National University of Computer & Emerging Sciences

CS 3001 – COMPUTER NETWORKS

Lecture 10
Chapter 3

21st September, 2023

Nauman Moazzam Hayat
nauman.moazzam@lhr.nu.edu.pk

Office Hours: 02:30 pm till 06:00 pm (Every Tuesday & Thursday)

Chapter 3: roadmap

UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol

- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app

- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP: User Datagram Protocol

- UDP use:
    - streaming multimedia apps (loss tolerant, rate sensitive)
    - DNS
    - SNMP
    - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
    - add needed reliability at application layer
    - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

```
                                                      INTERNET STANDARD
RFC 768                                                       J. Postel
                                                                     ISI
                                                         28 August 1980


                        User Datagram Protocol
                        ----------------------

Introduction
------------

This User Datagram  Protocol  (UDP)  is  defined  to  make  available  a
datagram   mode  of  packet-switched   computer   communication  in  the
environment  of  an  interconnected  set  of  computer  networks.   This
protocol  assumes  that the Internet  Protocol  (IP) [1] is used as the
underlying protocol.

This protocol  provides  a procedure  for application  programs  to send
messages  to other programs  with a minimum  of protocol mechanism.  The
protocol  is transaction oriented, and delivery and duplicate protection
are not guaranteed.  Applications requiring ordered reliable delivery of
streams of data should use the Transmission Control Protocol (TCP) [2].

Format
------


     0      7 8     15 16    23 24    31
    +--------+--------+--------+--------+
    |     Source      |   Destination   |
    |      Port       |      Port       |
    +--------+--------+--------+--------+
    |                 |                 |
    |     Length      |    Checksum     |
    +--------+--------+--------+--------+
    |
    |          data octets ...
    +---------------- ...
```
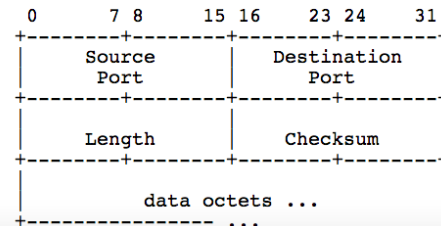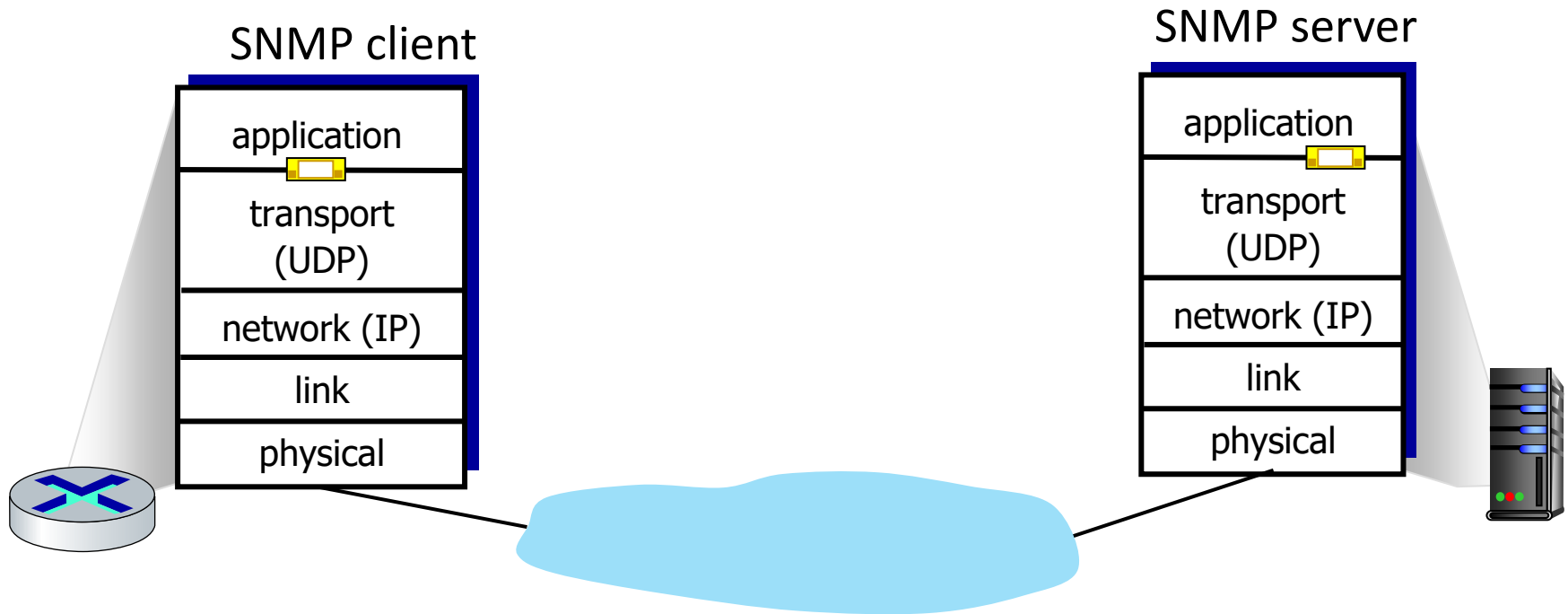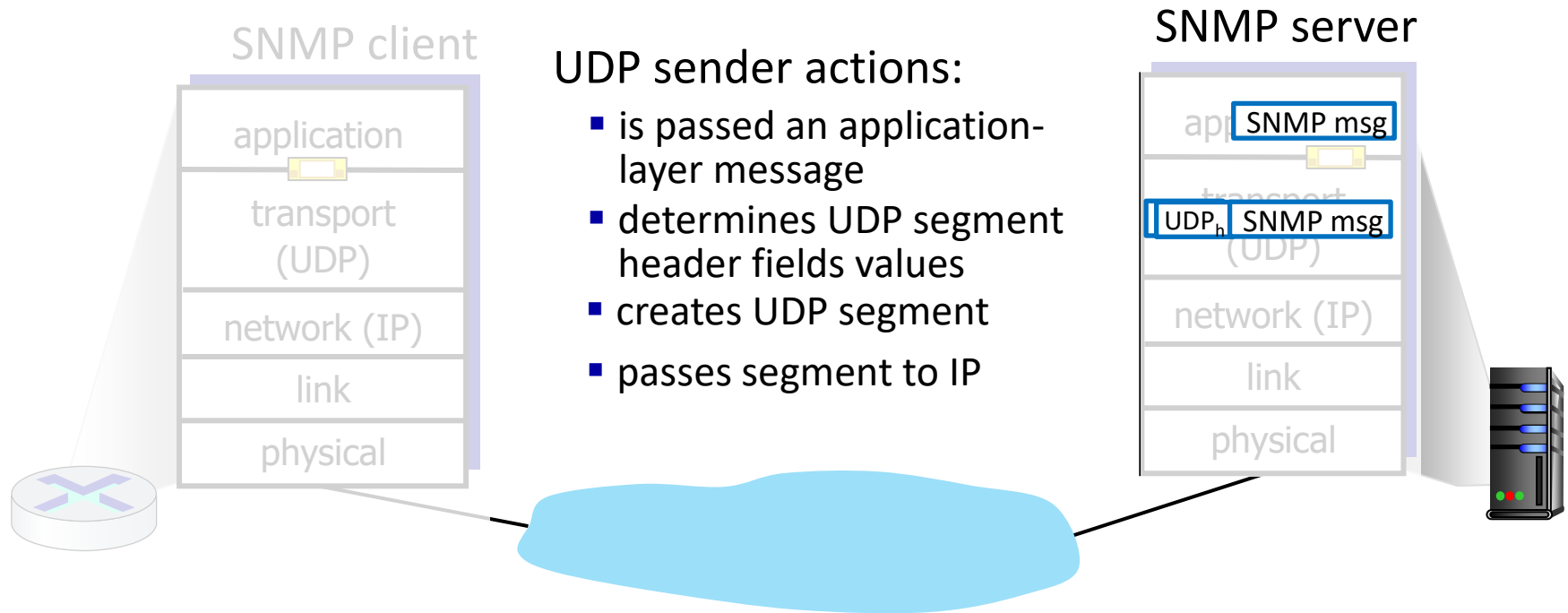
UDP: Transport Layer Actions

SNMP client

SNMP server

application	
transport (UDP)	
network (IP)	
link	
physical	

application	
transport (UDP)	
network (IP)	
link	
physical	

UDP: Transport Layer Actions

SNMP client

SNMP server

application
transport (UDP)
network (IP)
link
physical

UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

application	SNMP msg
transport (UDP)	UDP$_h$ SNMP msg
network (IP)	
link	
physical	

UDP: Transport Layer Actions

SNMP client

SNMP server

application

transport
(UDP)

SNMP msg

UDP_h | SNMP msg

link

physical

application

transport
(UDP)

network (IP)

link

physical

UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

UDP segment header

32 bits

source port #	dest port #
length	checksum

application
data
(payload)

length, in bytes of
UDP segment,
including header

data to/from
application layer

UDP segment format

UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum ≠ sender-computed checksum (as received)

Internet checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

sender:
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an example

example: add two 16-bit integers

```
           1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
           1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
          ─────────────────────────────────
wraparound (1) 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
          ─────────────────────────────────
     sum   1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum   0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

example: add two 16-bit integers

```
          1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0        0 1
          1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1        1 0
```

wraparound 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum   1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0

checksum   0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

Even though numbers have changed (bit flips), *no* change in checksum!

Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Chapter 3
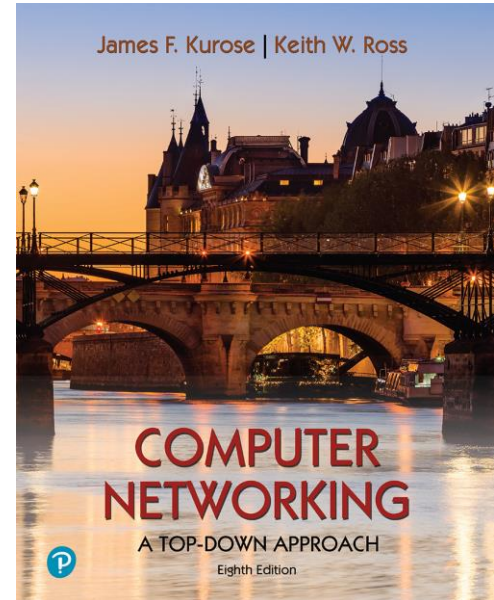# Transport Layer

A note on the use of these PowerPoint slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top-Down Approach
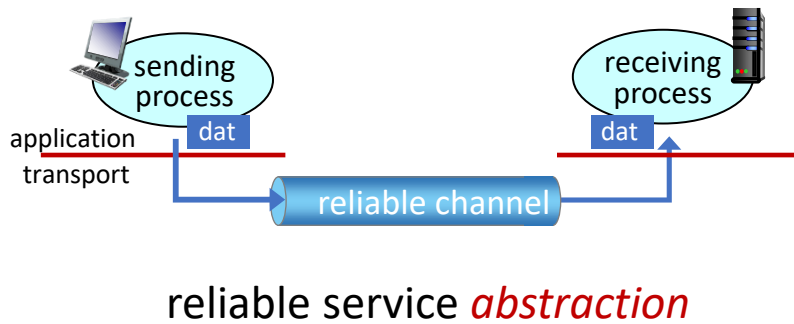8th edition
Jim Kurose, Keith Ross
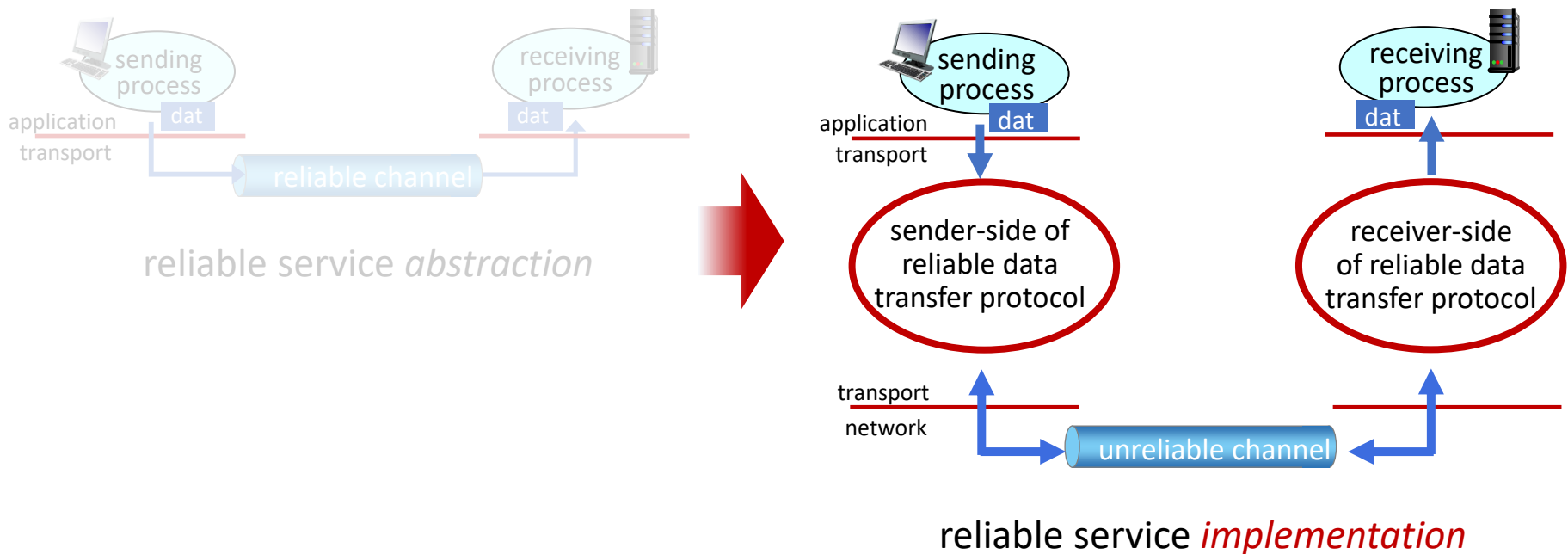Pearson, 2020

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
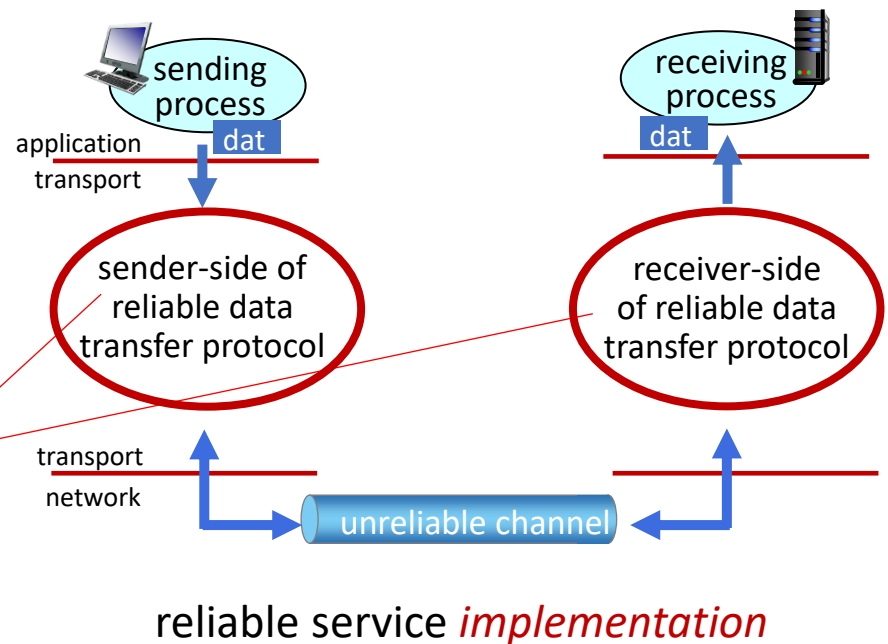
Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

reliable service *abstraction*

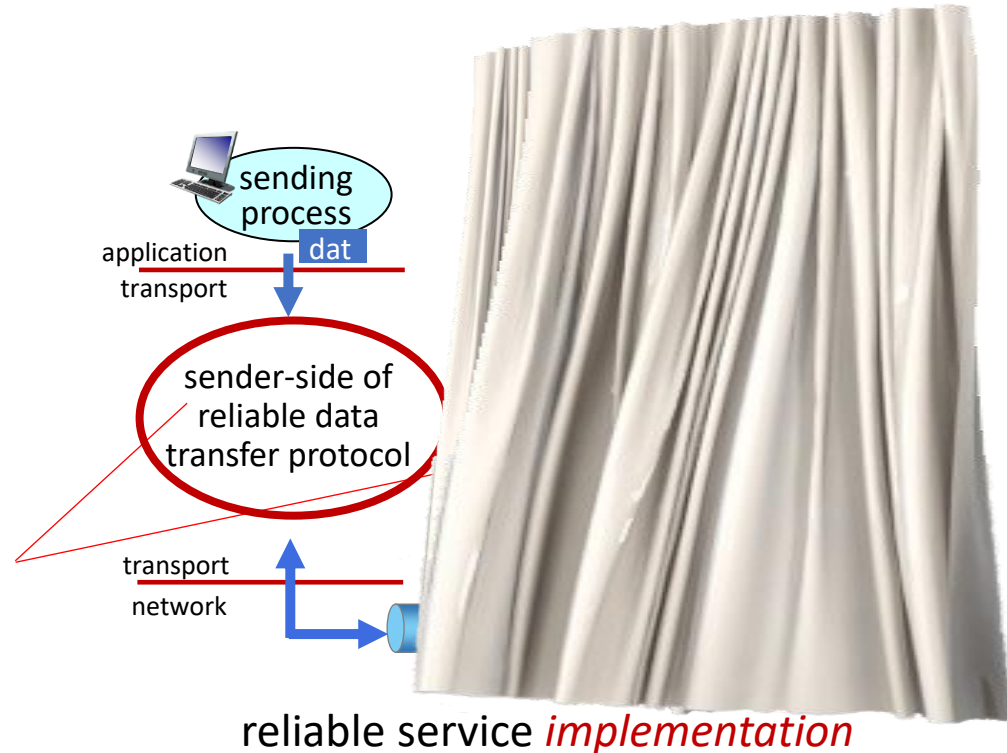reliable service *implementation*

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)
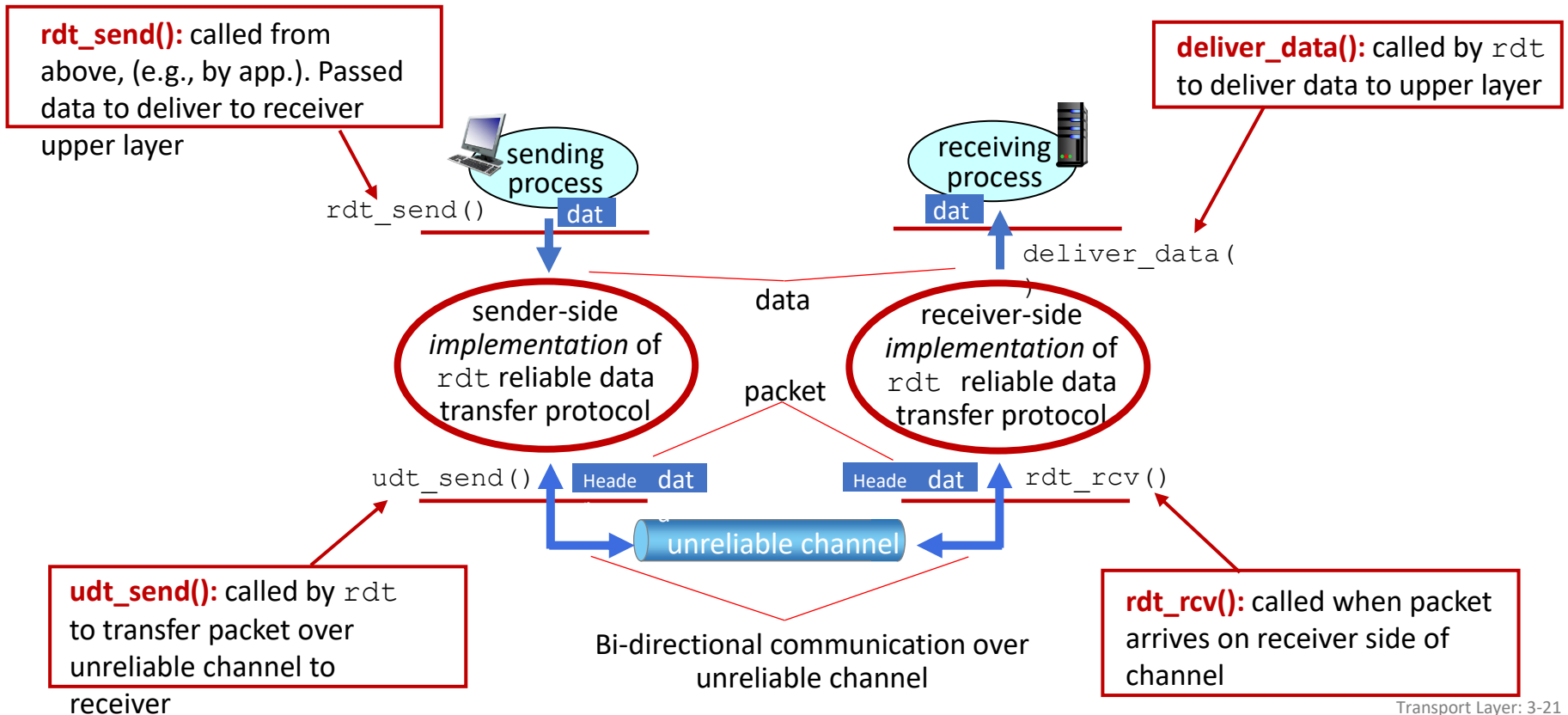
reliable service *implementation*

Principles of reliable data transfer

Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

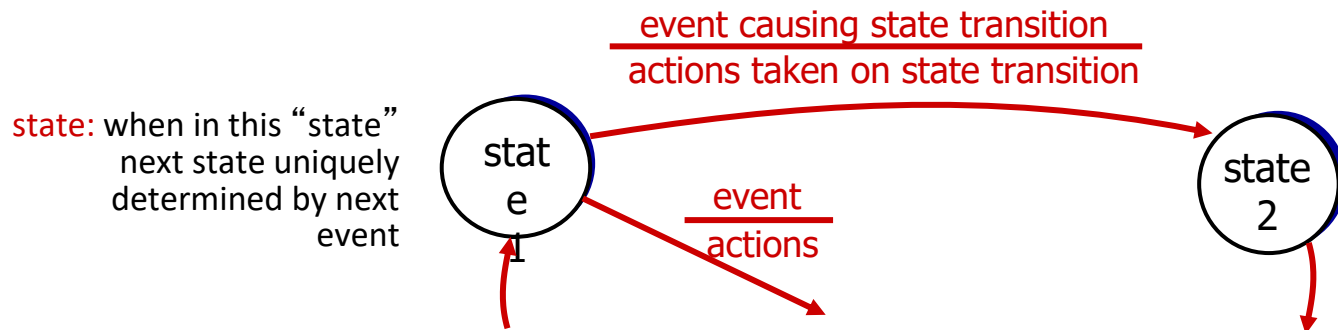- unless communicated via a message

reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer

udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on receiver side of channel

Bi-directional communication over unreliable channel

Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!

- use finite state machines (FSM) to specify sender, receiver

event causing state transition

actions taken on state transition

state: when in this "state"
next state uniquely
determined by next
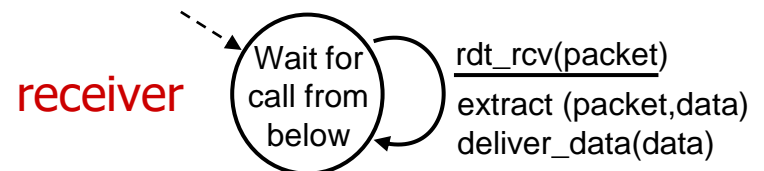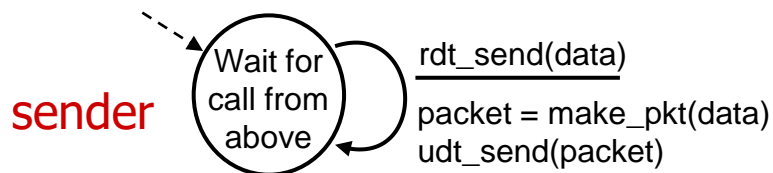event

state 1

event
actions

state 2

Finite State Machine (FSM)

- *Or simply a State Machine*
- *It is a mathematical model of computation.*
- *It is an abstract machine that can be in exactly one of a finite number of states at any given time.*
- *The FSM can change from one state to another in response to some event / input*
- *The change from one state to another is called a transition*
- *An FSM is defined by:*
  - *a list of its states*
  - *its initial state*
  - *and the event / input that trigger each transition*
  - *& the resultant actions that happen in response to the event*
- *Examples are vending machines, elevators, traffic signals, turnstile*
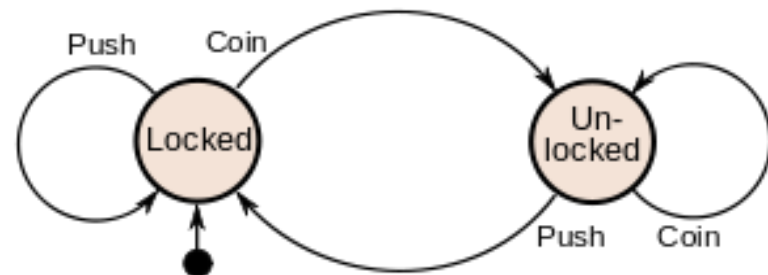
State Diagram of a Turnstile

rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- *separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

Wait for call from above

rdt_send(data)
—————————
packet = make_pkt(data)
udt_send(packet)

receiver

Wait for call from below

rdt_rcv(packet)
—————————
extract (packet,data)
deliver_data(data)

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

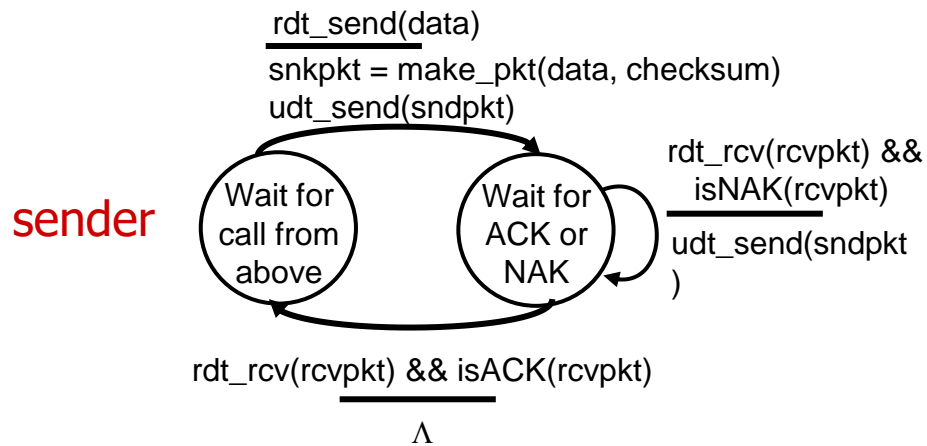*How do humans recover from "errors" during conversation?*

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors

- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender *retransmits* pkt on receipt of NAK
  - new mechanisms in `rdt2.0` (three new functionalities beyond `rdt1.0`):
    - error detection (e.g. checksum)
    - feedback: control msgs (ACK,NAK) from receiver to sender
    - retransmission: A packet arriving in error at receiver is retransmitted by the sender.

> **stop and wait**
>
> sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications

sender

rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

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

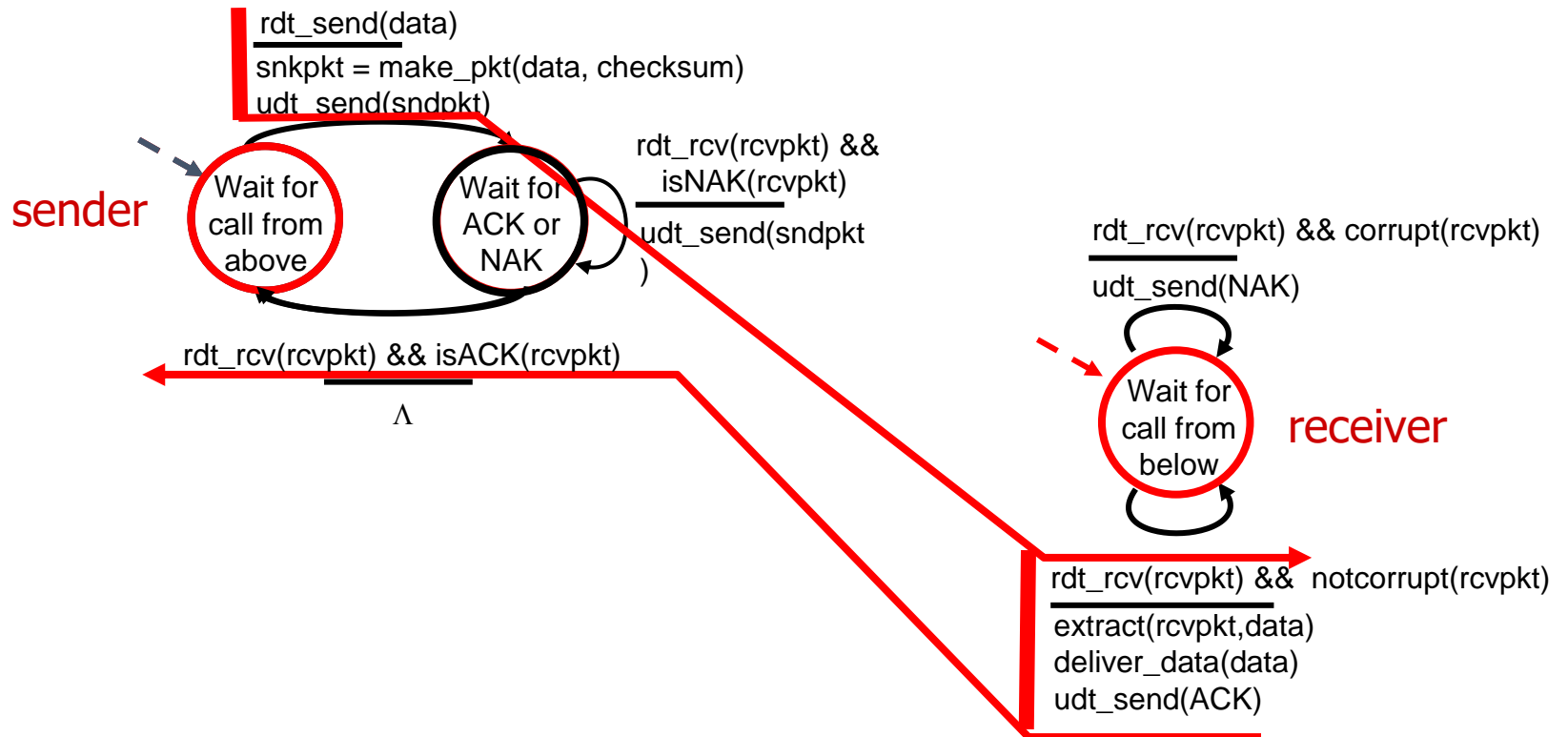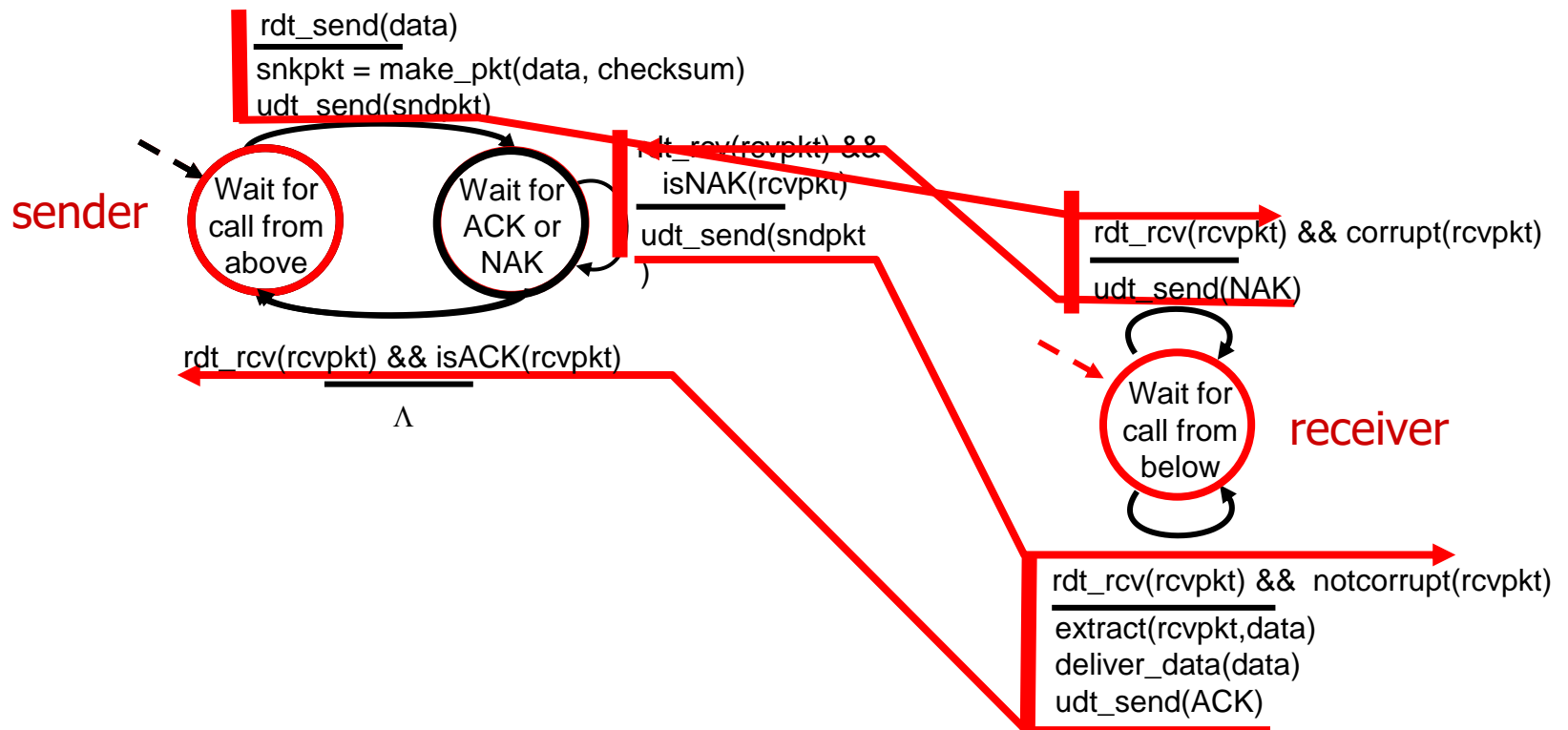rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

rdt2.0: FSM specification
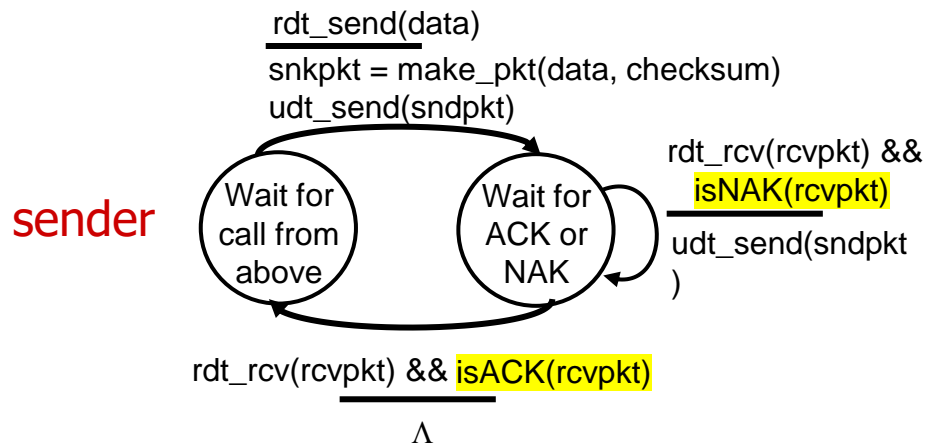
sender

Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender
  ▪ that's why we need a protocol!

rdt2.0: operation with no errors

rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

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

rdt2.0: corrupted packet scenario

sender

Wait for call from above

Wait for ACK or NAK

rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

udt_send(NAK)

receiver

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

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

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

Quiz 2 – Chapter 2