COMP 431
Internet Services & Protocols

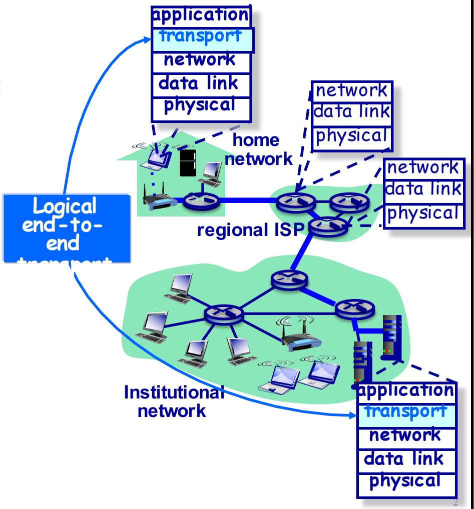
The Transport Layer
Principles of Reliable Data Delivery

Jasleen Kaur

Feb 20, 2020

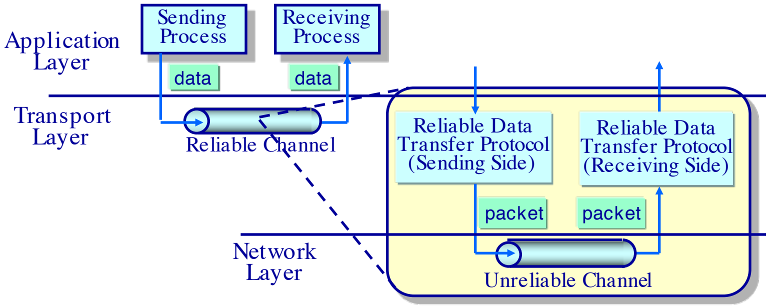
Transport Layer Protocols & Services
Outline

- Fundamental transport layer services
  - » Multiplexing/Demultiplexing
  - » Error detection
  - » Reliable data delivery
  - » Pipelining
  - » Flow control
  - » Congestion control
- Service implementation in Internet transport protocols
  - » UDP
  - » TCP



Fundamental Transport Layer Services

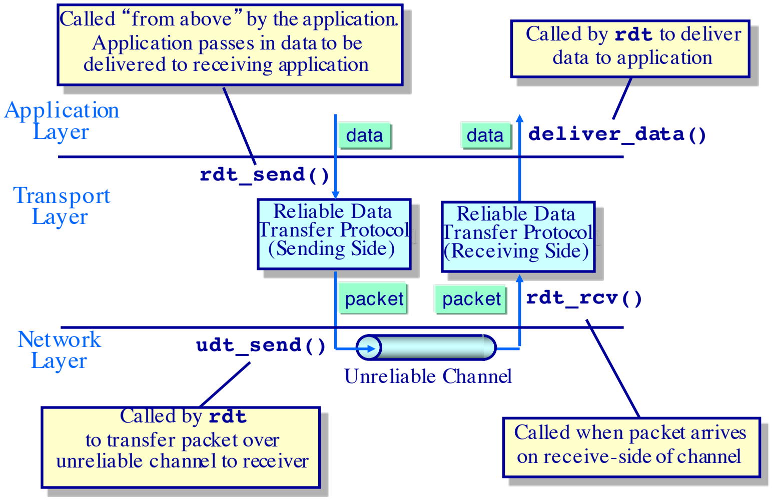
Principles of reliable data transfer



- Goal: Provide a reliable channel abstraction
  - » The characteristics of the underlying channel will determine the complexity of providing reliable communications
- Issues: *State* required at sender and receiver and number of *control messages* exchanged

Reliable Data Transfer

Programming interfaces



Called "from above" by the application. Application passes in data to be delivered to receiving application

Called by **`rdt`** to deliver data to application

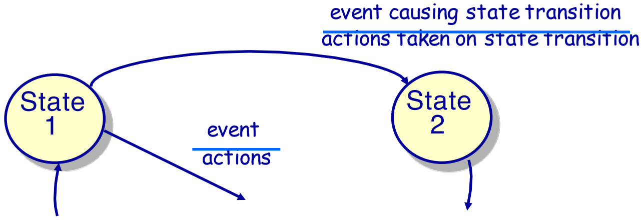
Called by **`rdt`** to transfer packet over unreliable channel to receiver

Called when packet arrives on receive-side of channel

Reliable Data Transfer

Protocol specification method

- Use finite state machines to specify sender and receiver algorithms
  - » When in a given state, the next state (and actions) are uniquely determined by the next event



event causing state transition

actions taken on state transition

State 1

State 2

event

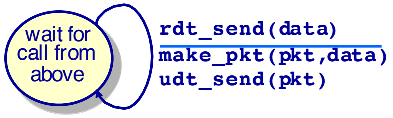
actions

Reliable Data Transfer Protocol 1.0

Reliable transfer over a reliable channel

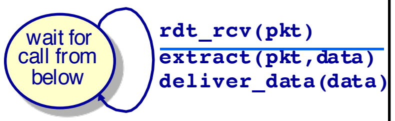
- The underlying channel is assumed to be perfectly reliable
  - » No bit errors
  - » No loss of packets

- Sender state machine
- Receiver state machine



wait for call from above

```
rdt_send(data)
make_pkt(pkt,data)
udt_send(pkt)
```

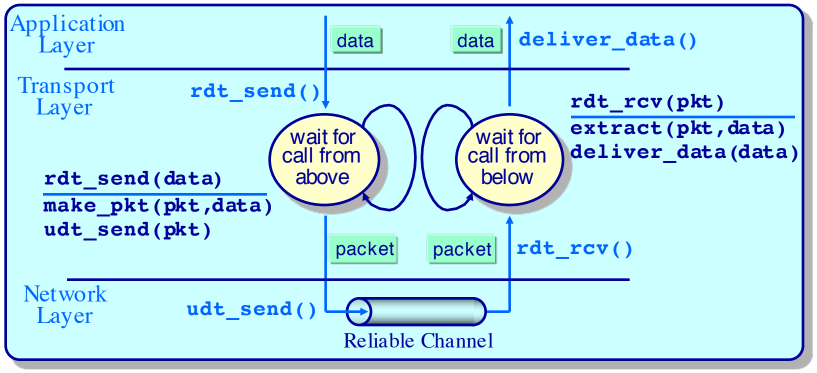


wait for call from below

```
rdt_rcv(pkt)
extract(pkt,data)
deliver_data(data)
```

Reliable Data Transfer Protocol 1.0

Programming interfaces



- This is the complete protocol under the assumption of a reliable network channel

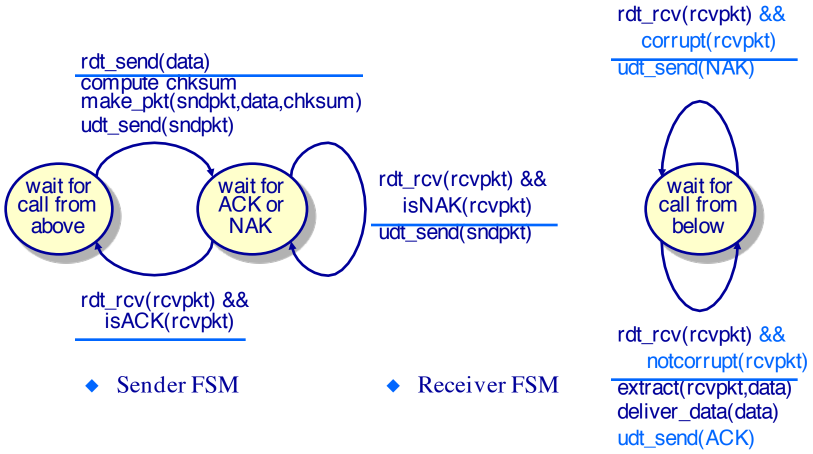
Reliable Data Transfer Protocol 2.0

Reliable transfer over a channel with bit errors

- Now assume the underlying channel may “flip” random bits in a packet
- How to detect errors?
- How to recover from errors:
  - » *acknowledgements (ACKs)* — the receiver explicitly tells the sender that a packet was received OK
  - » *negative acknowledgements (NAKs)* — the receiver explicitly tells the sender that a packet had errors
  - » Sender retransmits packet on receipt of NAK
- New mechanisms to deal with bit errors:
  - » Error detection
  - » Control messages (ACK, NAK) from a receiver to the sender
  - » Retransmission

Reliable Data Transfer Protocol 2.0

Reliable transfer over a channel with bit errors only



rdt_send(data)
compute chksum
make_pkt(sndpkt,data,chksum)
udt_send(sndpkt)

wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)

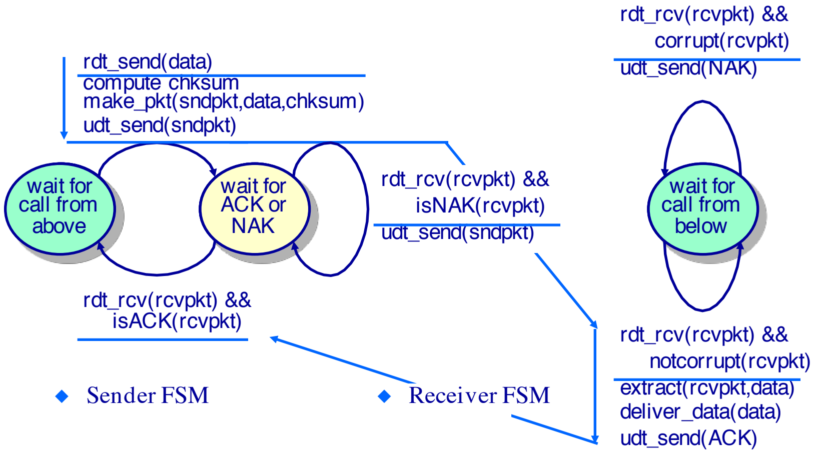
wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

- Sender FSM
- Receiver FSM

Reliable Data Transfer Protocol 2.0

Example 1: No errors occur



rdt_send(data)
compute chksum
make_pkt(sndpkt,data,chksum)
udt_send(sndpkt)

wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)

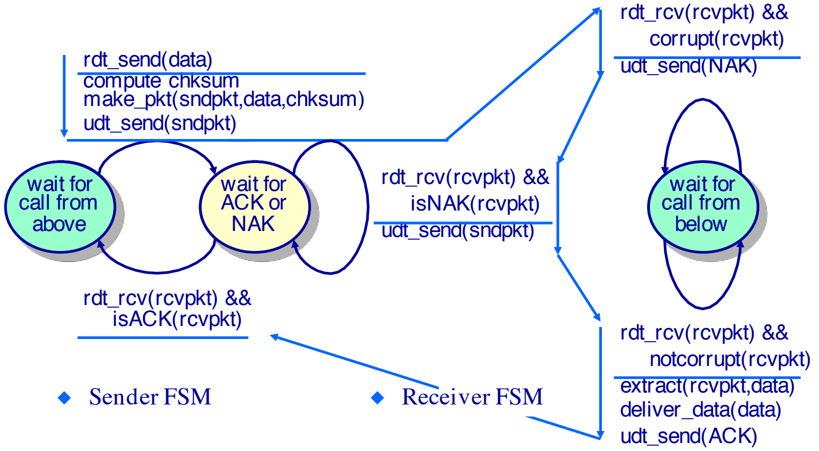
wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

- Sender FSM
- Receiver FSM

Reliable Data Transfer Protocol 2.0

Example 2: A corrupted packet arrives at the receiver



rdt_send(data)

compute chksum
make_pkt(sndpkt,data,chksum)
udt_send(sndpkt)

wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isACK(rcvpkt)

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

wait for call from below

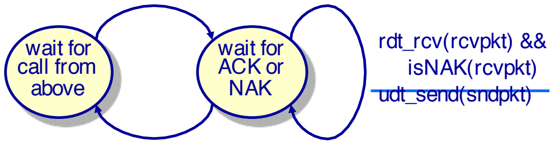
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

- Sender FSM
- Receiver FSM

Reliable Data Transfer Protocol 2.0

Simple… but wrong!

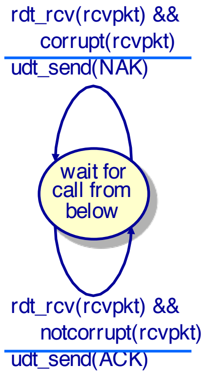


wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)



rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

wait for call from below

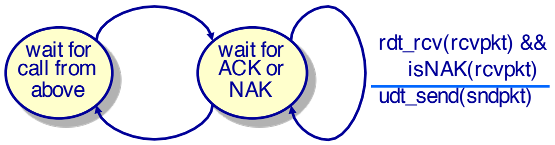
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

udt_send(ACK)

- What happens if an ACK/NAK is corrupted?
  - » Sender doesn’t know what happened at the receiver!
- What to do?
  - » Sender ACKs/NAKs the receiver’s ACK/NAK?
  - » Retransmit last data packet?
    - ❖ How to deal with duplicates?

Reliable Data Transfer Protocol 2.0

Simple… but wrong!

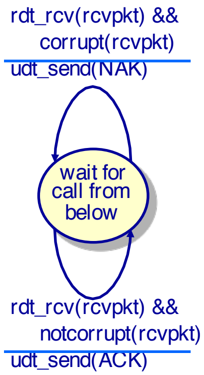


wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)



rdt_rcv(rcvpkt) && corrupt(rcvpkt)

udt_send(NAK)

wait for call from below

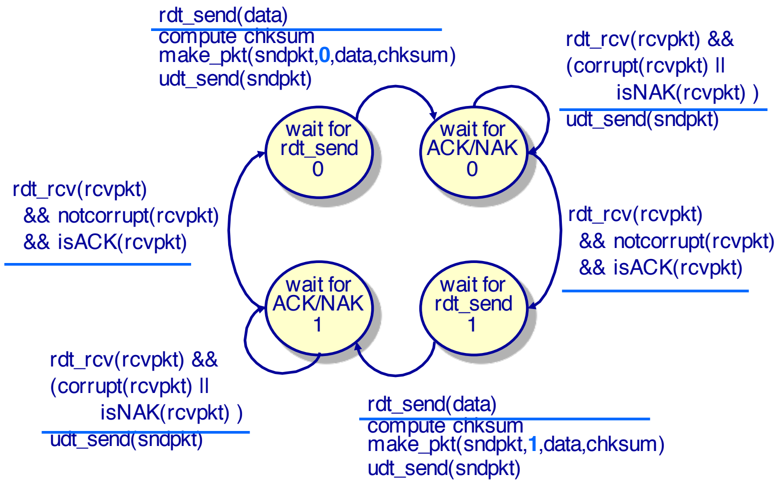
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

udt_send(ACK)

- Deal with corrupted ACKs/NAKs by retransmission of data packets
- Sender will add a sequence number to each packet to allow the receiver to detect duplicate packets
  - » Receiver's transport layer discards duplicate packets
- How much space to reserve in a header field for sequence numbers?

Reliable Data Transfer Protocol 2.1

Sender state machine to handle garbled ACKs/NAKs



rdt_send(data)

compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

udt_send(sndpkt)

wait for rdt_send 0

wait for ACK/NAK 0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)

wait for ACK/NAK 1

wait for rdt_send 1

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))

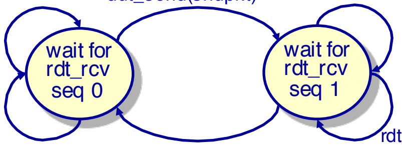
udt_send(sndpkt)

rdt_send(data)

compute chksum
make_pkt(sndpkt,1,data,chksum)
udt_send(sndpkt)

Reliable Data Transfer Protocol 2.1

Receiver state machine to handle garbled ACKs/NAKs



rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **has_seq0**(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sndpkt,ACK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

compute chksum
make_pkt(sndpkt,NAK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

compute chksum
make_pkt(sndpkt,NAK,
chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **has_seq1**(rcvpkt)

compute chksum
make_pkt(sndpkt,ACK,
chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **has_seq1**(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sndpkt,ACK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **has_seq0**(rcvpkt)

compute chksum
make_pkt(sndpkt,ACK,
chksum)
udt_send(sndpkt)

Reliable Data Transfer Protocol 2.1

Discussion (Handling garbled ACKs/NAKs)

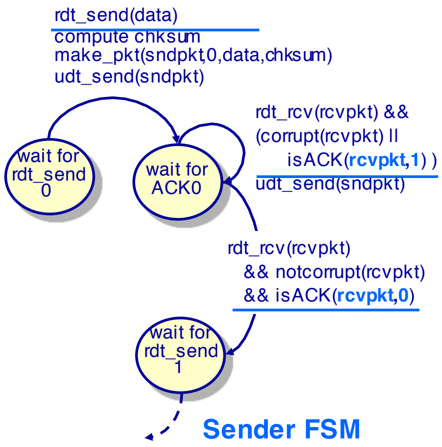
- Sender issues
  - Sequence number added to header
    - » Two sequence numbers suffice
  - Must check if received ACK/NAK is corrupted
  - Number of states doubles
    - » State encodes whether current packet has sequence number 0 or 1

- Receiver issues
  - Must check if received packet is duplicate
    - » State encodes whether expected packet sequence number is 0 or 1
  - Note: receiver can *not* know if its last ACK/NAK received OK at sender

Reliable Data Transfer Protocol 2.2

A NAK-free protocol

- Instead of NAKing, receiver sends ACK for last packet received OK
  - » Receiver must include the sequence number of packet being ACKed in ACK
- Receipt of duplicate ACKs at sender is equivalent to a NAK
  - » Sender retransmits current packet



rdt_send(data)

compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(**rcvpkt,1**))

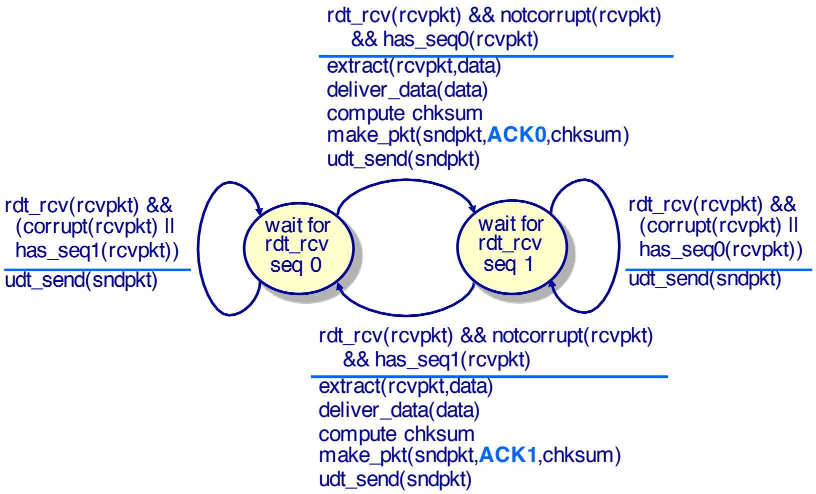
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(**rcvpkt,0**)

Sender FSM

Reliable Data Transfer Protocol 2.2

Receiver state machine to eliminate NAKs



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sndpkt,**ACK0**,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq0(rcvpkt))

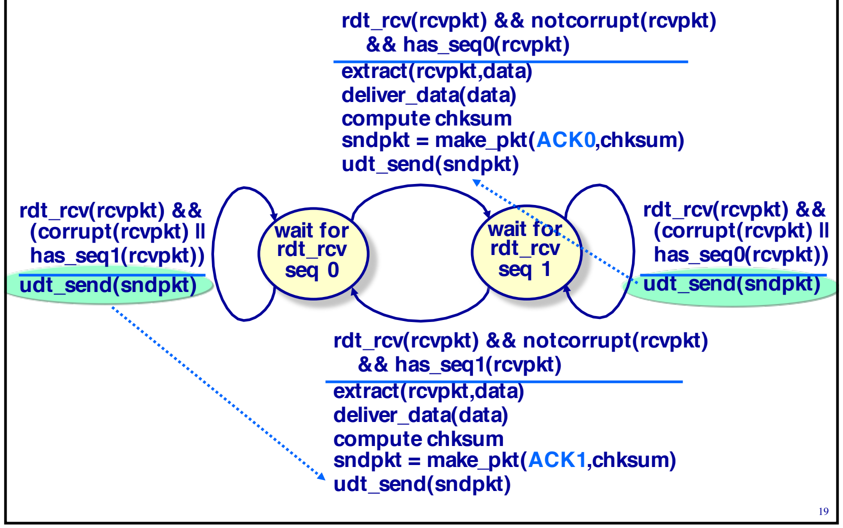
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sndpkt,**ACK1**,chksum)
udt_send(sndpkt)

Reliable Data Transfer Protocol 2.2

A buggy receiver state machine?



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
sndpkt = make_pkt(ACK0,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ll
has_seq1(rcvpkt))

udt_send(sndpkt)

wait for
rdt_rcv
seq 0

wait for
rdt_rcv
seq 1

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ll
has_seq0(rcvpkt))

udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
sndpkt = make_pkt(ACK1,chksum)
udt_send(sndpkt)

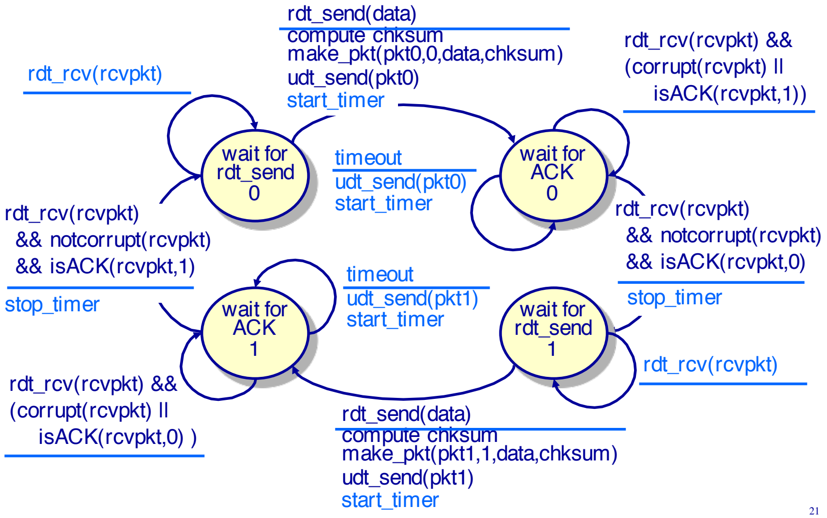
Reliable Data Transfer Protocol 3.0

Dealing with channels with errors *and* loss

- Now assume the underlying channel can also **lose** packets
- New problem: How to detect loss?
  - » Are checksums, ACKs, sequence numbers, retransmissions enough?
- Approach: sender waits “reasonable” amount of time and retransmits if no ACK received in this time
  - » Requires the use of a countdown timer
- What if packet (or ACK) just delayed beyond its timer?
  - » Retransmission will be duplicate…
  - » But use of sequence numbers already handles this!

Reliable Data Transfer Protocol 3.0

Sender state machine to handle lost/garbled packets



rdt_send(data)

compute chksum
make_pkt(pkt0,0,data,chksum)
udt_send(pkt0)
start_timer

rdt_rcv(rcvpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ll
isACK(rcvpkt,1))

wait for
rdt_send
0

timeout

udt_send(pkt0)
start_timer

wait for
ACK
0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)

stop_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)

stop_timer

wait for
ACK
1

timeout

udt_send(pkt1)
start_timer

wait for
rdt_send
1

rdt_rcv(rcvpkt)

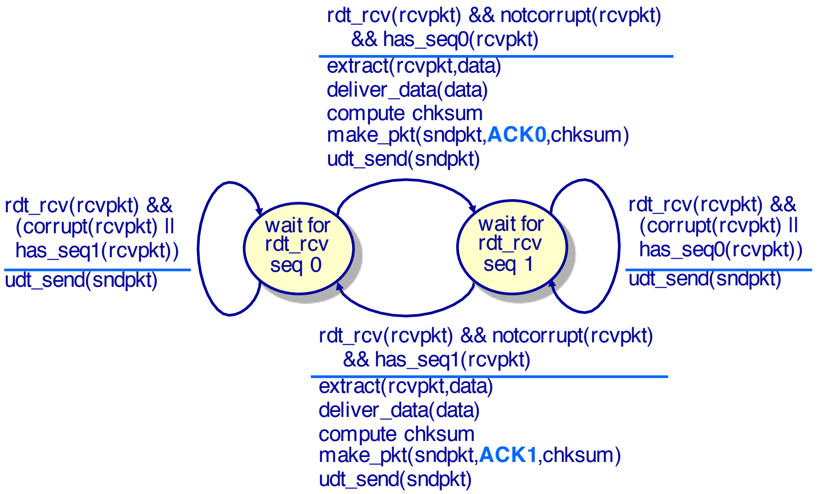
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ll
isACK(rcvpkt,0))

rdt_send(data)

compute chksum
make_pkt(pkt1,1,data,chksum)
udt_send(pkt1)
start_timer

Receiver State Machine for RDT 2.2

What changes are needed to handle lost/garbled packets?



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sndpkt,**ACK0**,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ll
has_seq1(rcvpkt))

udt_send(sndpkt)

wait for
rdt_rcv
seq 0

wait for
rdt_rcv
seq 1

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ll
has_seq0(rcvpkt))

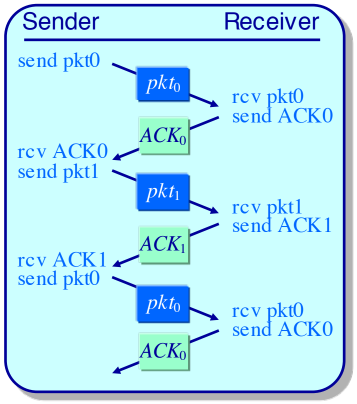
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sndpkt,**ACK1**,chksum)
udt_send(sndpkt)

Reliable Data Transfer Protocol 3.0

Execution examples



Sender Receiver

send pkt0

pkt_0

rcv pkt0
send ACK0

ACK_0

rcv ACK0
send pkt1

pkt_1

rcv pkt1
send ACK1

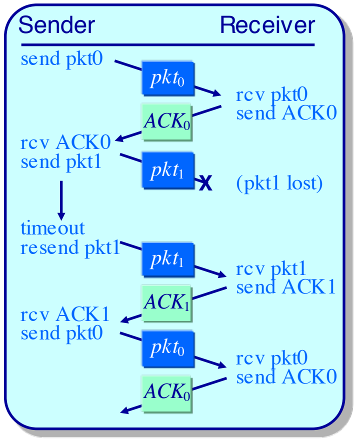
ACK_1

rcv ACK1
send pkt0

pkt_0

rcv pkt0
send ACK0

ACK_0



Sender Receiver

send pkt0

pkt_0

rcv pkt0
send ACK0

ACK_0

rcv ACK0
send pkt1

pkt_1

X

(pkt1 lost)

timeout
resend pkt1

pkt_1

rcv pkt1
send ACK1

ACK_1

rcv ACK1
send pkt0

pkt_0

rcv pkt0
send ACK0

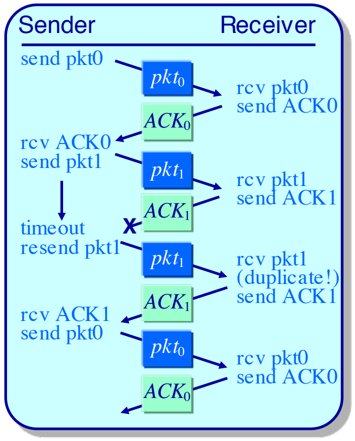
ACK_0

- Protocol operation with no loss

- Protocol operation with a lost packet

Reliable Data Transfer Protocol 3.0

Execution examples



Sender Receiver

send pkt0

pkt_0

rcv pkt0
send ACK0

ACK_0

rcv ACK0
send pkt1

pkt_1

rcv pkt1
send ACK1

ACK_1

X

timeout
resend pkt1

pkt_1

rcv pkt1
(duplicate!)
send ACK1

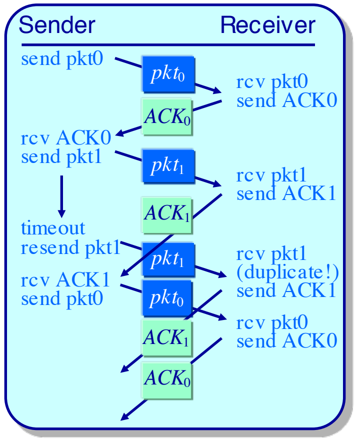
ACK_1

rcv ACK1
send pkt0

pkt_0

rcv pkt0
send ACK0

ACK_0



Sender Receiver

send pkt0

pkt_0

rcv pkt0
send ACK0

ACK_0

rcv ACK0
send pkt1

pkt_1

rcv pkt1
send ACK1

ACK_1

timeout
resend pkt1

pkt_1

rcv ACK1
send pkt0

rcv pkt1
(duplicate!)
send ACK1

pkt_0

rcv pkt0
send ACK0

ACK_1

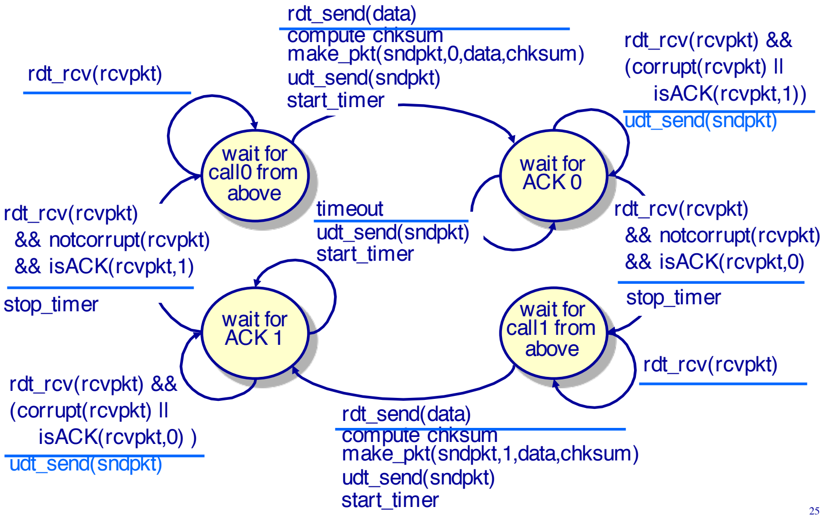
ACK_0

- Protocol operation with a lost ACK

- Protocol operation with a poor timeout value

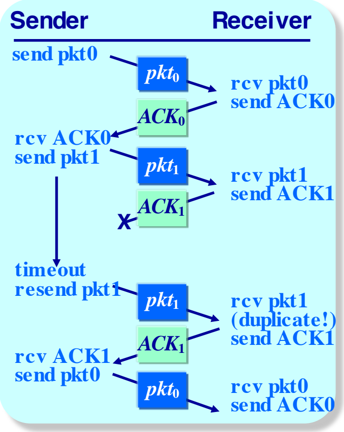
Reliable Data Transfer Protocol 3.1

Sender state machine to handle lost/garbled packets

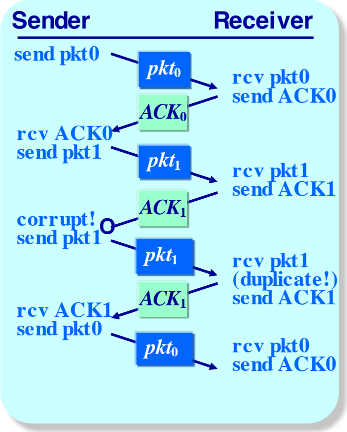


Reliable Data Transfer Protocol 3.1

Execution examples



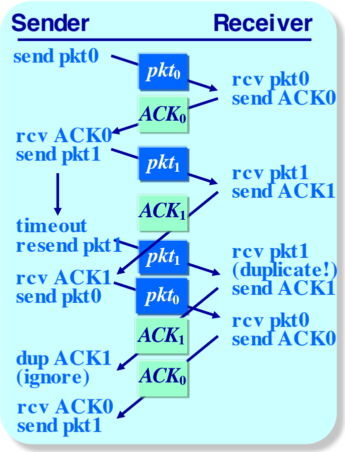
- Protocol operation with a lost ACK

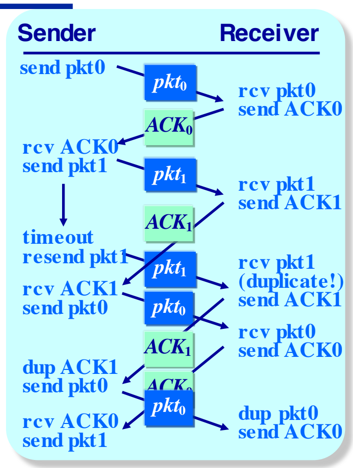


- Protocol operation with a corrupted ACK

Reliable Data Transfer Protocol 3.1

Execution examples

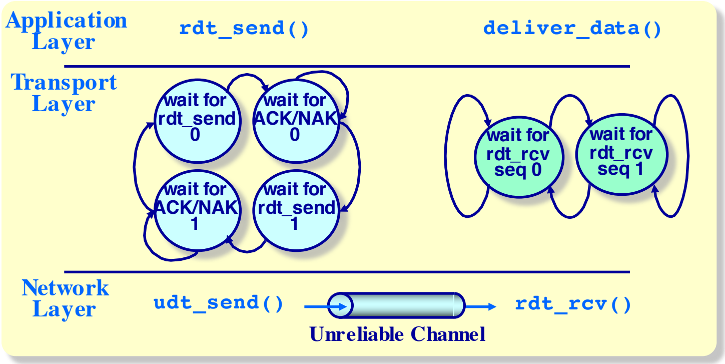




- **RDT 3.0 operation with a duplicate ACK**
- **RDT 3.1 operation with a duplicate ACK**

Reliable Data Delivery

Summary



- Reliable delivery requires a distributed algorithm for detecting and repairing packet errors and loss
  - » Reliability achieved at the expense of considerable overhead and complexity