Reliable Transport Service

Shankar

June 10, 2014

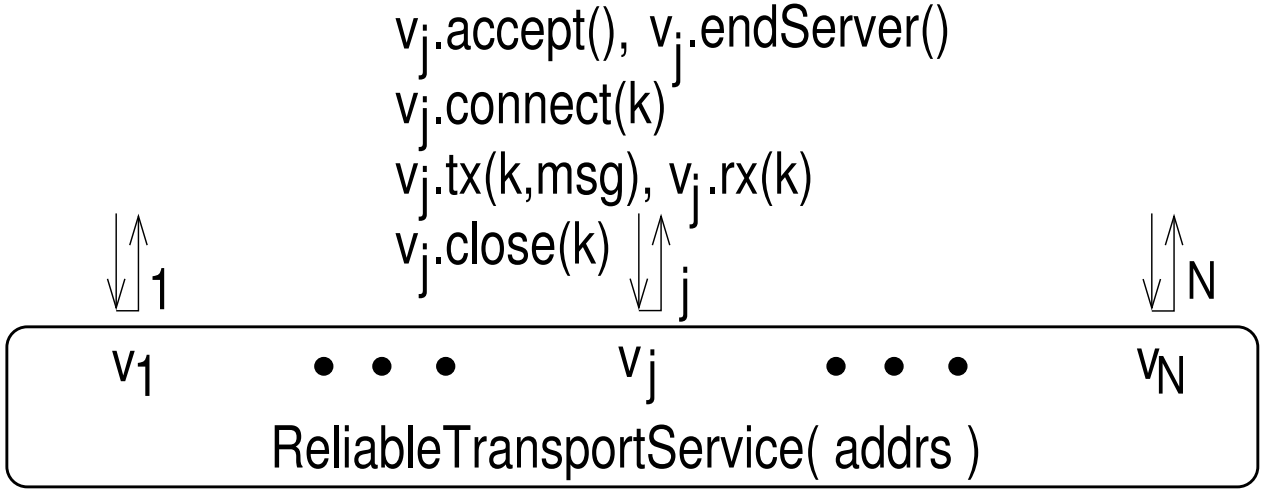
Outline

Overview: Reliable transport service w/o Abort

Program: Reliable transport service w/o Abort

Reliable transport service with Abort

Basic features



- Like reliable streaming Internet sockets
  - addr $\leftrightarrow$ [ip addr, tcp port]
- User starts as server (accept) or client (connect)
- Client-server and client-client connections
- Tx/rx data on connection
- Graceful closing
- Client or server can be first to open // Transaction TCP
- Non-abortable // resend until ack rcvd

Simplified description – 1

- `j.accept()` // willing to accept a conn req
  - `j` enters server mode (if not already so)
  - call blocks until connected or canceled
  - returns `[k]`: connected to `k`
    - overlapping `k.connect(j)`
  - returns `[]`: canceled
- `j.endServer()`
  - ends server mode at `j`
  - cancels any ongoing accept call
  - deletes any buffered conn reqs
- In server mode, incoming conn reqs are buffered
  - to be handled (by future accept call) or canceled

Simplified description – 2

- `j.connect(k)` // request connect to k
  - blocks until connected or rejected
  - returns `[k]`: connected to k
  - returns `[]`: rejected
- `j.tx(k, msg)` // send msg to k
  - call only when `j` connected to k
- `j.rx(k)`: // rcv msg from k
  - call only when `j` connected to k
  - blocking
  - returns msg or null (if remote closing and all msgs rcvd)

Simplified description – 3

- `j.close(k)` // no further `j.tx` on connection
  - call only when `j` connected to `k`
  - returns only after // graceful closing
    - remote has called `k.close(j)`
    - all msgs sent by remote have been rcvd
  - return cancels any ongoing `j.rx(k)`
- `j` can have multiple connections ongoing (to different addrs)
  - customary to identify connection to `k` by `[j,k]` // socket

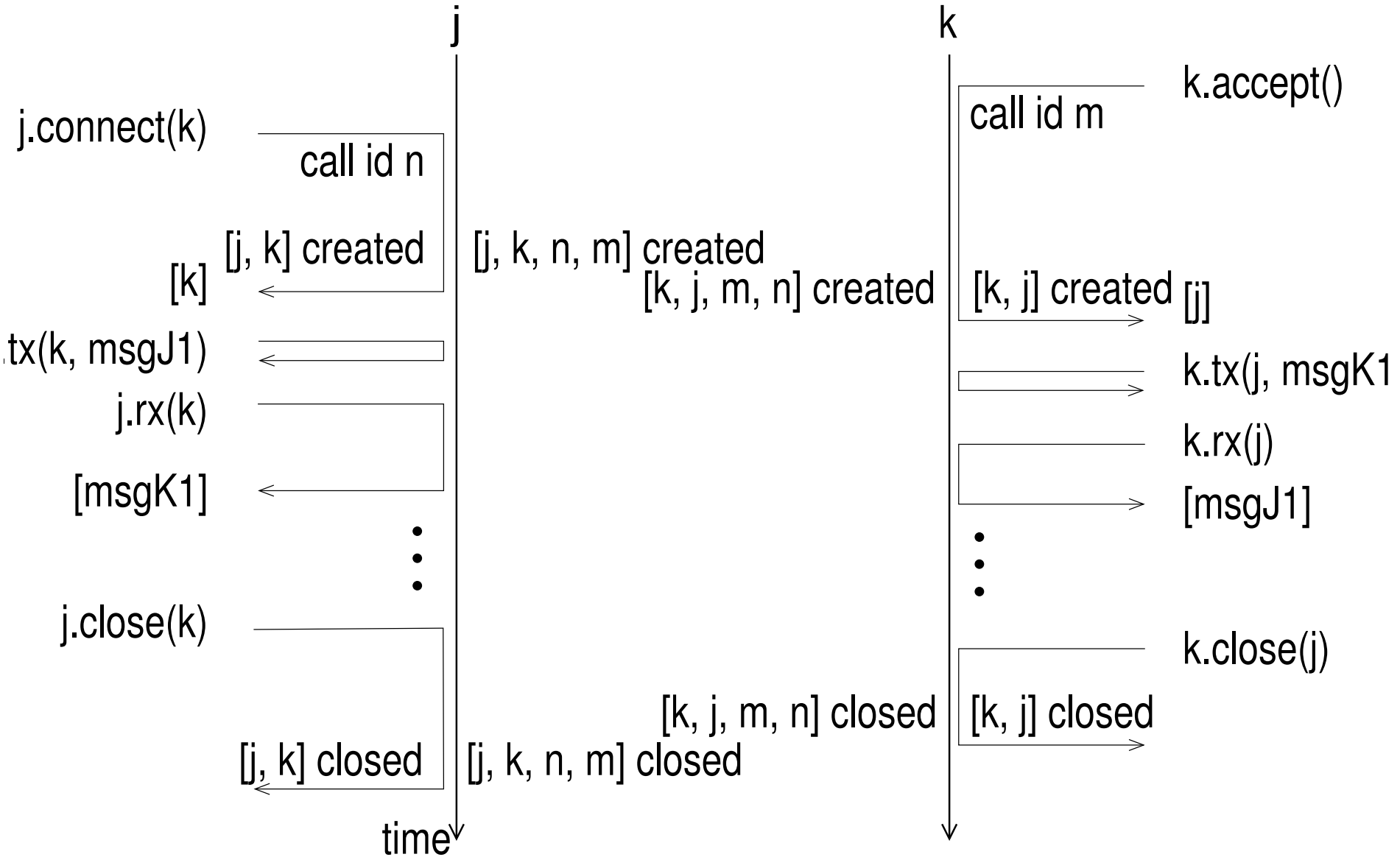
Issues to be addressed

- A connection should not see msgs of previous connections
  - `[j,k]` socket should rcv msgs only from the `[k,j]` socket to which it connected
  - and not from `[k,j]` sockets of previous `j-k` connections
- Conditions that ensure a connect call succeeds
  - progress, not performance
- ...

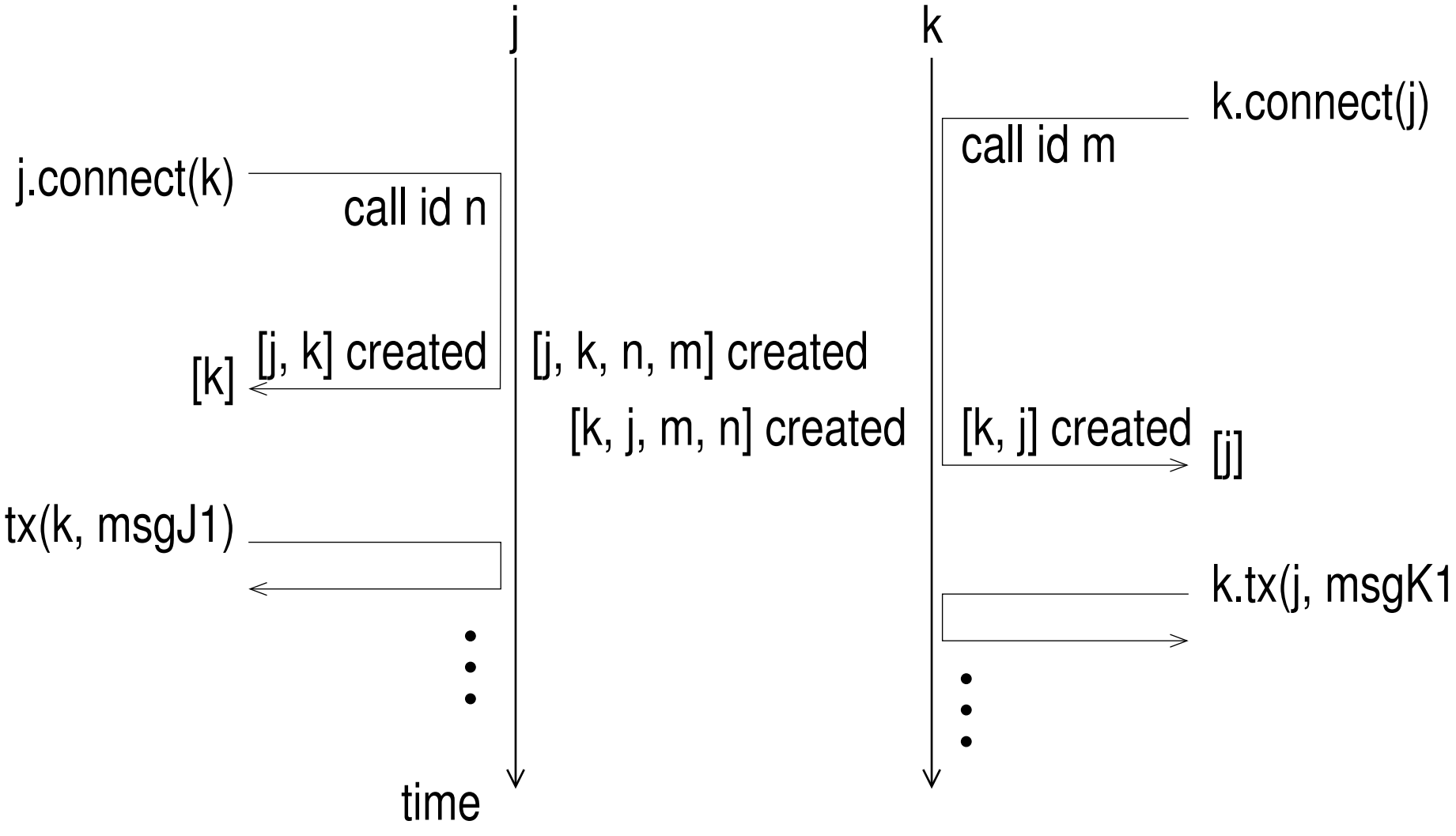
Service endpoints

- Tag every connect and accept call with a unique call id
- Service endpoint: 4-tuple of socket and local/remote call ids
- Example
  - `j.connect(k)` with call id `n` connects to `k.accept` with call id `m`
  - endpoint at `j`: `[j,k,n,m]`
  - corresponding endpoint at `k`, if created: `[k,j,m,n]`
- Require `[j,k,n,m]` to rcv msgs only from `[k,j,m,n]`
- Call ids are internal params // internal nondeterminism

Client-server connection example



Client-client connection example



Outline

Overview: Reliable transport service w/o Abort

Program: Reliable transport service w/o Abort

Reliable transport service with Abort

Helper functions – 1

- Endpoint `[j,k,n,m]`
  - opened: has been created
  - closed: opened, then `j.close(k)` called and returned
  - open: opened and not closed
  - closing: open and `j.close(k)` ongoing
- connecting(j,k,n): ongoing `j.connect(k)` with cid n
- accepting(j,n): ongoing `j.accept()` with cid n
- openedTo(j,n): endpoint `[k,j,m,n]` opened forsome k, m
- overlapped(j,k,n,m): connecting(j,k,n) overlapped with accepting(k,m) or with connecting(k,j,m)

Helper functions – 2

- connecting(j,k,n) has dedicated accept:
  - while connecting(j,k,n)
    - k.accept() is ongoing (and not canceled)
    - no other client attempts to connect to k
  - when connect was called
    - no [k,j] socket
    - no ongoing j.accept // to avoid potential k.connect(j)
- accepting(j,n) has dedicated connect:
  - while accept ongoing, for some k
    - k.connect(j) is issued
    - there is no [j,k] socket,
    - j.connect(k) is not issued // to avoid k.connect(j)

Service program – 1

- Parameters: `ADDR`
- Input fns
  - `j.accept()`, `j.endServer()`, `j.connect(k)`, `j.close(k)` `// cm`
  - `j.tx(k,msg)`, `j.rx(k)` `// dt`
- No output fns
- Main
  - `cidgen`: call-id generator, initially 0
  - addrs in server-mode
  - cids of ongoing accepts and connects
  - cid pairs of open sockets
  - history of accept/connect/close calls and returns
  - msg txh/rxh histories for each opened endpoint

Service program – 2

- input fn `j.accept()`
  - ic {no ongoing `j.accept`}
    `n` ← `cidgen++` ; put `j` in server-mode
  - output (`rval`, remote addr `k`, remote cid `m`)
    oc { `rval=[k]` & overlapped`(k,j,m,n)` & no open `[j,k]` &
    (not openedTo`(j,n)` or opened`(k,j,m,n)`)
    OR
    `rval=[]` & ongoing `j.endServer` & not openedTo`(j,n)`
    }
    update state ; return `rval`

- input fn `j.connect(k)`
  - ic {no open or connecting [j,k]}
    `n` ← cidgen++ ; update state
  - output (`rval`, remote cid `m`)
    oc { `rval` = `[k]` & overlapped(j,k,n,m) & no open `[j,k]` &
    (not openedTo(j,n) or opened(k,j,m,n))
    OR
    `rval` = `[]` & no dedicated accept & not openedTo(j,n)
    }
    update state ; return `rval`

- input fn `j.endServer()`
  - ic {no ongoing `j.endServer`}
    remove `j` from server-mode
  - oc { no ongoing `j.accept` }
    update state ; return
- input fn `j.tx(k, msg)`
  - ic {`[j,k]` open and not closing & no ongoing `j.tx(k,.)`}
    update state
  - oc { true }
    return

- input fn `j.endServer()`
  - ic {no ongoing `j.endServer`}
    remove `j` from server-mode
  - oc { no ongoing `j.accept` }
    update state ; return
- input fn `j.tx(k,msg)`
  - ic {`[j,k]` open and not closing & no ongoing `j.tx(k,.)`}
    update state
  - oc { true }
    return

- input fn j.rx(k)
  - ic {[j,k] open & no ongoing j.rx(k)}
  - output (rval, msg }

    let [n,m] be cid-pair of open [j,k]

    oc { rval = [0,msg] & (rxh[n,m] ∘ [msg] prefix-of txh[m,n])
    OR
    rval = [−1] & rxh[n,m] = txh[m,n] &
    closing(k,j,m,n) or closed(k,j,m,n)
    }

    update state ; return rval

- input fn `j.close(k)`
  - ic {`[j,k]` open & no ongoing `j.close(k)`}
  - let `[n,m]` be cid-pair of open `[j,k]`

    oc { no ongoing `j.tx(k,.)` or `j.rx(k)` &
    `rxh[n,m]` = `txh[m,n]` &
    closing`(k,j,m,n)` or closed`(k,j,m,n)`
    }

    update state ; return

- Progress assumption
  - accepting(j,n) & (dedicated connect or ending server)
    leads-to not accepting(j,n)
  - connecting(j,k,n) *leads-to* not connecting(j,k,n)
  - opened(j,k,n,m) *leads-to* opened(k,j,m,n)
  - ongoingj.endServer *leads-to* no ongoingj.endServer
  - ongoingj.tx(k,.) *leads-to* no ongoingj.tx(k,.)
  - ongoing j.endServer *leads-to* no ongoingj.endServer

// continued

// continued

- Progress assumption
  - ongoing j.tx(k,.) *leads-to* no ongoingj.tx(k,.)
  - ongoing j.rx(k) & open(j,k,n,m) &
    (rxh[n,m] $\neq$ txh[m,n] or closing(k,j) or closed(k,j,m,n))
    leads-to no ongoing j.rx(k)
  - closing(j,k,n,m) & (ongoing k.close(j) or closed(k,j,m,n))
    leads-to (no ongoing j.close(k) or no ongoing j.rx(k))

Outline

Basic features

- Abortless service is implementable over an LRD channel
  - need to resend a "to-be-acked" msg until the ack is rcvd
  - impractical in context of failures
- Abortable service: abort after specified number of resends
- Abortable service has many more acceptable evolutions
  - endpoint `[j,k,n,m]` opens but endpoint `[k,j,m,n]` never opens
  - `j.close(k)` returns before remote is closing or all data rcvd
  - `j.connect(k)` returns unsuccessfully even if it is not rejected