Reliable Transport Protocol

Shankar

June 16, 2014

Outline

Overview

Graceful-closing data transfer protocol

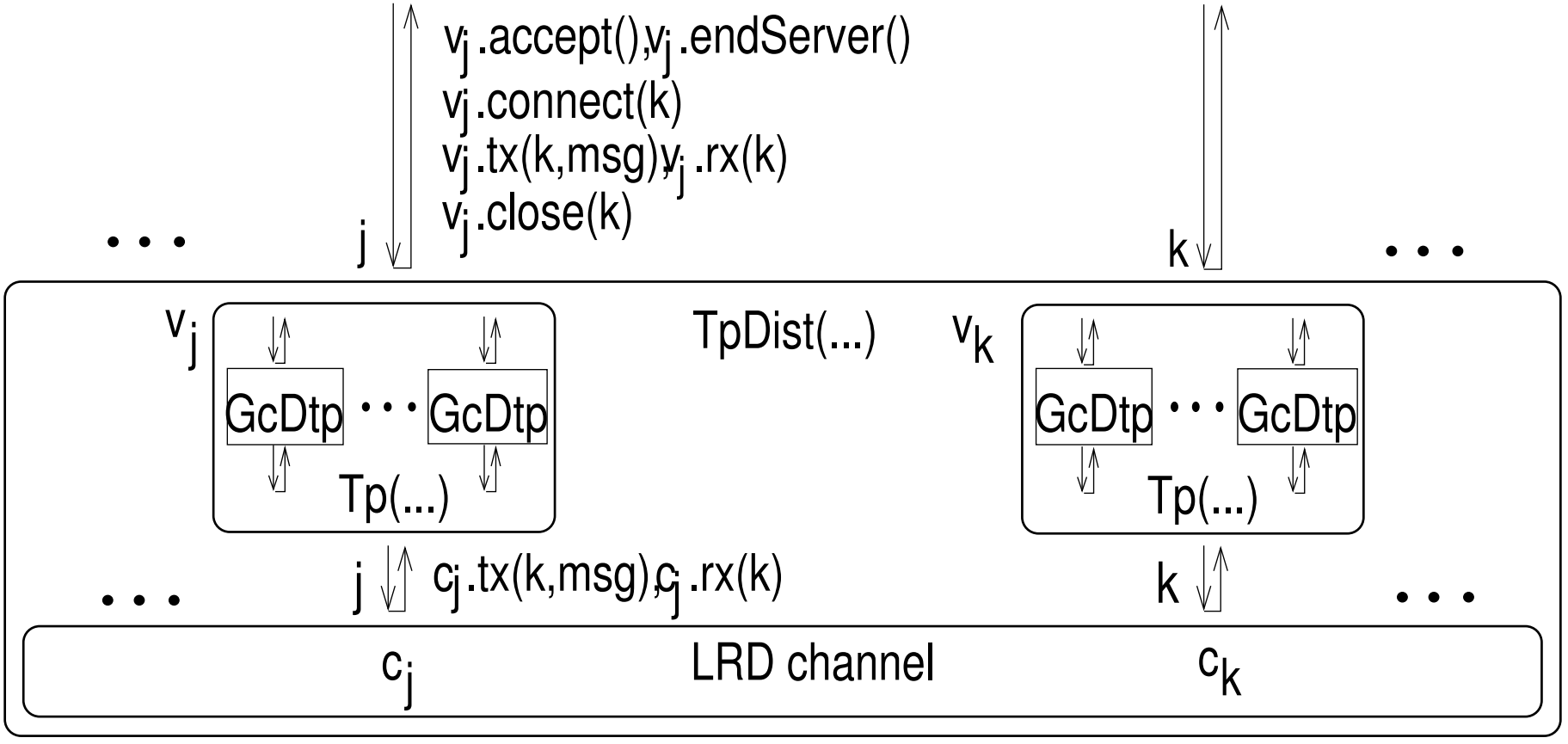
Transport protocol description

Transport protocol program: unbounded endpoint numbers

Transport protocol program: cyclic endpoint numbers

Transport protocol with abort

Transport protocol overview



- Implements reliable transport service using LRD channel
- Tp system at each address

Distributed program

- program TpDist(ADDR)

  // implements RelTransportService(ADDR)

  $\{c_j\}$ ← start LrdChannel(ADDR)

  for j in ADDR

  v_j ← start Tp(ADDR, j, c_j)

  return $\{v_j\}$

Similarities with TCP

- Connection establishment involves 3-way handshake
- Tp `j` maintains an endpoint for `k` only while interacting with `k`
  - opening, open, open-and-closing, closed
  - opening: active (if client) or passive (if server)
- Each endpoint gets a unique endpoint number when created
  - same role as TCP's initial sequence numbers
  - increasing but need not be consecutive // clock, counter
  - when open, maintains both local and remote endpoint numbers
- Endpoint's 4-tuple: `[j,k,n,m]` // addrs `j`, `k`; endpt numbers `n`, `m`

Differences with TCP

- Clean separation of connection establishment and data transfer
- Conn establishment provides dedicated virtual channel
  - by tagging msgs with endpoint's 4-tuple
- Can run any dtp (data transfer protocol) over this
  - for concreteness, use graceful-closing dtp (`GcDtp`)
- Tp system starts a dtp system when endpoint becomes open
  - relays msgs: dtp system $\leftrightarrow$ user, LRD channel
- Connect req msg indicates whether sender is client or server
  - TCP does not do this, and hence has a flaw

Outline

Overview

Graceful-closing data transfer protocol

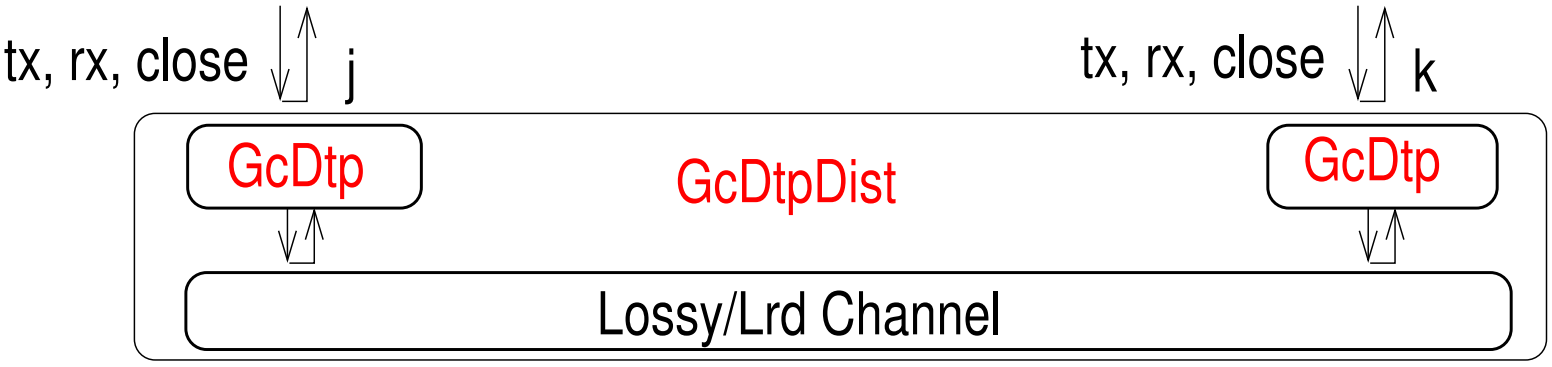
Transport protocol description

Transport protocol program: unbounded endpoint numbers

Transport protocol program: cyclic endpoint numbers

Transport protocol with abort

Graceful-closing dtp: review



- Input fns
  - tx(k, data)
  - rx(): returns [0, data] or [−1] (if in-data done)
  - close(): returns if data xfr done, remote closing
- Msgs: DAT, ACK, FIN, FINACK // FINs sent by close
- Local thread fns:
  - doTxDat(): sends DAT
  - doRxDatAck(): rcvs; sends ACK, FINACK

Graceful-closing dtp: properties

A_1 : if j.rx returns [0,db] then j.drxh $\circ$ [db] prefix-of k.dtxh

A_2 : if j.rx returns [−1] then j.drxh = k.dtxh, k closing/closed

A_3 : if j is closed then no ongoing j.tx or j.rx, j.drxh = k.dtxh, and k closing/closed

A_4 : if j.tx is ongoing then j.tx returns

A_5 : if j.rx ongoing, j.drxh $\neq$ k.dtxh, then j.rx returns

A_6 : if j.rx is ongoing, j.drxh = k.dtxh, k closing/closed then j.rx returns

A_7 : if j closing, k closing/closed then j becomes closed or j.rx not ongoing

Two modifications to GcDtp

- Terminate GcDtp system when it becomes closed:
  - fn close: insert endSystem() before the return
  - original: doRxDatAck() responds to FIN even when closed
  - now: tp system handles takes care of this

- Rename output calls: c.tx, c.rx $\rightarrow$ x.dTx, x.dRx
  - original: c is Lrd channel sid
  - now: c is tp sid // already has input fns tx, rx

Outline

Overview

Graceful-closing data transfer protocol

Transport protocol description

Transport protocol program: unbounded endpoint numbers

Transport protocol program: cyclic endpoint numbers

Transport protocol with abort

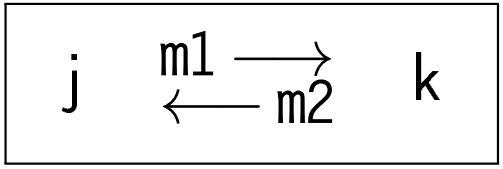
Transport protocol endpoints

- Endpoint (ep) [j, k, n, m]
  - maintained at tp j while interacting with tp k
  - n: ep's local number; ≥ 0
    - set from increasing clock/counter when ep created
  - m: ep's remote number; -1 if unknown
    - rcvd from remote endpoint [k, j, m, n]
- Ep [j, k, n, m]
  - active opening (aop): j.connect(k) ongoing
  - passive opening (pop): j.accept responding to k.connect(j)
  - open: connected to [k,j,m,n]; send/rcv data
  - closing: j.close(k) ongoing; rcv data // still open
  - closed: endpoint no longer exists

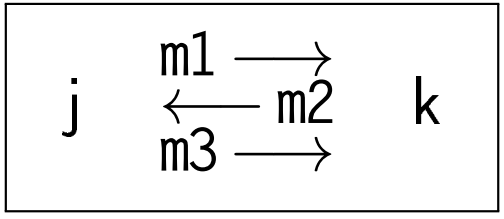
Interaction between tp systems

- Interaction between j and k is a succession of handshakes
  - each connect/close req initiates a handshake

2-way handshake

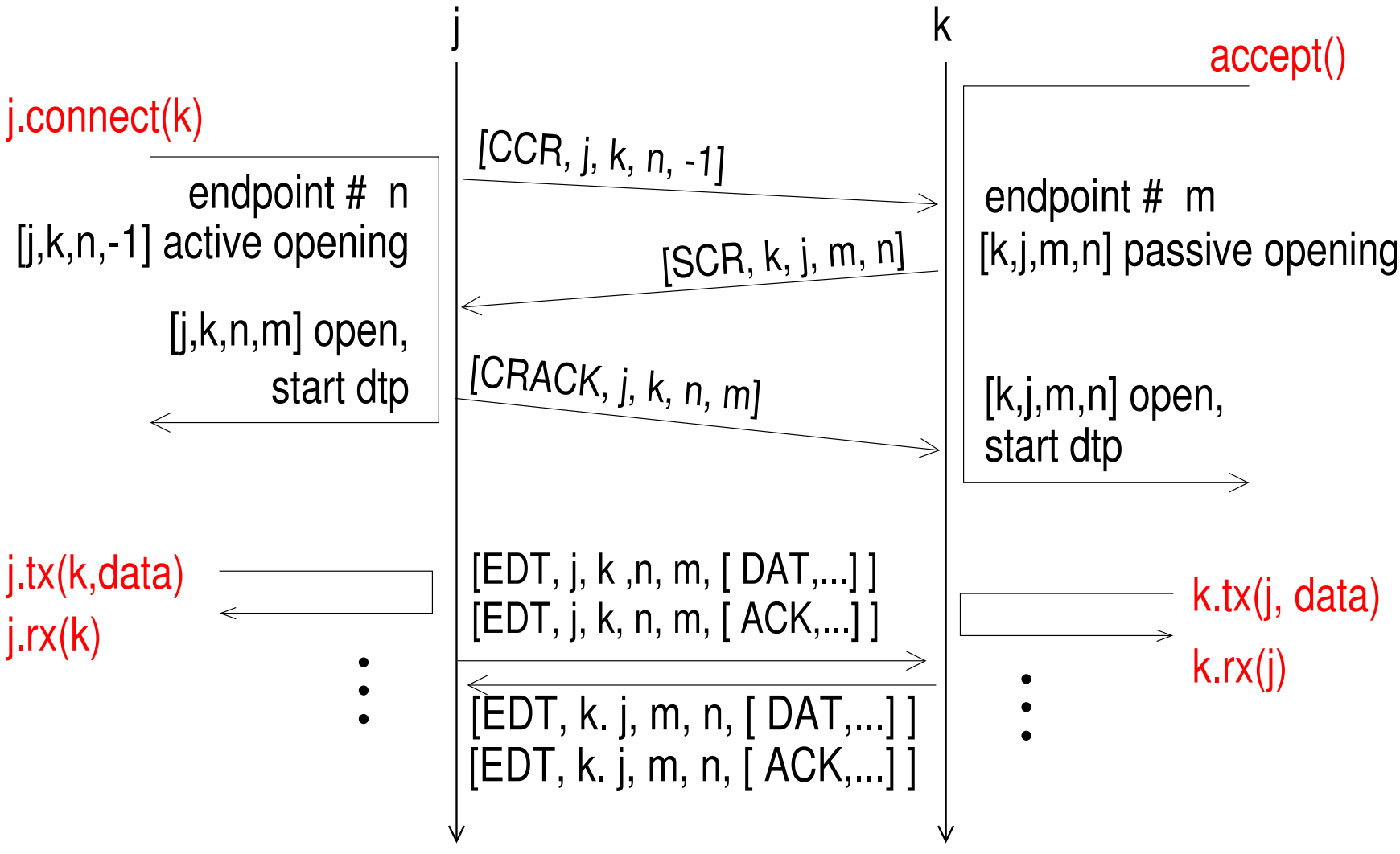


3-way handshake

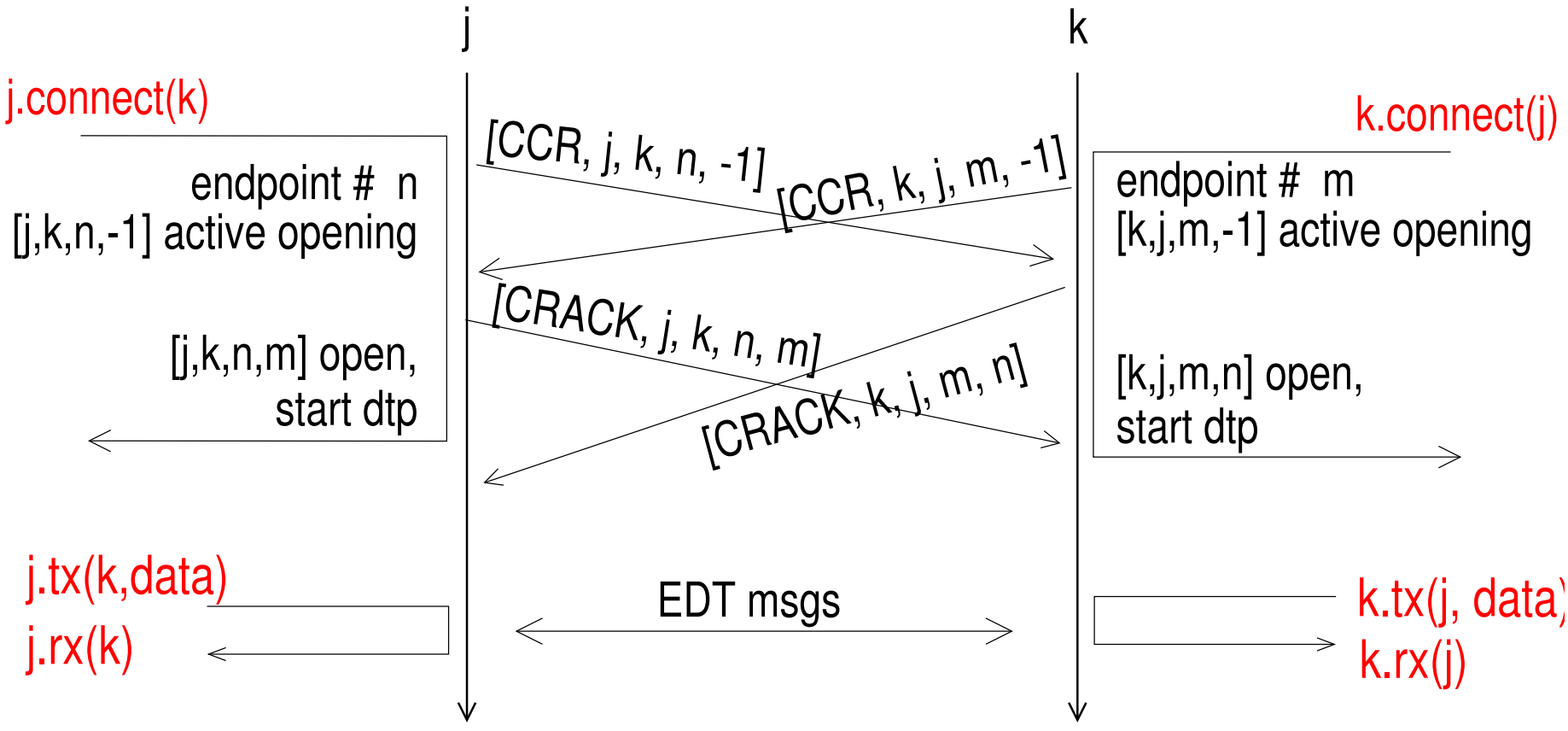


- Client-server connection: one 3-way handshake
- Client-client connection: two simultaneous 2-way handshakes
- Connect rejection: two simultaneous 2-way handshakes
- Data transfer: 2-way handshakes
- Close: two 2-way handshakes // one for each direction

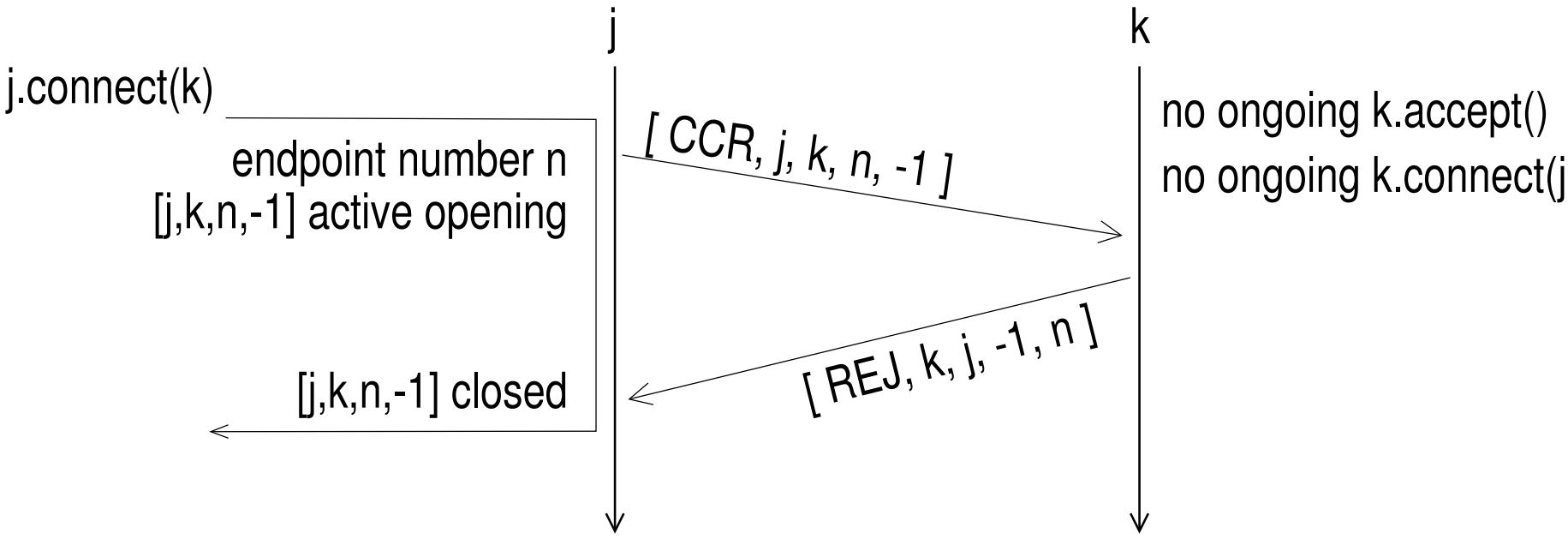
Client-server connection – assuming fifo channel



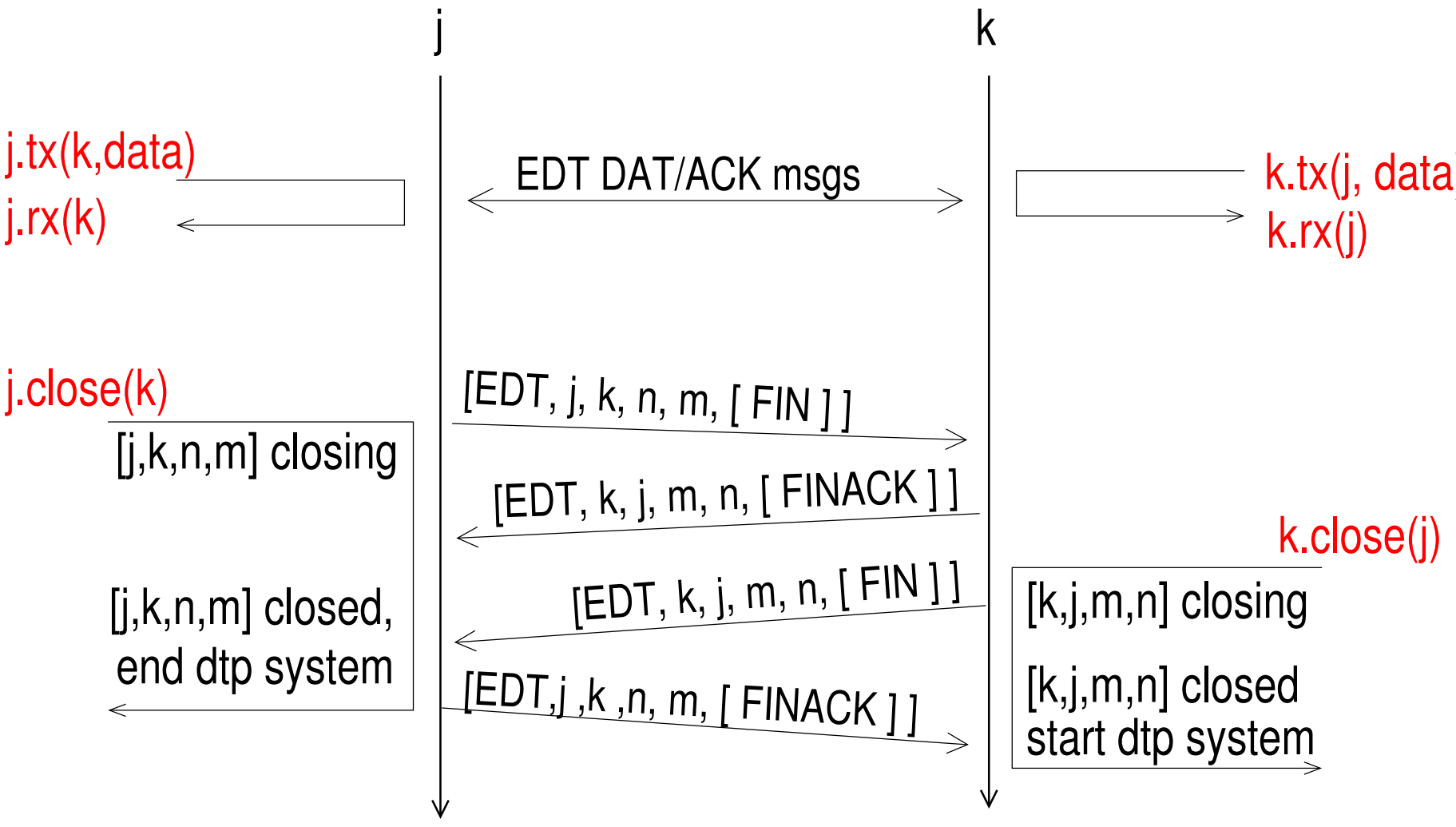
Client-client connection – assuming fifo channel



Client connect rejection – assuming fifo channel



Connection closing – assuming fifo channel



Handling LRD

- To overcome msg loss
  - non-final msgs resent until response rcvd // primary msg
  - final msg sent only in response // secondary msg
  - final msg sent even if handshake over from sender's perspective
- Old msgs can start handshakes, which need to be ended
  - handle via reject msgs / increasing ep numbers
  - example
    - accepting k rcvs old [CCR,j,k,m,−1]
    - starts pop [k,j,n,m], (re)sends [SCR,k,j,n,m]
    - if j not aop to k: j rejects SCR
    - if j aop [j,k,p,−1]: k becomes pop [k,j,q,p]

Tp messages from j to k

- [CCR, j,k,n,m]: client conn-req; primary; sent by aop [j,k,n,m]
- [SCR, j,k,n,m]: server conn-req; primary; sent by pop [j,k,n,m]
- [CRACK, j,k,n,m]: conn-req ack; secondary
  - response to [SCR|CCR, k,j,m,n] by aop/open [j,k,n,m]
- [REJ, j,k,n,m]: reject; secondary
  - response to [CCR, k,j,m,n] when not accepting or aop to k
  - response to [SCR, k,j,m,n] when not aop to k
- [EDT, j,k,n,m, dtmsg]: encapsulates dtpmsg
  - primary if dtmsg is DAT or FIN
  - secondary if dtmsg is ACK or FINACK

Overlapping handshakes

- Handshakes can overlap
  - j starts a handshake while k is still in previous handshake
  - msg of later handshake can end the earlier handshake
  - example
    - opening [j,k,n,m] rcvs `[EDT,k,j,m,n,.]`
    - becomes open

Buffering rcvd CCRs at j in server mode

- ccrBuff: map <addr, ep#> to store CCRs in server mode
- $ccrBuff_k$ exists and equals n iff
  - j in server mode
  - no [j,k] socket exists
  - at least one [CCR,k,j,..,.] rcvd
  - n is highest sender ep # in these CCRs
- An ongoing j.accept() gets its next CCR from ccrBuff
  - fifo or priority queuing

Outline

Overview

Graceful-closing data transfer protocol

Transport protocol description

Transport protocol program: unbounded endpoint numbers

Transport protocol program: cyclic endpoint numbers

Transport protocol with abort

Program Tp – 1

- Parameters: ADDR, j, c_j
- Input fns // called by service users
  - accept(), endServer(), connect(k), close(k) // cm
  - tx(k, data), rx(k) // dt
- Input fns // called by dtp_k
  - dtx(k,dtmsg), drx(k) // send/rcv dtmsg
- Local fns
  - doRx() // rcv msg, update state, tx secondary msg
- Output calls
  - c_j.tx(k, msg), c_j.rx() // send/rcv msg
- atomicity assumption: await
- progress assumption: weak fairness for all threads

Program Tp – 2

- Main
  - ngen: endpoint number generator // clock/counter, initially 0
  - For every k st ep [j,k] exists // initially none
    - st_k: status: aop, pop, rejected, open // no closed
    - ln_k: local number, ≥ 0
    - rn_n: remote number, -1 if null
  - Iff in server mode, no ep [j,k,..], ≥ 1 CCR rcvd from k
    - $ccrBuff_k$: highest sender ep # in these CCRs
  - For every open ep [j,k,..]
    - dtp_k: sid of dtp system for k
    - $dtpRxQ_k$: rcvd dtp msgs for dtp_k
  - startThread(doRx())

- input mysid.accept():

```
while (true)
    await (not server-mode or ccrBuff not empty)
        if (not server-mode)
            return []
        k ← earliest key in ccrBuff
        (st_k, ln_k, rn_k) ← (pop, ngen++, crBuff_k delete)
        while (true)
            if (st_k is pop)
                c_j.tx(k, [SCR, j, k, n, rn_k]
            else if (st_k = open)
                return [k]
            else // st_k = RJCT
                delete st_k, ln_k, rm_k, break
```

- input mysid.endServer():
  await (true)
    exit server-mode
    for (k in ccrBuff.keys)
      c_j.tx(k, [REJ, j, k, $ccrBuff_k$, −1])
    empty ccrBuff
  await (no ongoing accpet)
    return

- input mysid.connect(k):

```
await (true)
  (st_k, ln_k, rn_k) ← (aop, ngen++, −1)
while (true)
  await (true)
    if (st_k is aop)
      c_j.tx(k, [CCR, j, k, n, rn_k]
    else if (st_k = open)
      return [k]
    else // st_k = RJCT
      delete st_k, ln_k, rm_k
      return []
```

- input mysid.tx(k, data):
  dtp_k.tx(k, data)
  return
- input mysid.rx(k):
  rval $\leftarrow$ dtp_k.rx()
  return rval
- input mysid.dtx(k, dtmsg):
  await (true)
    c_j.tx([EDT, j, k, ln_k, rn_k, dtmsg])
  return rval
- input mysid.drx(k):
  await ($dtpRxQ_k$ not empty)
    remove head of $dtpRxQ_k$
  return it

- input mysid.close(k):
  dtp$_k$.close()
  delete dtp$_k$, dtpRxQ$_k$
  return

- doRx(): // executed by local thread
  while (true)
  (type, k, m, n, dtmsg) $\leftarrow$ c$_j$.rx()
  handle<type>(k, m, n, dtmsg) // dtmsg only for EDT

- helper fn startDtp(k):
  dtp$_k$ $\leftarrow$ start GcDtp(j, k, mysid, ...)
  dtpRxQ$_k$ $\leftarrow$ []

- helper fn handleCCR(k, m, n):

```
if (no st_k)
    if (not server-mode)   c_j.tx([REJ, j, k, n, m])
    else if (no ccrBuff_k  or  m > ccrBuff_k)   ccrBuff_k ← m
else if (st_k) is aop  and  n = ln_k)
    (st_k, rn_k) ← (open, m)
    startDtp(k)
else if (st_k) is pop  and  m > rn_k)
    rn_k ← m
else if (st_k) is open)
    if ([m, n] = [rn_k, ln_k])
        c_j.tx([CRACK, j, k, n, m])
    else if (m > rn_k)
        c_j.tx([REJ, j, k, n, m])    // dtpRxQ_k.append([FINACK]) ??
```

- helper fn handleSCR(k, m, n):

```
if (no st_k)
    c_j.tx([REJ, j, k, n, m])
else if (st_k) is aop  and  n = ln_k)
    (st_k, rn_k) ← (open, m)
    startDtp(k)
    c_j.tx([CRACK, j, k, n, m])
else if (st_k) is pop  and  m > rn_k)
    c_j.tx([REJ, j, k, n, m])
else if (st_k) is open)
  if ([m, n] = [rn_k, ln_k])
      c_j.tx([CRACK, j, k, n, m])
  else if (m > rn_k)
      c_j.tx([REJ, j, k, n, m])    // dtpRxQ_k.append([FINACK]) ??
```

- helper fn handleCRACK(k, m, n):
  if (st_k exists and st_k is aop or pop and
  [m, n] = [rn_k, ln_k])
  $st_k \leftarrow$ open
  startDtp(k)

- helper fn handleREJ(k, m, n):
  if (st_k exists and st_k is aop or pop and
  [m, n] = [rn_k, ln_k])
  $st_k \leftarrow$ rjctd

- helper fn handleEDT(k, m, n, dtmsg):
  if (no st_k)
    if (dtmsg = [FIN])
      c_j.tx([EDT, j, k, n, m,,[FINACK]])
  else if (st_k is aop or pop and
    [m, n] = [rn_k, ln_k])
    $st_k \leftarrow$ open
    startDtp(k)
    $dtpRxQ_k$.append(dtmsg)
  else if (st_k is open and
    [m, n] = [rn_k, ln_k])
    $dtpRxQ_k$.append(dtmsg)

Outline

Overview

Graceful-closing data transfer protocol

Transport protocol description

Transport protocol program: unbounded endpoint numbers

Transport protocol program: cyclic endpoint numbers

Transport protocol with abort

Using cyclic endpoint numbers

- Above protocol can use modulo-N endpoint numbers if

$$\mathsf{N} \geq \frac{6L + 4W + 2C}{\delta}$$

  - L: max message lifetime of the LRD channel
  - δ: min time between ngen increases (new endpoints) at an addr
  - W: max opening duration of an endpoint
  - C: max open duration of an endpoint

- Note
  - L, δ arise as in sliding window protocol
  - W, C arise because j tracks k only while opening/open to it
  - set C to 0 for correctness with Pr $\approx 1 - (1/\mathsf{N}^2)$

- Let j rcv msg [., k, j, m, n] when it has ep [j, k, $\widetilde{n}$, $\widetilde{m}$]
-

Message	Receiving endpoint	Possible tests
[CCR|SCR,k,j,m,n]	[pop|aop,$\widetilde{n}$,$\widetilde{m}$]	n = $\widetilde{n}$; m > $\widetilde{m}$; m > ccrBuff[k]
[CCR|SCR,k,j,m,n]	[open,$\widetilde{n}$,$\widetilde{m}$]	m = $\widetilde{m}$; n = $\widetilde{n}$; m > $\widetilde{m}$
[CRACK|REJ,k,j,m,n]	[aop|pop,$\widetilde{n}$,$\widetilde{m}$]	m = $\widetilde{m}$; n = $\widetilde{n}$
[EDT,k,j,m,n,.]	[aop|pop|open,$\widetilde{n}$,$\widetilde{m}$]	m = $\widetilde{m}$; n = $\widetilde{n}$

- Need K st m − $\widetilde{m}$ and n − $\widetilde{n}$ wrt above tests

- Following are invariant

F_1 : ([j,k,$\widetilde{n}$,$\widetilde{m}$] exists) $\Rightarrow$ $\widetilde{m} \leq$ k.nGen

F_2 : ([j,k,$\widetilde{n}$,$\widetilde{m}$] exists) and ([.,k,j,m,n] rcvbl) $\Rightarrow$ m $\leq$ k.nGen

F_3 : ([j,k,$\widetilde{n}$,$\widetilde{m}$] exists) $\Rightarrow$ $\widetilde{n} \leq$ j.nGen

F_4 : ([j,k,$\widetilde{n}$,$\widetilde{m}$] exists) and ([.,k,j,m,n] rcvbl) $\Rightarrow$ n $\leq \widetilde{n}$ b

F_5 : ([j,k,$\widetilde{n}$,$\widetilde{m}$] opening) $\Rightarrow$ k.nGen $\leq \widetilde{m} + (L + 2W)/\delta$

F_6 : ([j,k,$\widetilde{n}$,$\widetilde{m}$] open) $\Rightarrow$ k.nGen $\leq \widetilde{m} + (C + L + 2W)/\delta$

- Following are invariant

G_1 : [j,k,$\widetilde{\mathtt{n}}$,$\widetilde{\mathtt{m}}$] opening and [.,k,j,m,n] rcvbl $\Rightarrow$
$(\mathtt{m} = -1$ or $\widetilde{\mathtt{m}} = -1$ or $\mathtt{m} \leq \widetilde{\mathtt{m}} + (L + 2W)/\delta)$

G_2 : ([j,k,$\widetilde{\mathtt{n}}$,$\widetilde{\mathtt{m}}$] open) and ([.,k,j,m,n] rcvbl) $\Rightarrow$
$\mathtt{m} \leq \widetilde{\mathtt{m}} + (C + L + 2W)/\delta$

G_3 : ([j,k,$\widetilde{\mathtt{n}}$,$\widetilde{\mathtt{m}}$] opening or open) and ([CCR|SCR,k,j,m,n] rcvbl) $\Rightarrow$
$\mathtt{m} \geq \widetilde{\mathtt{m}} - (L + W)/\delta$ and $\mathtt{n} \geq \widetilde{\mathtt{n}} - (2L + 2W)/\delta$

G_5 : ([j,k,$\widetilde{\mathtt{n}}$,$\widetilde{\mathtt{m}}$] opening) and ([CRACK,k,j,m,n] rcvbl) $\Rightarrow$
$\mathtt{m} \geq \widetilde{\mathtt{m}} - (2L + 2W)/\delta$ and $\mathtt{n} \geq \widetilde{\mathtt{n}} - (2L + W)/\delta$

G_6 : [j,k,$\widetilde{\mathtt{n}}$,$\widetilde{\mathtt{m}}$] opening and [REJ,k,j,m,n] rcvbl $\Rightarrow$
$\mathtt{m} \geq \widetilde{\mathtt{m}} - (3L + 2W)/\delta$ and $\mathtt{n} \geq \widetilde{\mathtt{n}} - (2L + W)/\delta$

G_7 : [j,k,$\widetilde{\mathtt{n}}$,$\widetilde{\mathtt{m}}$] opening or open and [EDT,k,j,m,n,.] rcvbl $\Rightarrow$
$\mathtt{m} \geq \widetilde{\mathtt{m}} - (L + C + W)/\delta$ and $\mathtt{n} \geq \widetilde{\mathtt{n}} - (L + C + 2W)/\delta$

- From the above, the following hold

$H_1 : \mathtt{m} \leq \widetilde{\mathtt{m}} + (C + L + 2W)/\delta$ // G_1, G_2

$H_2 : \mathtt{m} \geq \widetilde{\mathtt{m}} - \max(3L + 2W, L + C + W)/\delta$ // G_3–G_7

$H_3 : \mathtt{n} \leq \widetilde{\mathtt{n}}$ // F_4

$H_4 : \mathtt{n} \geq \widetilde{\mathtt{n}} - \max(2L + 2W, L + C + 2W)/\delta$ // G_3–G_7

$H_5 : -\max(3L+2W, L+C+W)/\delta \leq \mathtt{m}-\widetilde{\mathtt{m}} \leq (C+L+2W)/\delta$ // H_1, H_2

$H_6 : -\max(2L + 2W, L + C + 2W)/\delta \leq \mathtt{n} - \widetilde{\mathtt{n}} \leq \widetilde{\mathtt{n}}$ // H_3, H_4

$H_7 : K \leq \mathtt{m} - \widetilde{\mathtt{m}}, \mathtt{n} - \widetilde{\mathtt{n}} \leq K$, where $K = (3L + 2W + C)/\delta$ // H_5, H_6

Tp with modulo-N ep #s

- Ep #s in msgs and vars now range over $-1..N-1$
- Optional: ngen is now modulo-N
- Tests involving these values are now as follows
-

Old test	New test
m = rn[k]	no change
n = ln[k]	no change
m > ccrBuff[k]	$1 \leq \text{mod}(m - \text{ccrBuff[a1]}) \leq N/2$
m > rn[k]	(rn[k] = −1 and m ≠ −1) or $1 \leq \text{mod}(m - \text{rn[k]}) \leq N/2$

Outline

Overview

Graceful-closing data transfer protocol

Transport protocol description

Transport protocol program: unbounded endpoint numbers

Transport protocol program: cyclic endpoint numbers

Transport protocol with abort

Extending Tp with aborts

- Tp now aborts endpoint if response to a primary message not rcvd after K resends
  - returns of functions distinguish between closing (or rejection) and abort
  - use the abortable dtp program and service