acyclic
- j.nxt never points to j
- ...

Now to make argument rigorous
Outline

Path reversal: algorithm
Path reversal: safety analysis
Path reversal: progress analysis
Path reversal: serializability analysis
Basic safety properties

- \textit{Inv} $A_1$--$A_3$ holds \hspace{1cm} // via inv rule
  
  $A_1$: forone ($j$: either ($j$ eating) or (obj in transit to $j$))

  $A_2$: $j$.nxt $\neq$ nil $\Rightarrow$ ($j$.lst $\neq$ nil and ($j$ not thinking))

  $A_3$: ($j$ thinking) $\Rightarrow$ $j$.lst $\neq$ nil

- \textit{Inv} $B_1$--$B_2$ holds \hspace{1cm} // via inv rule assuming \textit{Inv} $A_1$--$A_3$

  $B_1$: LR has exactly 1 undirected path between every two nodes

  $B_2$: for all ($j$: $j$.lst $\neq$ $j$)
- $Inv \ B_3 - B_5$ holds \quad // via inv rule assuming $Inv \ A_1 - A_3, B_1 - B_3$

$B_3 :$ forall $j$: exactly 1 of the following holds
  - $j$ thinking or
  - $[\text{REQ}, j]$ in transit or
  - forsome ($k$: $k.nxt = j$) or
  - $[\text{OBJ}]$ in transit to $j$ or
  - $j$ eating

$B_4 :$ forall ($j$: at most one $[\text{REQ}, j]$ in transit)

$B_5 :$ forall ($j$: $j.nxt \neq j$ and $\text{num}(k: k.nxt = j) \leq 1$)
Want a digraph $Pr$ that captures relative priorities of nodes

- Want $j \rightarrow k$ in $Pr$ to mean $j$ has lower priority than $k$
  - $j \rightarrow k$ is a **pr-next** edge: $k \rightarrow j$ is a next edge
  - $j \rightarrow k$ is a **pr-last** edge: $j \rightarrow k$ is a last edge and $j$ thinking
  - $j \rightarrow k$ is a **pr-request** edge: $j \rightarrow k$ is a request edge

$Pr$: digraph [addresses; req/pr-next/pr-last edges]

Define

- **pr-path**: directed path in $Pr$
- **j pr-reachable from** $k$: pr-path from $k$ to $j$
- **lr-path**: undirected path in $LR$
Some safety properties of $Pr$

- $Inv \ C_1-C_3$  
  // via inv rule assuming $Inv \ A_1-A_3, B_1-B_5$

  $C_1$ : ($Pr$ in-tree) and
  ($Pr$'s root eating or obj in transit to it)

  $C_2$ : pr-path from $k$ to $j \Rightarrow$
  forall $x$ on the lr-path between $j$ and $k$:
  pr-path from $x$ to $j$

  $C_3$ : ($j$ not thinking) and $j.lst=k \neq \text{nil} \Rightarrow$
  ((pr-path from $k$ to $j$) and (k hungry))
Proof of Inv $C_1$ – 1

- Initially $Pr$ is the same as $LR$, so $C_1$, $C_2$, $C_3$ hold.

- $j$ starts eating: $Pr$ not affected, so $C_1$ preserved

- $j$ issues req when $j.lst = w$:
  - $j-w$ goes from pr-last edge to pr-req edge. $C_1$ preserved

- $j$ rcvs req $k$ when thinking:
  - $j-w, k-j \rightarrow k-w, j-k$. $C_1$ preserved (# edges, connectivity preserved)

- $j$ rcvs req $k$ when not thinking, $j.lst = \text{nil}$:
  - $k-j$ goes from pr-req edge to pr-next edge. $C_1$ is preserved
Proof of Inv $C_1 \rightarrow 2$

- j rcvs req k when not thinking, $j.1st = w \neq \text{nil}$:
  - k–j replaced by k–w.
  - (j–w, j–k are not in Pr)

- # edges preserved, so suff to show connectivity preserved

- old Pr has pr-path$(w,j)$
  - suff if old Pr has no pr-path$(w,k)$

- assume old Pr has pr-path$(w,k)$
  - all nodes on lr-path$(k,w)$ have pr-path to k
  - so lr-path$(k,w)$ avoids lr-edges k–j and j–w
  - so undirected cycle in old LR
Proof of Inv $C_1$ – 3

- $j$ stops eating when $j.nxt=x$, $j.lst=w$:
  - $x-j$ replaced by $j-w$
  - old $Pr$ in-tree/root $j$; to show new $Pr$ in-tree/root $x$
  - suff if old $Pr$ has pr-path($w,x$); assume not so
  - so old $Pr$ has pr-path($w,y$) and pr-edge $y-j$, where $y \neq x$
    - $y-j$ is also a lr-edge
    - if $y=w$, then old LR has cycle $[y,j,y]$ // negates $B_1$
    - if $y \neq w$, then lr-path($y,w$) avoids $j$
      lr-path and lr-edges $y-j$, $j-w$ form lr-cycle // $C_1, C_2$
Outline

Path reversal: algorithm
Path reversal: safety analysis
Path reversal: progress analysis
Path reversal: serializability analysis
Progress metric in $Pr$

- $Pr$ can have several next-edge paths
- **next-queue**: a maximal next-edge path  
  \[\text{tail} \oplus \cdots \leftarrow j \leftarrow \cdots \leftarrow \bullet \text{hd}_j\]
  
- $\text{hd}_j$: head of $j$’s next-queue
  - $j$ if $j$ has no incoming next edge

- Goal: fn $F(j)$ st
  - increases while req $\text{hd}_j$ in transit
  - has upper bound at which $\text{hd}_j$ has obj (and no req msg)
Consider $\alpha_j$: set of nodes with pr-paths to $hd_j$

Following hold

$D_1: \alpha_j$ increases when req $hd_j$ is rcvd by a system that is thinking or whose last pointer is nil

$D_2: \alpha_j$ does not decrease while $j$ is hungry
Proof of $D_1$

- Let req $hd_j$ be rcvd by k

- Prior to rcv, $k \notin \alpha_j$  // Pr in-tree, has req edge $[hd_j, k]$

- Different cases of $k$
  - $k$ thinking:
    pr-req $hd_j$–$k$ $\rightarrow$ pr-last $k$–$hd_j$
    $\alpha_j \uparrow$ by $k^+$

  - $k$ not thinking, $k$.lst nil:
    pr-req $hd_j$–$k$ $\rightarrow$ pr-next $k$–$hd_j$
    $k$ becomes $hd_j$, $\alpha_j \uparrow$ by $k^+$  // as in above figure

  - $k$ not thinking, $k$.lst = $x \neq$ nil:
    pr-req $hd_j$–$k$ $\rightarrow$ pr-req $hd_j$–$x$
    $\alpha_j$ no change
Proof of $D_2$

- Consider steps other than rx of req $hd_j$
- $z$ starts eating: neither Pr nor $\alpha_j$ change
- $z$ issues a request: pr-last $z \rightarrow$ pr-req $\quad // \alpha_j$ same
- $z$ not in $\alpha_j$: does not decrease $\alpha_j$
- $z$ in $\alpha_j$ sends object to $y$:
  - old Pr: $z$ is $hd_j$, Pr-root
  - new Pr: $y$ is $hd_j$, Pr-root $\quad // \alpha_j$ same (at max)
- $z$ in $\alpha_j$ rcvs req $k$ when $z.lst = x \neq$ nil:
  - pr-req $k-j \rightarrow$ pr-req $k-x,$
  - old Pr has pr-path($z,x$) (from $C_3$) $\quad // \alpha(j)$ does not decrease
- $z$ in $\alpha_j$ rcvs req $k$ when $z.lst$ nil: <do it>
To compensate for $\alpha_j$, want fn $\beta_j$ that
- $X_1: \uparrow$ when req $hd_j$ rcvd by non-thinking $k$ with non-nil 1st
- $X_2: \downarrow$ only if $\alpha_j \uparrow$ simultaneously

Consider $\beta_j$: set of non-thinking nodes whose 1st equals $hd_j$

- $\beta_j$ and $\alpha_j$ are disjoint // pr-path from $hd_j$ to $\beta_j$ (from $C_2$)

- $X_1$ holds because rcv adds $k$ to $\beta_j$

- $X_2$ holds. $x$ leaves $\beta_j$ in only two ways:
  - $x$ starts thinking: creates pr-last $x-hd_j$, so $\alpha_j \uparrow$ by $x$
  - $x$ rcvs req $k$:
    - pr-req $k-x \rightarrow k-hd(j)$
    - so $\alpha_j \uparrow k$
    - ($k \notin$ old $\alpha_j$ ($C_1-C_2$))
So $F_j = [\alpha_j \text{size}, \beta_j \text{size}]$ under lexicographic ordering works.

We have established the following ($D_5$ used in serializability):

$D_3: (j \text{ eating} \text{ and } k \text{ hungry}) \text{ leads-to } j.\text{nxt} \neq \text{nil}$

$D_4: ((j \text{ eating} \text{ and } j.\text{nxt} \neq \text{nil}) \text{ leads-to } j.\text{nxt} = \text{nil})$

$\Rightarrow (k \text{ hungry} \text{ leads-to } k \text{ eating})$

$D_5: ((j \text{ and } k \text{ are hungry}) \text{ and } (j \text{ pr-reachable from } k))$

$\text{unless } ((j \text{ eating}) \text{ and } (k \text{ hungry}))$
Outline

Path reversal: algorithm
Path reversal: safety analysis
Path reversal: progress analysis
Path reversal: serializability analysis
Goal: Transform any finite evolution $x$ via commutations to a serial evolution $y$ with the same set of sends and rcvs

Let $p$ do the $i$th eating step in $x$, and $q$ do the preceding one. The $i$th eating step is the culmination of

- one $H(p)$ step ($p$ becomes hungry)
- one or more $R(., p)$ steps (rcv req $p$)
- one $T(q)$ step ($q$ starts thinking)
- one $E(p)$ step ($p$ starts eating)

Let $v_i$ be the sequence of the above steps

Let $w$ be the sequence of $x$-steps not in any $v_i$

$x$ is a merge of $v_1, v_2, \ldots, w$

Will show that $y$ is $v_1 \circ v_2 \circ \cdots \circ w$ // hence same cost
Lemma 16.1: Let $f$ and $g$ be two successive steps in $x$ st
- $f$ belongs to $v_i$ and
- $g$ belongs to $v_j$, $j > i$, or to $w$

Then $f$ and $g$ commute wrt the msgs sent and rcvd

$x$ can be transformed to $y$ by repeatedly applying lemma 16.1

Proof of lemma follows
Proof of Lemma 16.1

Let \( g \) be \( H \) or \( R \) of \( v_i \), involving req \( p \).
Let \( f \) be \( H \) or \( R \) of \( v_j \), involving req \( q \).

- Let \( g \) rcv msg sent by \( f \). Then \( g \) in \( v_j \) // contradiction
- Let \( f \) and \( g \) be of same node \( x \).
  Then pr-path\((q, p)\) just after \( f \).
  So \( p \) eats before \( q \) (from \( D_5 \)). // contradiction

Hence \( f \) and \( g \) commute, preserving sends and rcvs

Let \( g \) be \( H \) or \( R \) of \( v_i \), involving req \( p \).
Let \( f \) be \( H \) or \( R \) of \( w \).

- same as above case
Let $g$ be $E$ of $v_i$, ie, rcv obj.
Let $f$ be $H$ or $R$ of $v_j$ or $w$, ie, rcv req.

- $g$ rcvs obj and $f$ sends req. So $g$ does not rcv from $f$.
- Let $f$ and $g$ be at the same system.
  
  Req rcv step ($f$) is same whether hungry ($f, g$) or eating ($g, f$).
  
  So $f$ and $g$ can be interchanged.

Hence $f$ and $g$ commute, preserving sends and rcvs.
Let $g$ be $T$ of $v_i$, ie, send obj.
Let $f$ be $H$ or $R$ of $v_j$ or $w$.
- $g$, being a $T$, does not rcv from $f$
- Let $f$ and $g$ be at the same system, say $x$.
  Then $f$ cannot be $H$ (o/w $g$ could not be $T$)
  Thus $f$ is a $R$ step.
- Suppose $x.1$st was nil prior to $f$.
  Then $g$ would send obj in response to $f$, so $f$ belongs to $v_i$; // contradicts $j > i$.
- Suppose $x.1$st was non-nil prior to $f$.
  Then $f$ and $g$ commute because req rcv ($f$) same whether eating ($f, g$) or thinking ($g, f$).