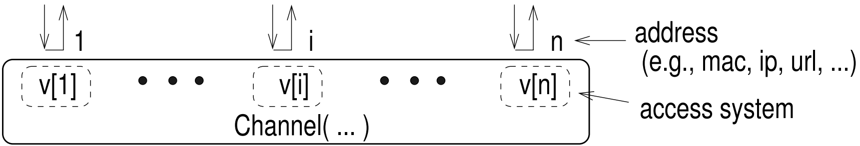
Message-Passing Services (aka Channels)

Shankar

September 18, 2014

Overview: Msg-passing service (aka Channel)



- Distributed service: addresses and access systems
- Connection-less: tx and rx without any prior intimation
  - input fns at `v[i]`: tx, rx
- Connection-oriented: tx and rx only after connection established
  - input fns at `v[i]`: `connect`, `accept`, `close`, `tx`, `rx`, $\cdots$
- Quality: fifo, lossy, LRD (loss, reorder, duplicate), $\cdots$
- Benefits of internal nondeterminism

Outline

- Service FifoChannel(ADDR)
  - ADDR: set of addresses // ADDR.size > 0
  - input fns at j: *s*.tx(k,msg), *s*.rx() // *s*: access system
  - msgs are Seq

- Main:
  - $\mathtt{txh}_{j,k}$ ← [] // seq of msgs sent at j to k
  - $\mathtt{rxh}_{j,k}$ ← [] // seq of msgs rcvd at k from j
  - $\mathtt{v}_j$ ← sid() // access system at j
  - return {$\mathtt{v}_j$} // map from ADDR to sids

  - In an implementation
    - separate return for each $\mathtt{v}_j$
    - but the collection of returns is effectively atomic // Why?

Fifo Channel – 2

- input void v_j.tx(k,msg) // at j, tx msg to k
  - ic {$k \neq j$ and no ongoing v_j.tx(.)}
    append msg to $txh_{j,k}$;
  - oc {true}
    return

- input Seq v_j.rx() // at j, rx msg
  - ic {no ongoing v_j.rx()}
  - output msg, k // rcvd msg, sender addr
    oc {$rxh_{k,j} \circ$ [msg] prefix-of $txh_{k,j}$}
    append msg to $rxh_{k,j}$;
    return msg;

  - k is internal param → internal nondeterminism // Avoidable?

Fifo Channel – 3

- atomicity assumption: input parts and output parts

- progress assumption
  - ongoing v_j.tx(.) *leads-to* not ongoing v_j.tx(.) // tx
  - $txh_{k,j}$.size $\geq$ i *leads-to* // rx
    $rxh_{k,j}$.size $\geq$ i or not ongoing v_j.rx()
  - sum($txh_{k,j}$.size: k in ADDR) $\geq$ i *leads-to* // weaker rx
    sum($rxh_{k,j}$.size: k in ADDR) $\geq$ i or
    not ongoing v_j.rx()

- Program FifoChannelInverse(ADDR, Map<ADDR,Sid> v)
  - main: $txh_{j,k}$, $rxh_{j,k}$, $\cancel{v_j}$
  - output fns: doTx(j,k,msg), doRx(j)
  - progress condition: no change

- output doTx(j, k, msg)
  - oc $\{k \neq j$ and no ongoing v_j.tx(.)$\}$
    append msg to $txh_{j,k}$;
    v_j.tx(k, msg)
  - ic {true}

Fifo Channel Inverse – 2

- output doRx(j)
  - oc {no ongoing v_j.rx()}

    [msg, k] ← v_j.rx()   // local vars
  - ic {$rxh_{k,j} \circ$ [msg] prefix-of $txh_{k,j}$}

    append msg to $rxh_{k,j}$;

- Recall k: internal param in service
  - not present in regular implementation
  - but returned by above implementation v_j.rx()

  $\boxed{\text{augment regular implementation with auxiliary}}$

Outline

Lossy Channel

- Lossy channel is a fifo channel except msgs can be lost
- `LossyChannel(.)`: `FifoChannel(.)` with two changes to v_j.rx
  - output condition:

    $\{rxh_{k,j} \circ [msg]$ ~~prefix-of~~ subsequence-of $txh_{k,j}\}$

  - progress – option 1:

    (*msg* repeatedly sent to j) and
    (j repeatedly calls rx)
    $\Rightarrow$ j receives *msg*

  - progress – option 2

    (msgs from *msgset* repeatedly sent to j) and
    (j repeatedly calls rx)
    $\Rightarrow$ j receives msg from *msgset*

Lossy channel: formalizing rx progress

- Progress – option 1
  - helper functions
    - *nbr*(txh, msg): # of txh entries that equal msg
    - *increasing*(txh, msg):
      nbr(txh, msg) $= i$ *leads-to* *nbr*(msg, txh) $> i$
  - *increasing*($\mathtt{txh}_{j,k}$, msg) and
    (not ongoing v_j.rx *leads-to* ongoing v_j.rx)
    $\Rightarrow$ *increasing*($\mathtt{rxh}_{j,k}$, msg)
- Progress – option 2
  - above except
    - *nbr*(txh, msgset): # of txh entries that are in msgset

Outline

LRD Channel

- LRD channel can lose, reorder and duplicate msgs
  - any message sent in the past can be (again) received
- Service program: option 1
  - LossyChannel(.) with v_j.rx.oc changed to
    $\{rxh_{k,j} \circ [msg]$ ~~subsequence-of~~ in $txh_{k,j}\}$
- Service program: option 2
  - txh_j: seq of msgs sent to j from anywhere
  - rxh_j: seq of msgs rcvd at j from anywhere
  - any msg in txh_j can be received

- Main: txh_j, rxh_j, return $\{v_j\}$

- input void v_j.tx(k,msg)
  - ic {...}
    append msg to txh_k;
  - oc {true}
    return

- input Seq v_j.rx()
  - ic {...}
  - output msg
    oc {msg in txh_j}
    append msg to rxh_j;
    return msg;

- Progress assumption for rx
  - *increasing*(txh_j, msg) and
    (not ongoing v_j.rx *leads-to* ongoing v_j.rx)
    $\Rightarrow$ *increasing*(rxh_j, msg)

- No internal nondeterminism

Outline

Connection-oriented channel – 1

- Connection management + data transfer within connections
- Simplistic connection mgmt
- Addr j
  - closed: inactive
  - accepting: waiting for any remote connect request
  - opening: waiting for response to local connect request
  - open: connected to a remote address
- Input functions
  - `j.accept()`: closed → accepting → open // server
  - `j.connect(k)`: closed → opening → open/closed // client
  - `j.close()`: open → closing → closed
  - `j.tx(msg)`: only while open
  - `j.rx()`: only while open

Connection-oriented channel – 2

- Require msgs of one connection to not show up in another
  - tag each connect attempt with a unique connect number ("cn")
  - cn identifes connection and its txh/rxh
- Add remote addr and cn to "opening" and "open"
  - opening to addr k with cn n
  - open to addr k with cn n
- Become closed only when // graceful closing
  - remote has closed connection or is closing
  - all incoming data received
  - no ongoing rx or tx

- Main
  - $nc \leftarrow 0$ // connect number counter
  - j's status ← closed // connection status of j
  - $txh_{j,n} \leftarrow rxh_{j,n} \leftarrow []$ // tx/rx histories for j with cn n
  - $v_j \leftarrow$ sid() // access system

- input ADDR j.accept()
  - ic {j is closed}

    j becomes accepting;
  - output k

    oc {k is opening to j}

    j becomes open to k with k's cn;

    return k

- input bool j.connect(k)
  - ic { j is closed }
    $n \leftarrow nc{+}{+}$;
    j becomes opening to k with cn n
  - output bool rval
    oc {rval $\Leftrightarrow$ k is open to j with cn n}
    if (rval)
        j becomes open to k with cn n;
    else
        j becomes closed;
    return rval;

- input void j.close()
  - ic {j is open and is not closing}
    $k \leftarrow$ j's remote addr; $n \leftarrow$ j's cn;
  - oc {(k is closing or not open to j with cn n) and
    $rxh_{j,n} = txh_{k,n}$ and
    (j has no ongoing tx or rx)
    }
    j becomes closed;
    return;

- input void j.tx(msg)
  - ic { j has no ongoing tx, is open, and is not closing }
    $n \leftarrow$ j's cn;
    $txh_{j,n}$.append(msg);
  - oc {true}
    return;

- input Seq j.rx()
  - ic { j is open and has no ongoing rx }
    k ← j's remote addr; n] ← j's cn;
  - output rval, msg
    oc {(rval = [−1] and $rxh_{j,n} = txh_{k,n}$ and
    remote is closing or not open to k with cn n)
    or
    (rval = [0,msg] and
    ($rxh_{j,n} \circ$ [msg]) prefix-of $txh_{k,n}$)
    }
    if (rval[0] = 0)
    $rxh_{j,n}$.append(msg);
    return rval;

- Progress assumption
  - ongoing j.accept and ?? *leads-to* not ongoing j.accept
  - ongoing j.connect *leads-to* not ongoing j.connect // ??
  - ongoing j.tx *leads-to* not ongoing j.tx
  - ongoing j.rx and (j open to k with cn n) and
    $rxh_{j,n}$.size $<$ $txh_{k,n}$.size or
    ($rxh_{j,n}$.size $=$ $txh_{k,n}$.size and
    (k closing or not open to j with n))
    leads-to (not ongoing j.rx)
  - ongoing j.close and (j open to k with cn n) and
    (k closing or not open to j with n) and
    (j.rx repeatedly called)

    leads-to not ongoing j.close

Outline

Distributed multiplexor program


```
program MuxorDist(ADDR, PORT)

    // start connection-less channel
    {c_j} ← startSystem(FifoChannel(ADDR));

    // start muxor system at each address
    for (j in ADDR)
        v_j ← startSystem(Muxor(ADDR, PORT, j, c_j));

    // return muxor system sids
    return {v_j};
```

Multiplexor program – 1


```
program Muxor (ADDR, PORT, aL, cL)
    // aL: local address.  cL: local channel system
    rbuffp ← [];                          // rx queue for port p
    startThread(doRx());                  // thread to receive msgs

    input mysid.tx (pR, aR, msg)          // remote port/addr pR/aR
        await (true)
            cL.tx(aR, [pR, msg])
        return

    input Seq mysid.rx (pR)               // local port p
    • await (rbuff[pR] ≠ [])
            msg ← rbuff[pR][0];
            rbuff[pR].remove();
        return msg;
```

```
// receive msg from channel, and buffer; local thread
function doRx()
   while (true)
   • msg ← cL.rx();
     await (true)
        rbuff[msg[0]].append(msg[1]);
```

```
atomicity assumption {awaits}

progress assumption {weak fairness of threads}
```