

# Internet Technology

## 05. Transport Layer

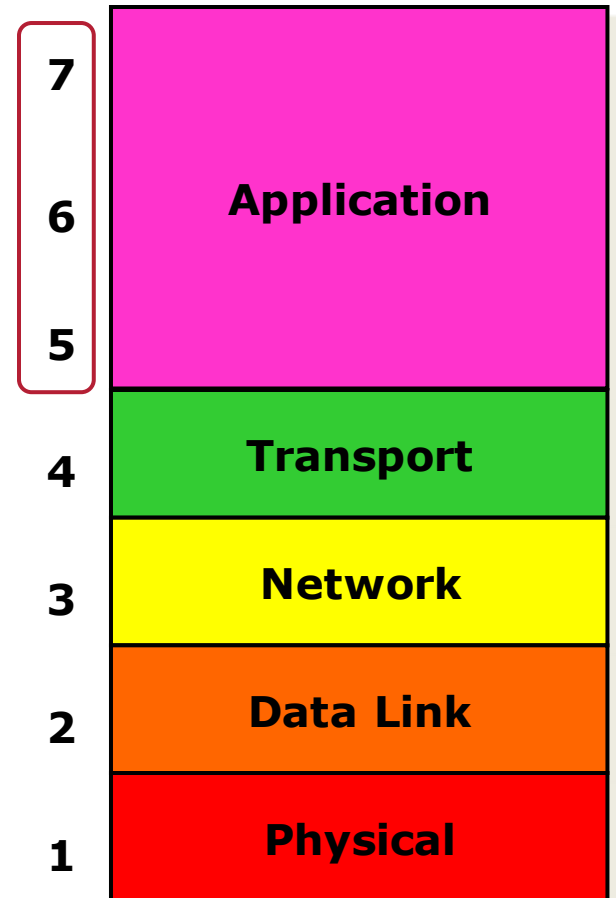
Paul Krzyzanowski

Rutgers University

Spring 2016

# Transport Layer

- Transport Layer
  - Provides logical communication channels between apps
- Transport layer managed by end systems
  - Routers are unaware; they provide network layer services
- Multiple transport protocols available
  - Under IP: TCP, UDP, SCTP, and more



Internet Protocol Layers

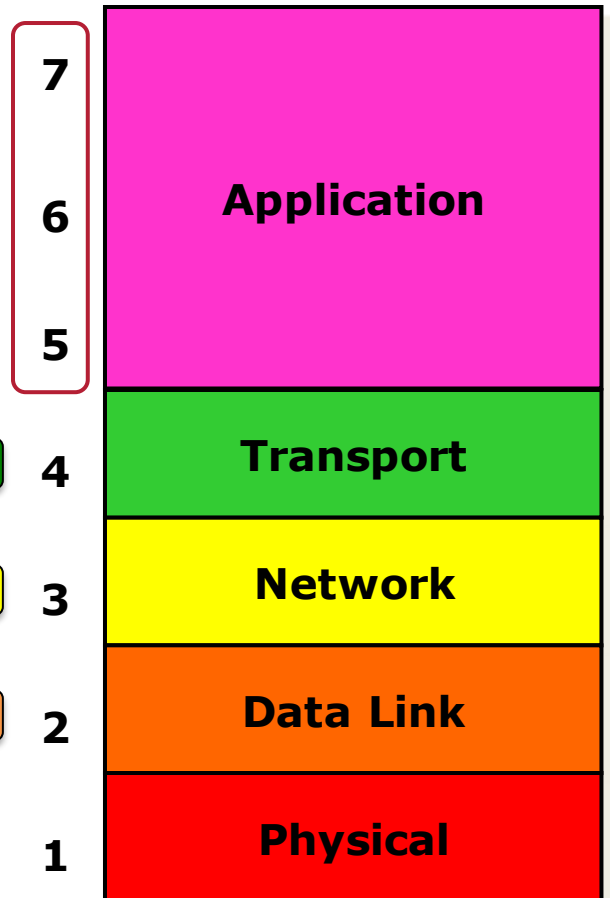
# Transport Layer

- Network Layer
  - Logical connection between hosts
- Transport Layer
  - Logical connection between processes
  - Transport layer **multiplexing** & **demultiplexing**
- Most common transport-layer protocols in IP:  
TCP & UDP
  - UDP: unreliable data transfer
  - TCP
    - Reliable data transfer
    - In-order delivery
    - Flow control
    - Congestion control

segments

datagrams

frames



Internet Protocol Layers

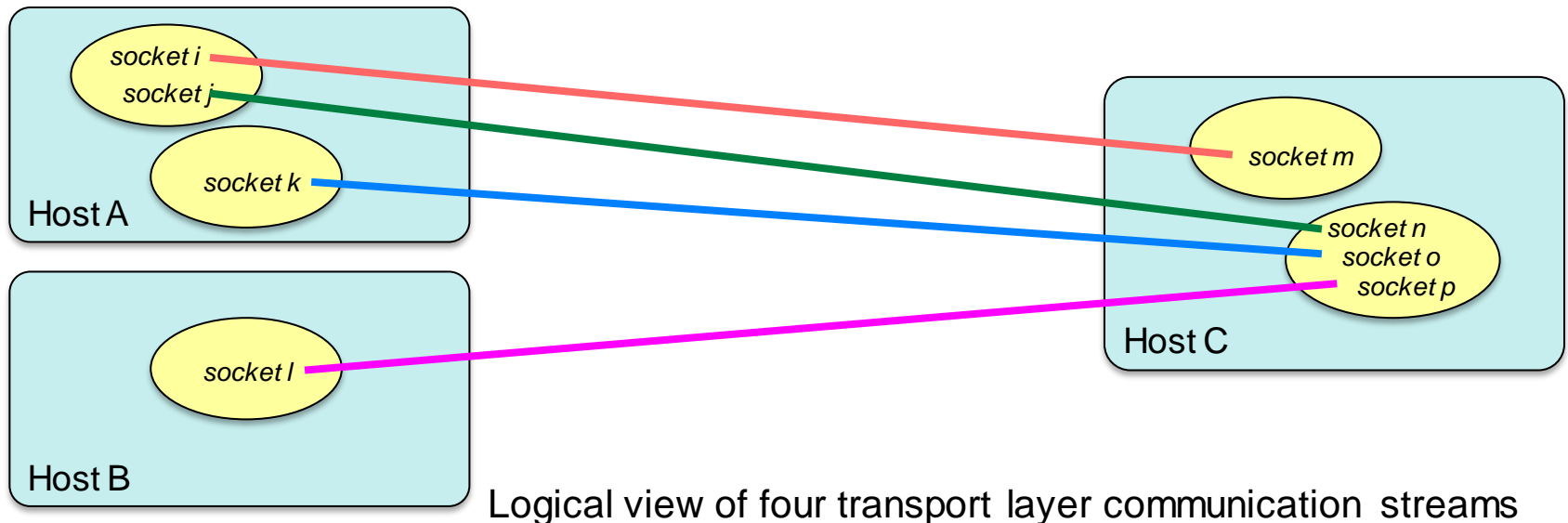
Today, we'll discuss

- Transport layer multiplexing/demultiplexing
- Reliable data transfer

# Transport Layer Multiplexing & Demultiplexing

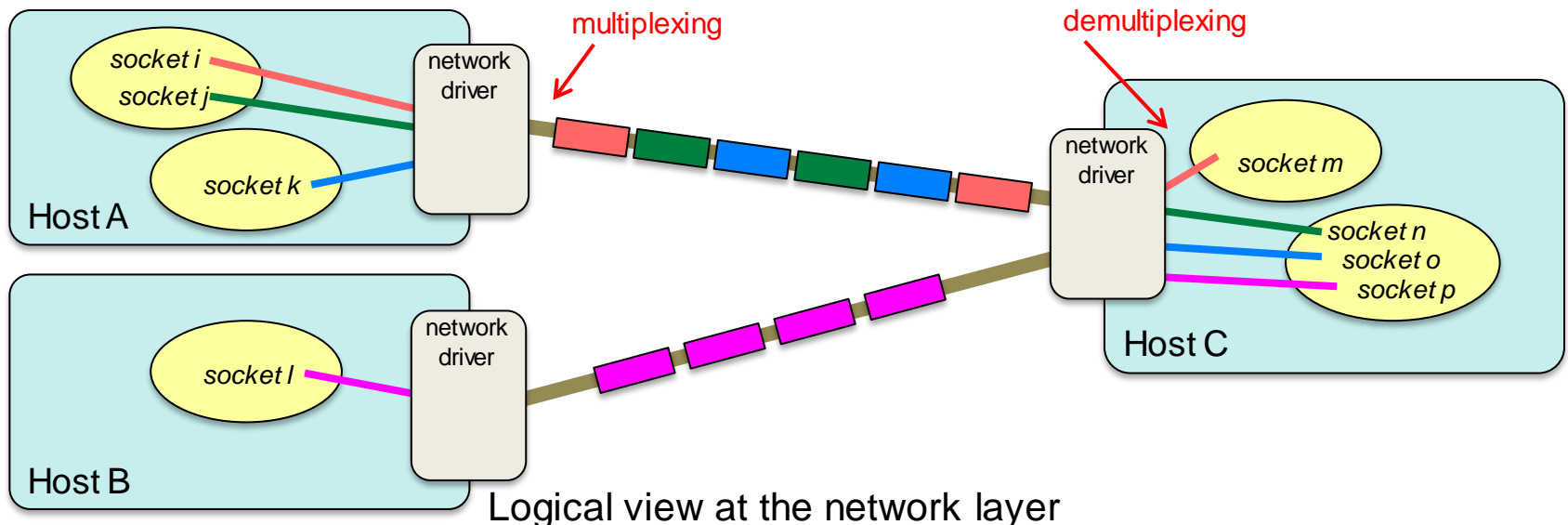
# Transport Layer Multiplexing

- Problem:  
Multiple communication channels over one network link
  - This is a problem whenever a protocol at one layer is used by multiple protocols or communication sessions at a higher layer



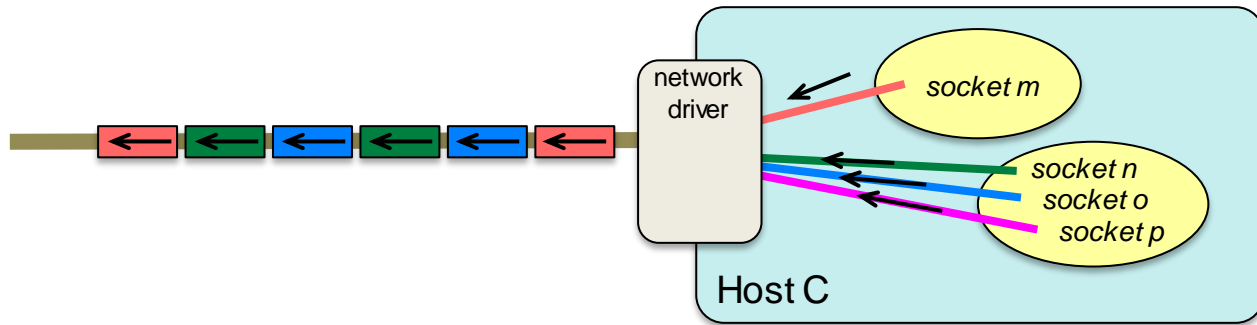
# Transport Layer Multiplexing

- Problem:  
Multiple communication channels over one network link
  - This is a problem whenever a protocol at one level is used by multiple protocols or communication sessions at one
- Need to identify which segment belongs to which channel

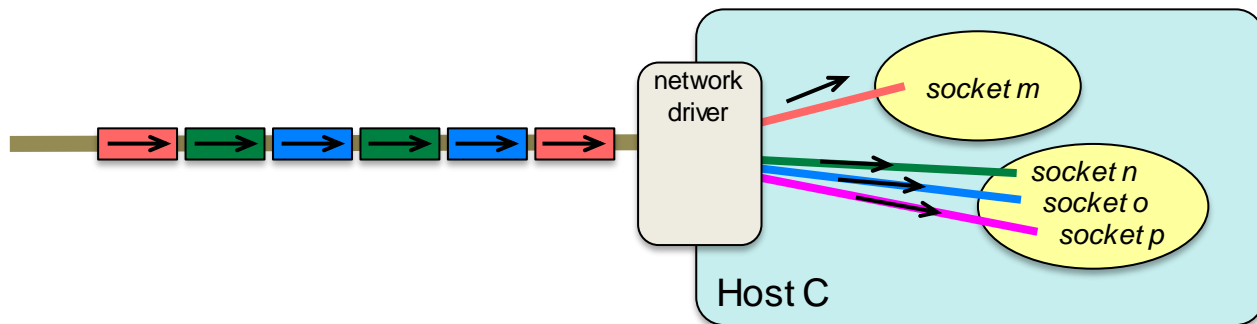


# Multiplexing & Demultiplexing

## Multiplexing



## Demultiplexing





# How is it done?

- Transport layer protocols in IP have **port numbers**
  - 16 bit integers (0 .. 65535)
  - IP header (network layer) has **source address, destination address**
  - TCP/UDP headers (transport layer) have **source port, destination port**
- Each socket is uniquely identified in the operating system
- Before a socket can be used, it is created & named
  - **socket** system call creates a unique socket
  - **bind** system call associates a local address with the socket
    - With an address of INADDR\_ANY, the socket is associated with ALL local interfaces
    - With a port of 0, the OS assigns a random unused port number to the socket

# UDP multiplexing & demultiplexing

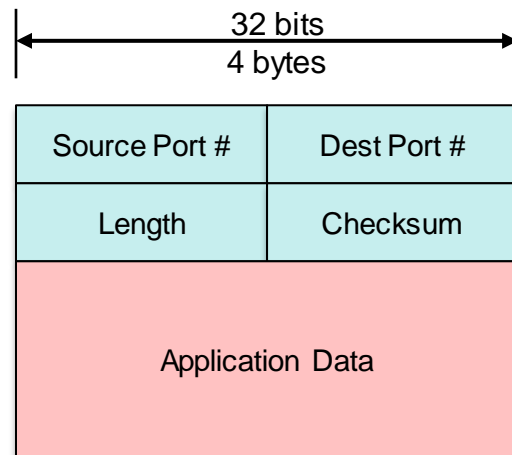
- A UDP socket is identified by its port number
- All UDP segments addressed to a specific port # will be delivered to the socket identified by that port number
  - A socket will request data via `recv()`, `recvfrom()`, or `recvmsg()` system calls
  - OS looks for a UDP socket with a matching destination port: hash table of socket structures; hash key created from UDP destination port
- Limited demultiplexing
  - Segments addressed to the same *(host, port)* from different processes or different systems will be delivered to the same socket!
  - The receiver can get the source address & port to know how to address reply messages

# Why use UDP?

- **Control the timing of data**
  - A UDP segment is passed to the network layer immediately for transmission
  - TCP uses congestion control to delay transmission
- **Preserve message boundaries**
  - With TCP, multiple small messages may be consolidated into one TCP segment
- **No connection setup**
  - TCP requires a three-way handshake to establish a connection
- **No state to keep track of**
  - Less memory, easier fault recovery, simple load balancing
- **Less network overhead**
  - 8-byte header instead of TCP's 20-byte header

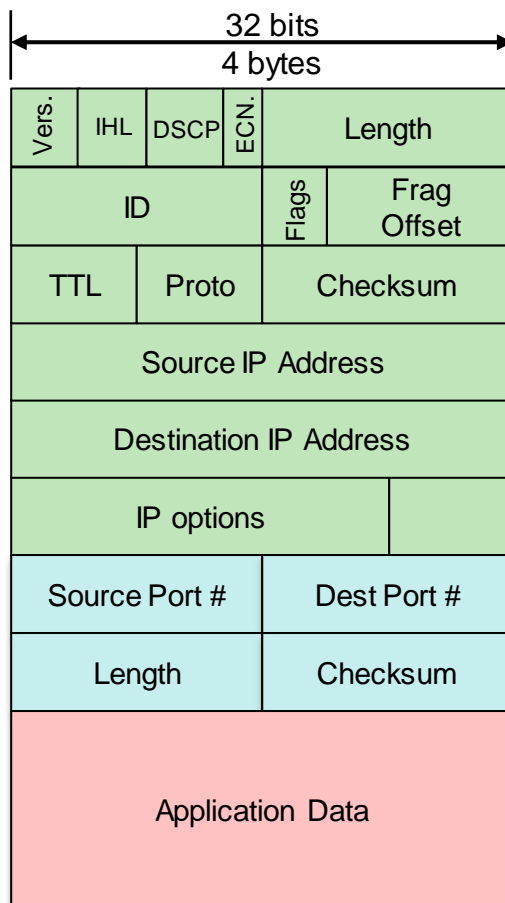
# UDP Structure

- Defined in RFC 768
- Eight byte header



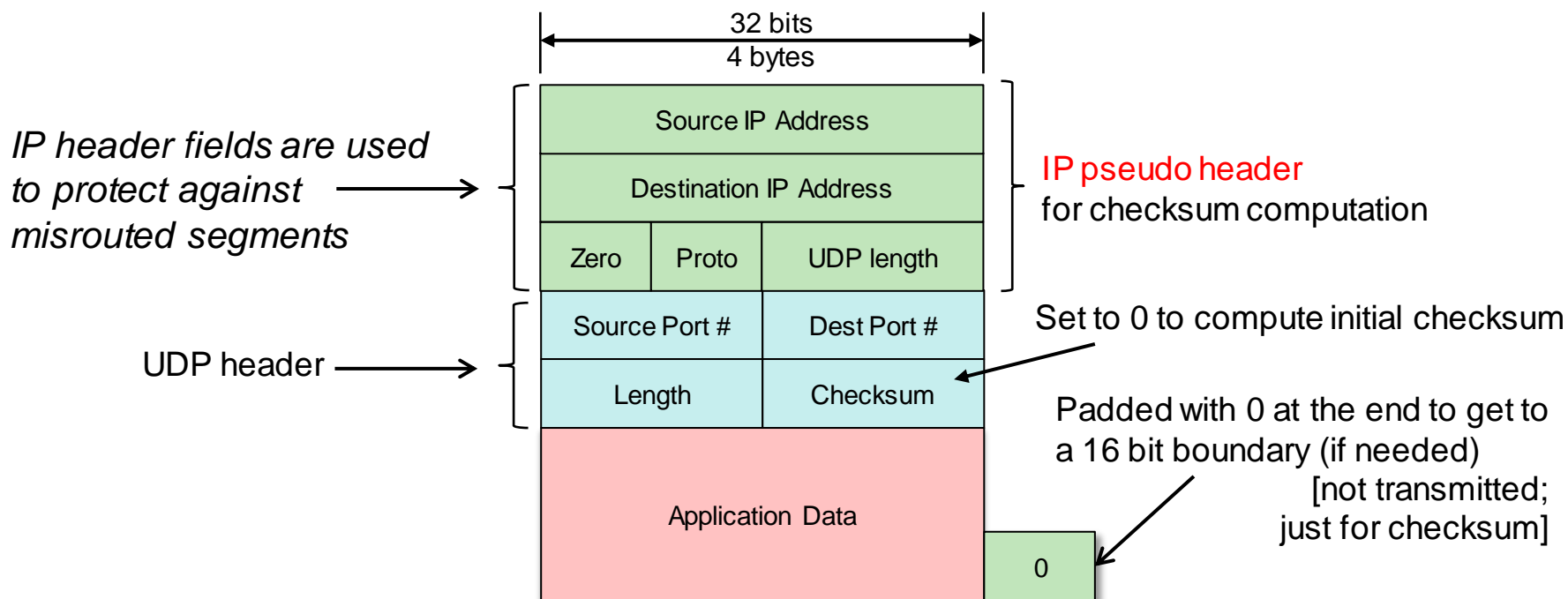
# UDP Structure in context

Eight byte header within a 20 byte IP header



# UDP Checksum

- IP does not guarantee error-free packet delivery
- The UDP header contains a 16-bit checksum
  - Checks for data corruption
- Checksum is generated by the sender and validated only by the receiver only: segments with bad checksums are simply dropped



# UDP Checksum Calculation

- **Sender**
  - Iterate over 16-bit words in the Pseudo header + UDP segment
  - UDP checksum field = 0
  - Create a **one's complement checksum**
    - Add two 16-bit values. If overflow, add 1 to the result
    - Do this for all the data you need to checksum
    - Invert the bits of the result to get the checksum value
- **Receiver**
  - Perform the same one's complement sum on all data *including* the checksum field
  - The result should be all 1s (**0xffff**)

**The same checksum calculation is used for the IP header, UDP header, & TCP header**

# One's Complement Checksum Example

- How to compute a One's complement

- Sum the numbers
- Add any overflow carry to the result

- Create checksum for:

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

```
    0 1 1 0  1 0 1 1  0 0 0 0  1 0 1 0
+   1 0 1 1  0 0 0 1  1 1 0 0  1 0 0 0
=  1 0 0 0 1  1 1 0 0  1 1 0 1  0 0 1 0
+
=  0 0 0 1  1 1 0 0  1 1 0 1  0 0 1 1
+  1 1 0 0  0 0 0 0  1 1 1 1  0 1 0 1
=  1 1 0 1  1 1 0 1  1 1 0 0  1 0 0 0
```

- Then invert the bits

```
~ 1 1 0 1  1 1 0 1  1 1 0 0  1 0 0 0
= 0 0 1 0  0 0 1 0  0 0 1 1  0 1 1 1 ← checksum
```



# One's Complement Checksum Example

- Validate
  - Sum the numbers, including the checksum

```
          0 1 1 0  1 0 1 1  0 0 0 0  1 0 1 0
          1 0 1 1  0 0 0 1  1 1 0 0  1 0 0 0
          1 1 0 0  0 0 0 0  1 1 1 1  0 1 0 1
add the checksum → 0 0 1 0  0 0 1 0  0 0 1 1  0 1 1 1
```

```
    0 1 1 0  1 0 1 1  0 0 0 0  1 0 1 0
+   1 0 1 1  0 0 0 1  1 1 0 0  1 0 0 0
=  1 0 0 0 1  1 1 0 0  1 1 0 1  0 0 1 0
+
=  0 0 0 1  1 1 0 0  1 1 0 1  0 0 1 1
+  1 1 0 0  0 0 0 0  1 1 1 1  0 1 0 1
=  1 1 0 1  1 1 0 1  1 1 0 0  1 0 0 0
+  0 0 1 0  0 0 1 0  0 0 1 1  0 1 1 1
=  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1
```

← add checksum

- A result of all 1's (= -0) means the transmission was good

# TCP multiplexing & demultiplexing

- Every TCP socket is identified by:

(source address, destination address, source port, destination port)

- A TCP socket has a state:

- **LISTEN**: the socket is used only for accepting connections

- **ESTABLISHED**: the socket is connected

- Other states that we'll ignore for now:

- Connection setup:

- SYN\_SENT: trying to establish a connection

- SYN\_RCVD: received a connection request

- Connection teardown:

- FIN\_WAIT\_1: socket has been closed by the local application; no acknowledgement from remote

- FIN\_WAIT\_2: socket has been closed by the local application; remote acknowledged the closing

- CLOSING: socket has been closed by the local & remote apps; remote has not acknowledged close

- TIME\_WAIT: connections closed; waiting to be sure that the remote side received the last ACK

- Let's look at an example

# Server: Create a new socket

```
svr = socket(AF_INET, SOCK_STREAM, 0);
```

Address family: Internet (IPv4)  
Type: "stream" – connection-oriented (TCP)

Create a new socket at the server: it has no addresses so far

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.11.5.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State

N.B.: We refer to a socket table here for convenience but it is just a logical construct. The actual implementation is operating-system specific but this data is generally stored in a list of socket buffer structures. On Linux, for example, the kernel function *tcp\_v4\_lookup* will search for either a listening or an established socket with specific addresses and ports (see *net/ipv4/tcp\_ipv4.c*, around line 507)

# Server: Bind – assign a local address

```
bind(svr) ;
```

Assign a local address (INADDR\_ANY) and port (1234) to the socket

**Client (135.10.10.1)**

Local Addr	Local Port	Remote Addr	Rem Port	State

**Server (192.11.5.8)**

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			

# Server: Make it a listening socket

```
listen(svr, 10);
```

Set the state of the socket to *listen*. This socket can *only* be used to accept connections

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.11.5.8)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

svr →

# Server: Wait for a connection

```
snew = accept(svr);
```

Wait for an incoming connection on this socket

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.11.5.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

# Client: Create a new socket

```
s = socket();
```

Create an new socket at the client: no addresses so far

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

← S

Server (192.11.5.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

# Client: Assign a local address & port #

`bind(s) ;`

Assign any local address (INADDR\_ANY) and have the OS pick a port (port=0)

**Client (135.10.10.1)**

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801			

← S

**Server (192.11.5.8)**

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN



# Client: Connect to the server

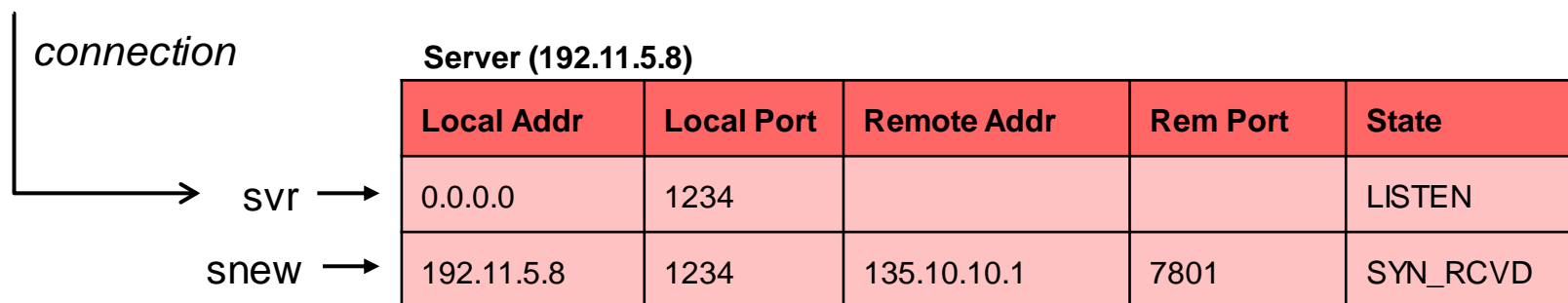
```
connect(s, dest_addr);
```

Connect to address 192.11.5.8, port 1234

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801			

← S



- Send a connection establishment request to address 192.11.5.8, port 1234 (TCP segment to port 1234 with a connection setup bit set; we'll look at the exact handshake later)
- On the server, search the table for a LISTEN socket where
  - packet's destination addr == table's local addr (0.0.0.0 matches any incoming addr)
  - packet's destination port == table's local port
- Create a new socket for the connection

# Client: Complete the connection

```
connect(s, dest_addr);
```

Server acknowledges the connection; Client fills in the entry

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED ← S

Server (192.11.5.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew →	192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED

Now we can talk!

# Communicate

## Client-to-server communication

- Server finds socket by searching for a TCP socket with these properties:
  1. Status == ESTABLISHED
  2. IP src addr == remote addr
  3. TCP src port = remote port
  4. IP dest addr == local addr
  5. TCP dest port == local port

### Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED ← S

### Server (192.11.5.8)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN
192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED

svr →

snew →

# Communicate

## Server-to-client communication

- Client finds socket by searching for a TCP socket with these properties:
  1. Status == ESTABLISHED
  2. IP src addr == remote addr
  3. TCP src port = remote port
  4. IP dest addr == local addr
  5. TCP dest port == local port

### Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED

← S

### Server (192.11.5.8)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN
192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED

svr →

snew →

# Two clients sharing the same port

Different source address disambiguates the sessions

Client (135.10.10.2)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED

Server (192.11.5.8)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN
192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED
192.11.5.8	1234	135.10.10.2	7801	ESTABLISHED

svr →  
snew1 →  
snew2 →

# Two endpoints sharing the same address

The OS will not allow two sockets to share the same port on one client

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	<b>7801</b>	192.11.5.8	1234	ESTABLISHED
0.0.0.0	<b>7802</b>	192.11.5.8	1234	ESTABLISHED

Server (192.11.5.8)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN
192.11.5.8	1234	135.10.10.1	<b>7801</b>	ESTABLISHED
192.11.5.8	1234	135.10.10.1	<b>7802</b>	ESTABLISHED

svr →

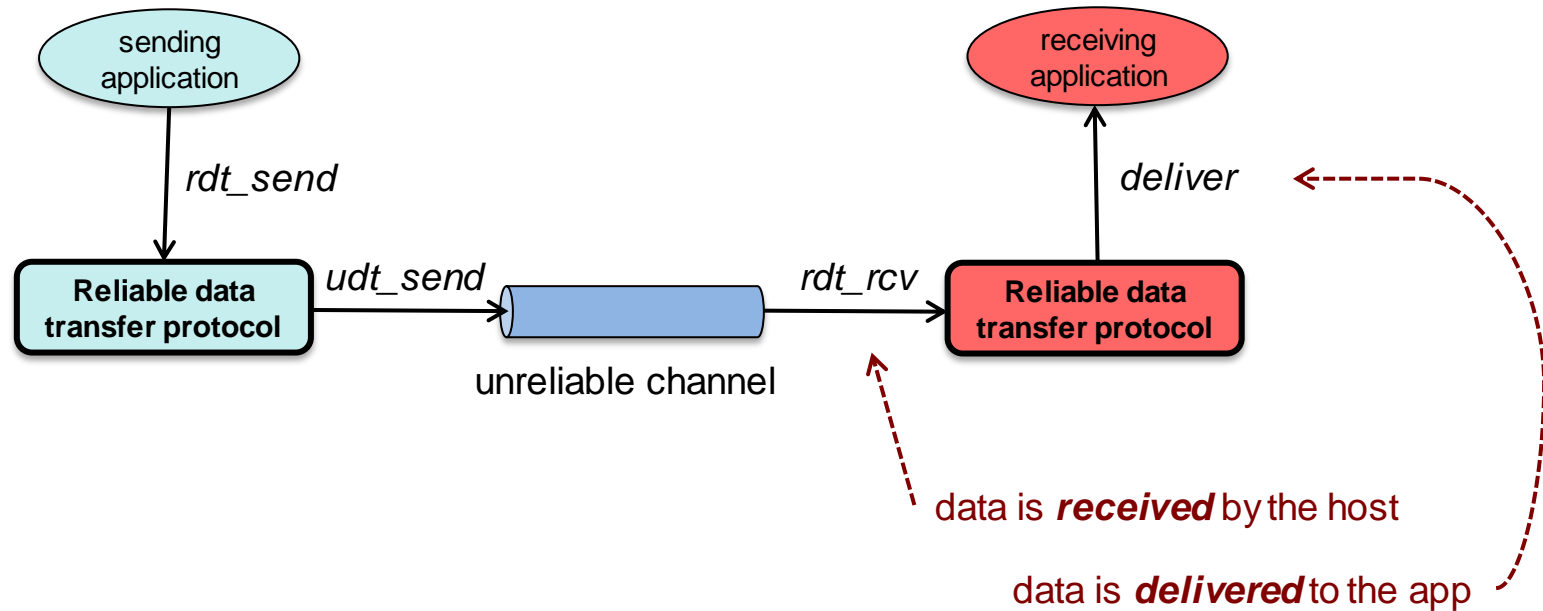
snew1 →

snew2 →

# Reliable Data Transfer

# Reliable Data Transfer (RDT) Goal

Develop a protocol for transmitting data reliably over an unreliable network

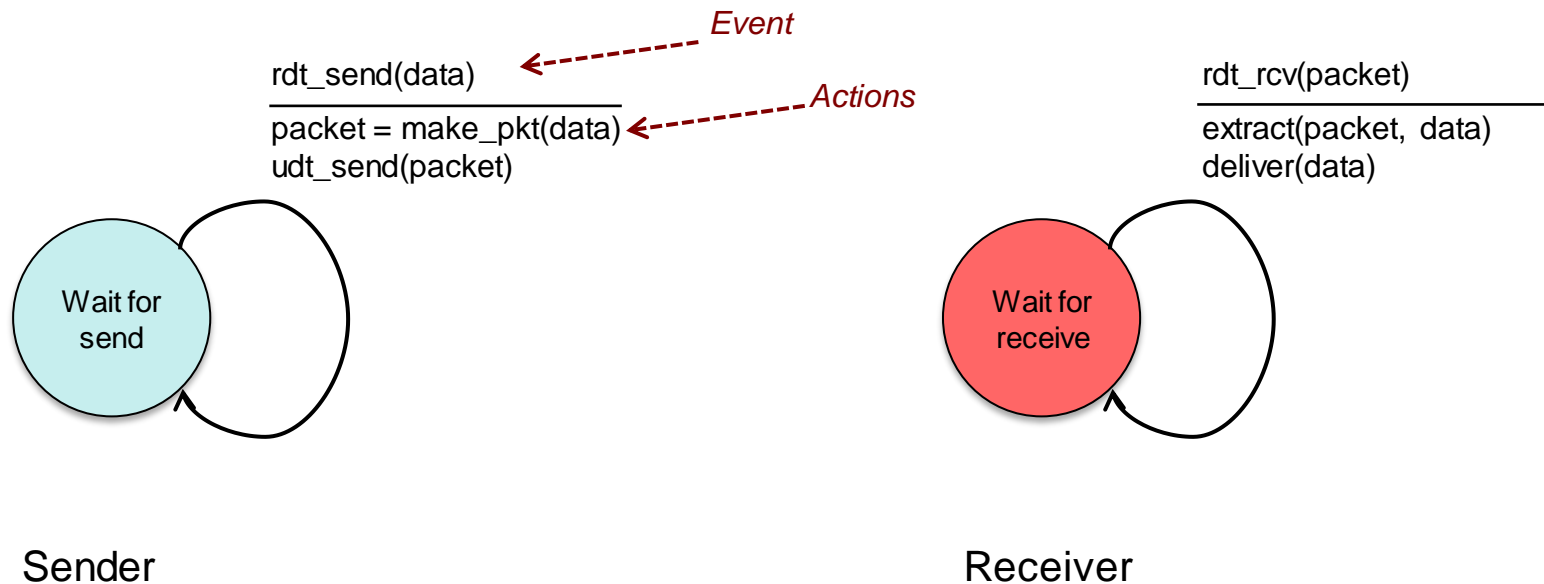




# RDT over a reliable channel

- Assume the channel is reliable
- Trivial – nothing to do!

Here's the finite state machine (FSM):



# RDT over a channel with bit errors

- All packets are received
- Some might be corrupt
- Approach
  - Acknowledge each packet
    - **Positive acknowledgement** (ACK): *“I got it; looks good!”*
    - **Negative acknowledgement** (NAK): *“Please repeat”*
  - Sender retransmits a packet if it receives a NAK
  - **ARQ (Automatic Repeat reQuest)**
    - Set of protocols that use acknowledgements & retransmission

# We need to support three capabilities

---

## 1. Error detection

- How do we know if the packet is corrupt?
- Use a checksum (**error detecting code**)

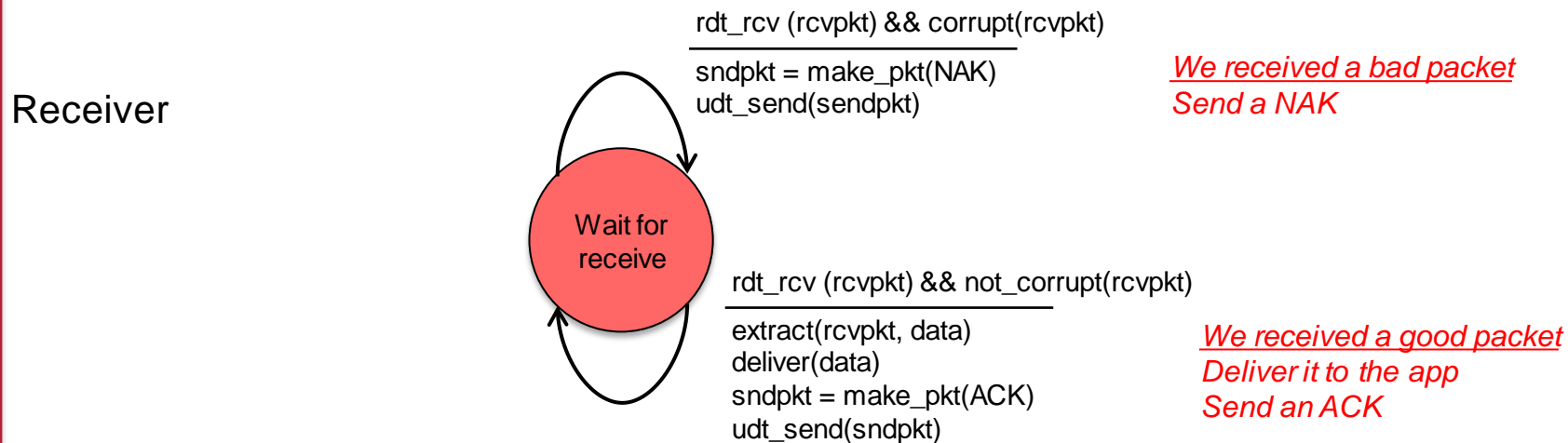
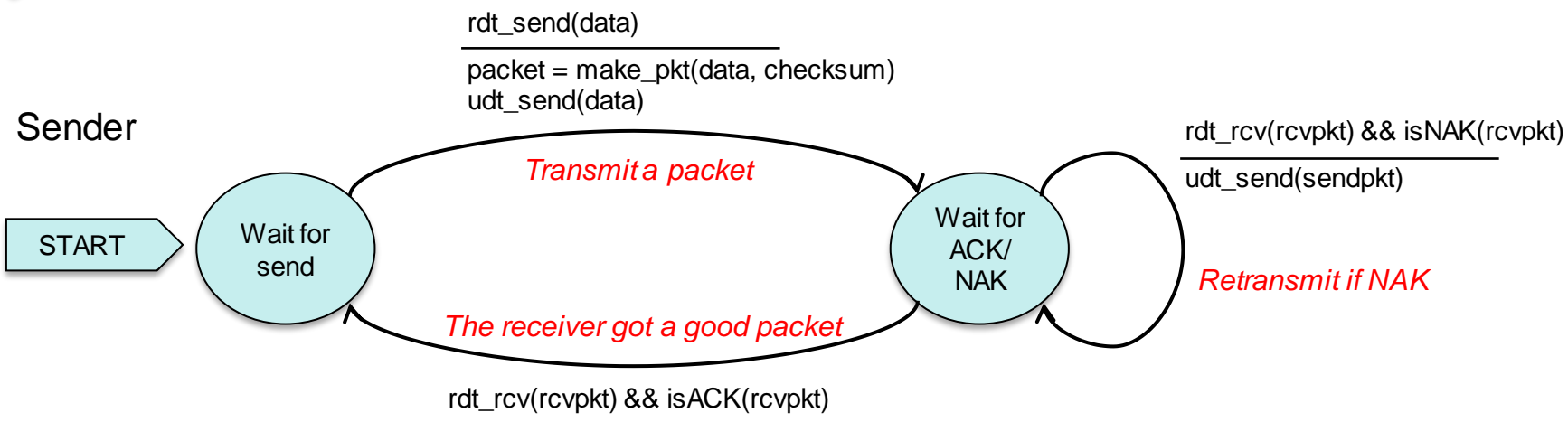
## 2. Receiver feedback

- The receiver will acknowledge each packet with an ACK or NAK

## 3. Retransmission

- If a sender gets a NAK, the packet will be retransmitted

# RDT over a channel with bit errors



# Stop-and-wait

---

- The sender cannot send any data until it receives an ACK for the previously sent packet
- This type of protocol is a **stop-and-wait** protocol

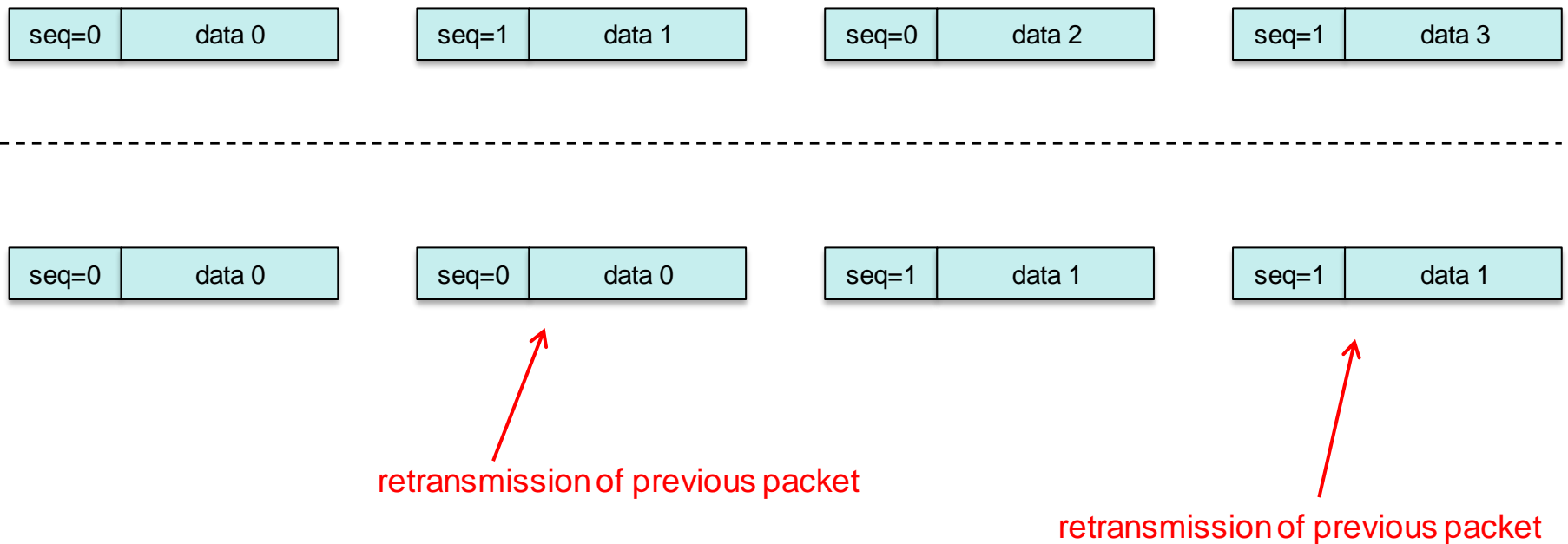
# What about a corrupted ACK/NAK message?

- The sender does not know whether the last packet was received correctly or not
- We can
  - Have the sender send a “please repeat” in response to a corrupt ACK/NAK
    - But what if that gets corrupted?
  - Add a robust error correcting code
    - Works for a channel that does not lose data
  - Resend the data in response to a corrupted ACK/NAK
    - Duplicate packets may be received
    - Receiver needs to distinguish between new data & a retransmission
    - Use a **sequence number**. Here, we only need a 1-bit number.

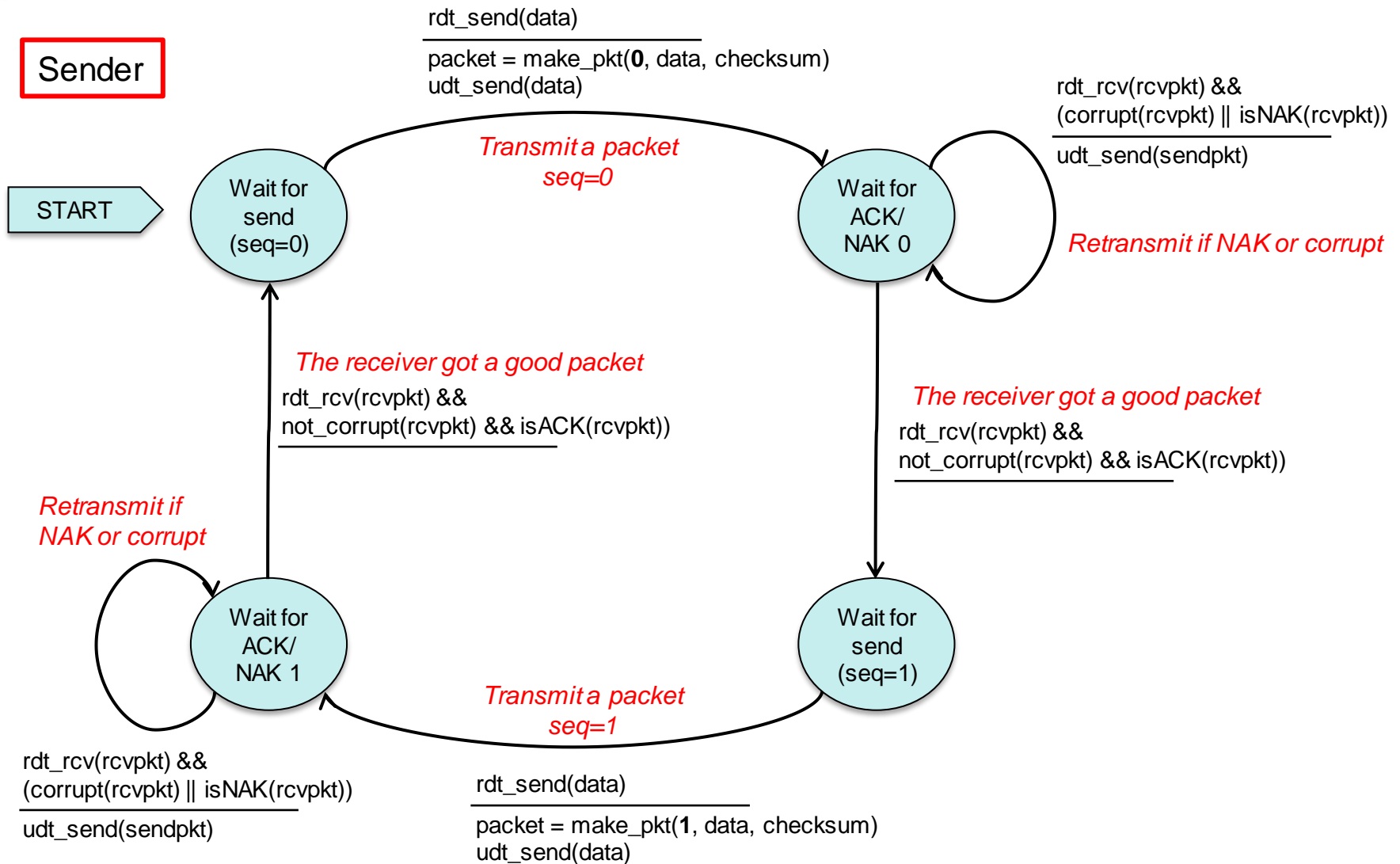
# A 1-bit sequence number

Sequence bit flip-flops between consecutive messages

Alternating bit protocol



# RDT over a channel with bit errors





# RDT over a channel with bit errors

## Receiver

### Corrupt data – send NAK

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
-----
sndpkt = makepkt(NAK, checksum)
udt_send(sndpkt)
```

START

### Duplicate data – send ACK

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
&& has_seq1(rcvpkt)
-----
sndpkt = makepkt(ACK, checksum)
udt_send(sndpkt)
```

### Received seq=0

### Deliver to app; Send ACK

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
&& has_seq0(rcvpkt)
-----
extract(rcvpkt, data)
deliver(data)
sndpkt = make(ACK, checksum)
udt_send(sndpkt)
```

### Corrupt data – send NAK

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
-----
sndpkt = makepkt(NAK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
&& has_seq0(rcvpkt)
-----
```

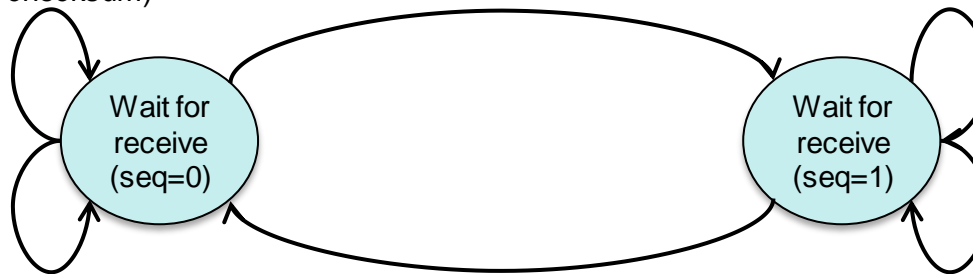
```
sndpkt = makepkt(ACK, checksum)
udt_send(sndpkt)
```

### Duplicate data – send ACK

### Received seq=1

### Deliver to app; Send ACK

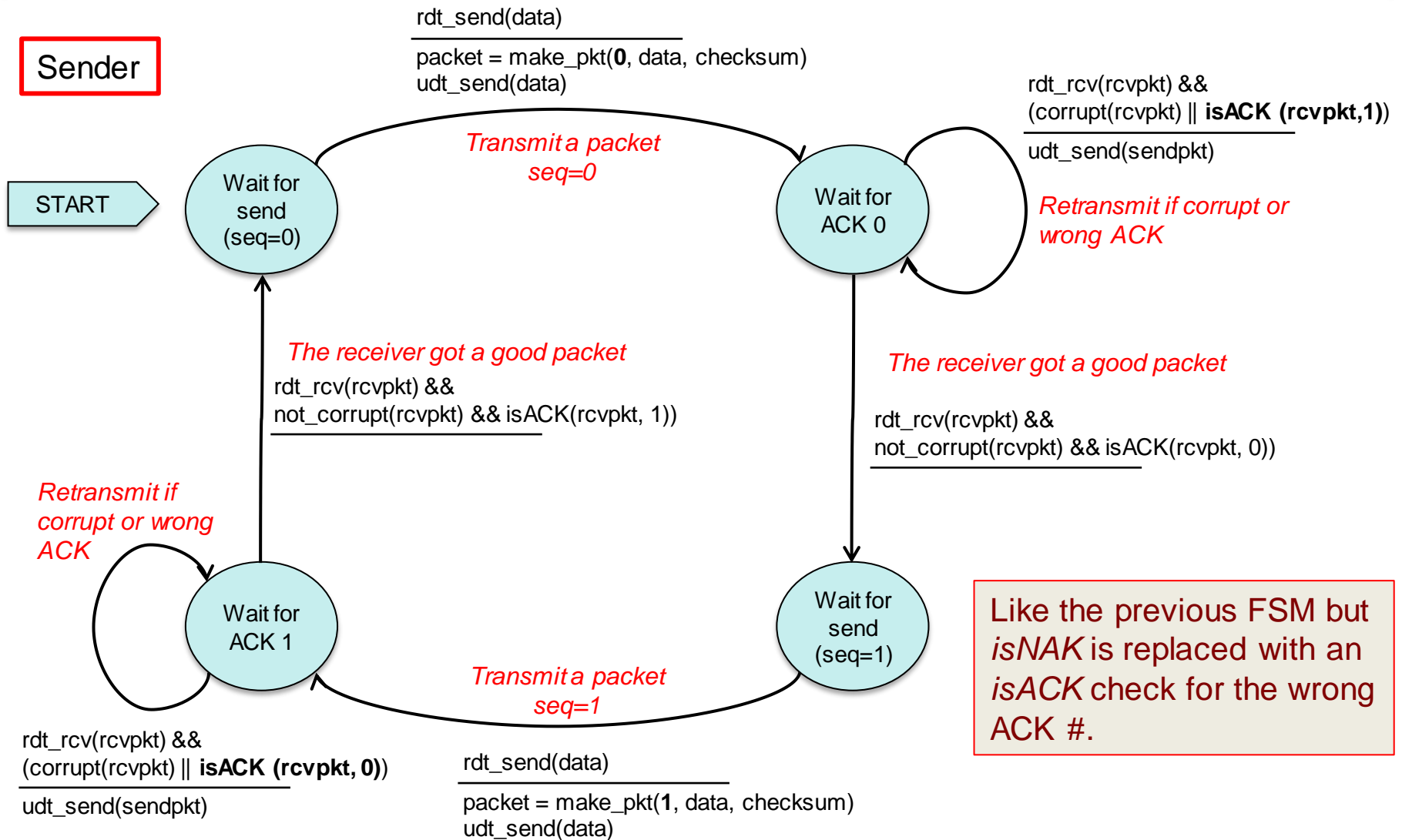
```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
&& has_seq1(rcvpkt)
-----
extract(rcvpkt, data)
deliver(data)
sndpkt = make(ACK, checksum)
udt_send(sndpkt)
```



# RDT over a channel with bit errors

- If a corrupted packet is received
  - Send a NAK
- If a duplicate packet is received
  - Send an ACK since we already processed the packet
- We can get rid of NAKs
  - Send an ACK for the last correctly received packet
  - If a sender receives duplicate ACKs, it knows that the previous packet has not been received correctly
  - Modify protocol: add **sequence numbers** to ACKs

# RDT over a channel with bit errors – no NAK



# RDT over a channel with bit errors – no NAK

Receiver

*Received seq=0*

*Deliver to app; Send ACK #0*

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
&& has_seq0(rcvpkt)
```

```
extract(rcvpkt, data)
```

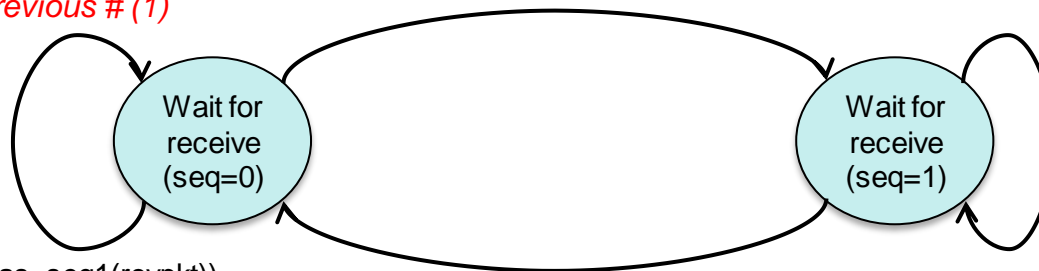
```
deliver(data)
```

```
sndpkt = make(ACK, 0, checksum)
```

```
udt_send(sndpkt)
```

*Corrupt or duplicate packet.  
Send ACK with previous # (1)*

*Corrupt or duplicate packet.  
Send ACK with previous # (0)*



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

```
sndpkt = makepkt(ACK, 1, checksum)
```

```
udt_send(sndpkt)
```

*Received seq=1*

*Deliver to app; Send ACK #1*

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
&& has_seq1(rcvpkt)
```

```
extract(rcvpkt, data)
```

```
deliver(data)
```

```
sndpkt = make(ACK, 1, checksum)
```

```
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) || has_seq0(rcvpkt))
sndpkt = makepkt(ACK, 0, checksum)
udt_send(sndpkt)
```

# RDT over a lossy channel

---

- We considered only bit errors
  - Packets were always delivered
- How do we detect & deal with packet loss?

# Dealing with packet loss

- Burden of detection & recovery is on sender
- If sender's packet is lost OR receiver's ACK is lost
  - Sender will not get a reply from the receiver
- Approach
  - Introduce a **countdown timer**. Set the timer at transmit
  - If time-out and no reply, retransmit
  - How long to wait? Maximum round-trip delay?
    - Long wait until we initiate error recovery
    - Pick a “likely loss” time
    - Retransmit if no response within that time
    - Introduces possibility of **duplicate packets**
      - But we already know how to deal with them

# RDT over lossy channel – with a timer

**Sender**

rdt\_send(data)  
 packet = make\_pkt(0, data, checksum)  
 udt\_send(data)  
**start\_timer**

*Retransmit if corrupt, wrong ACK, or timeout*

rdt\_rcv(rcvpkt) &&  
 (corrupt(rcvpkt) || isACK(rcvpkt,1))  
 (do nothing; wait for a timeout)

timeout  
 udt\_send(sendpkt)  
**start\_timer**

*The receiver got a good packet*  
rdt\_rcv(rcvpkt) &&  
 not\_corrupt(rcvpkt) && isACK(rcvpkt, 0)  
**stop\_timer**

*The receiver got a good packet*  
rdt\_rcv(rcvpkt) &&  
 not\_corrupt(rcvpkt) && isACK(rcvpkt, 1)  
**stop\_timer**

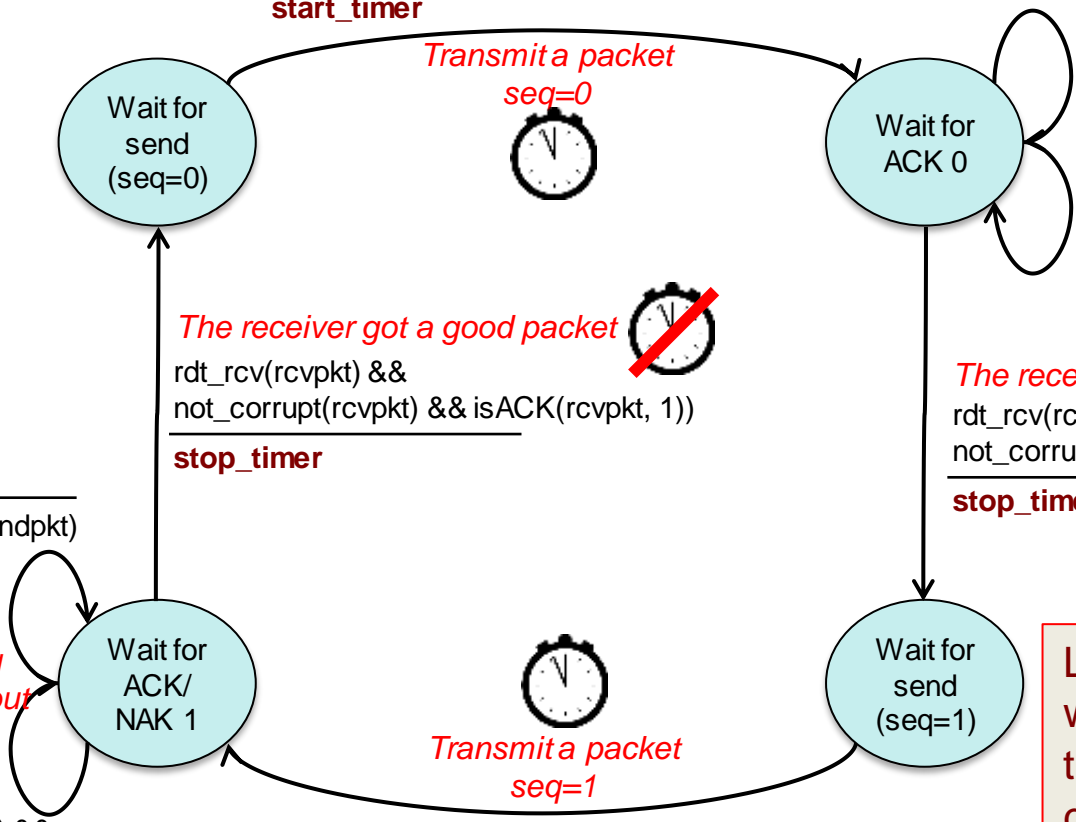
timeout  
 udt\_send(sendpkt)  
**start\_timer**

*Retransmit if  
 corrupt, wrong  
 ACK, or timeout*

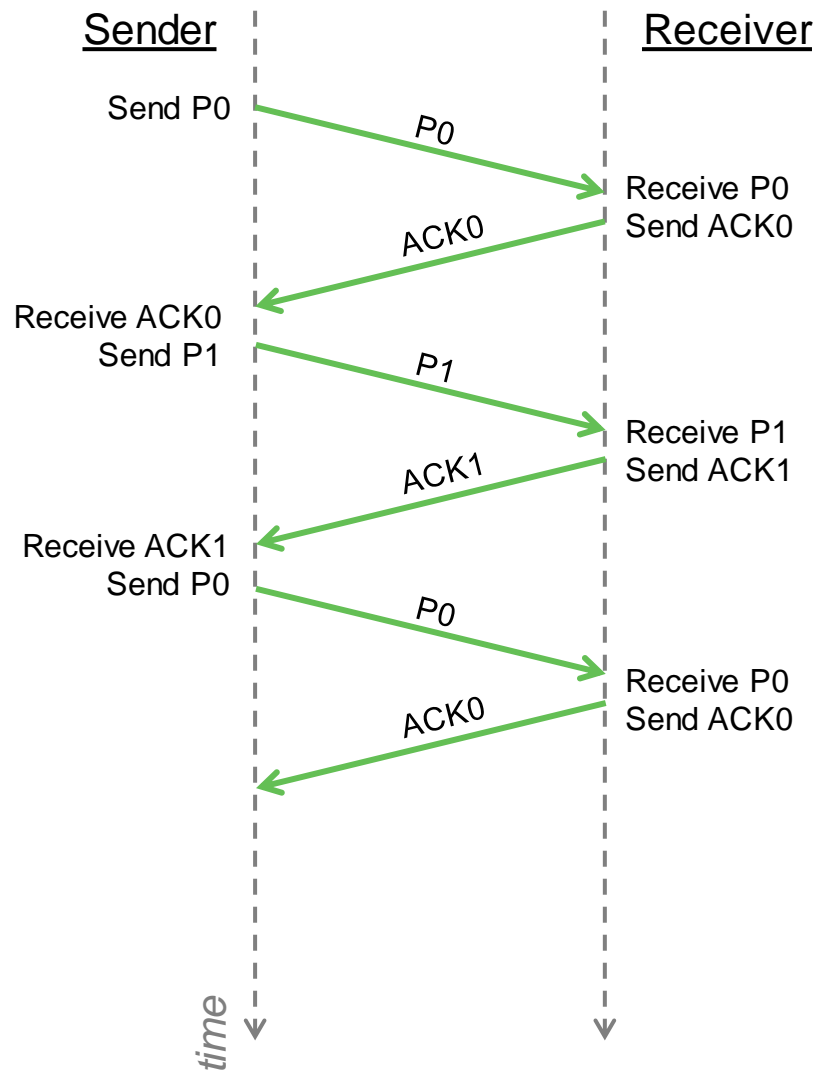
rdt\_rcv(rcvpkt) &&  
 (corrupt(rcvpkt) || isACK(rcvpkt, 0))  
 (do nothing; wait for a timeout)

rdt\_send(data)  
 packet = make\_pkt(1, data, checksum)  
 udt\_send(data)  
**start\_timer**

Like the previous FSM but with a timer set on transmit and a timeout check when waiting for an ACK

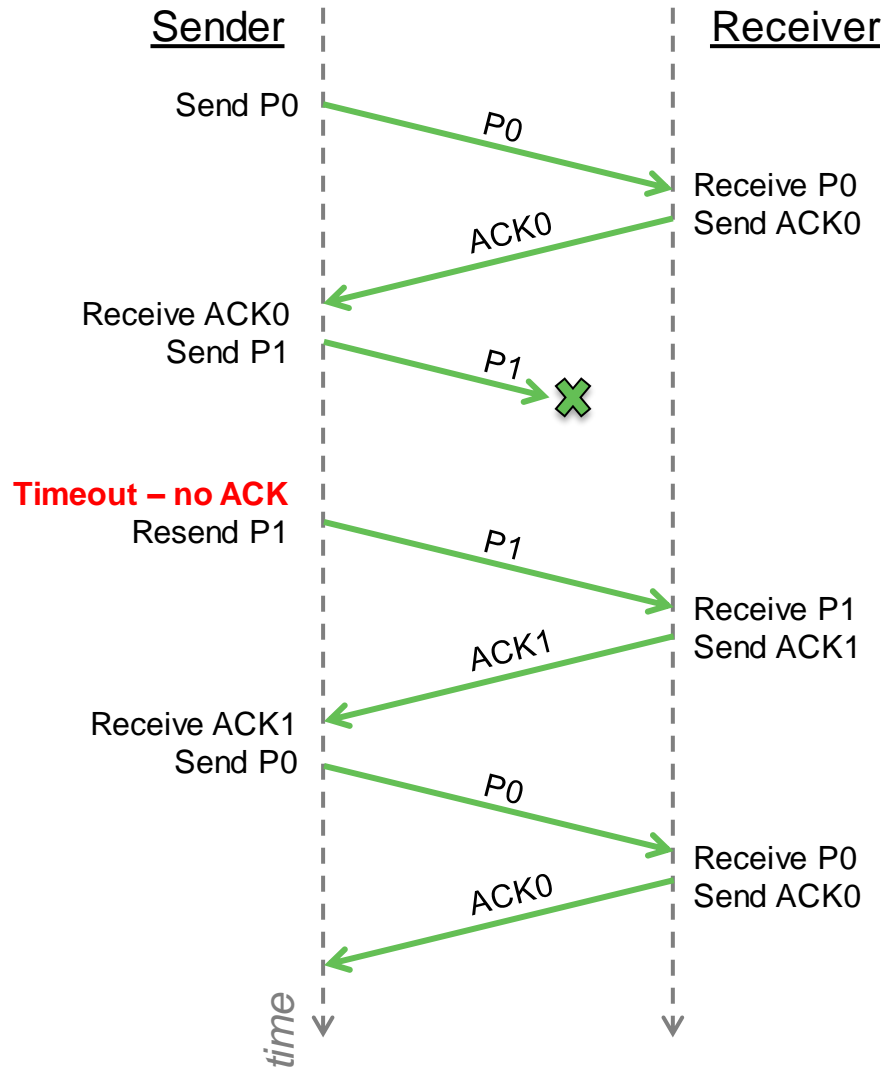


# RDT – Alternating Bit Protocol: no loss

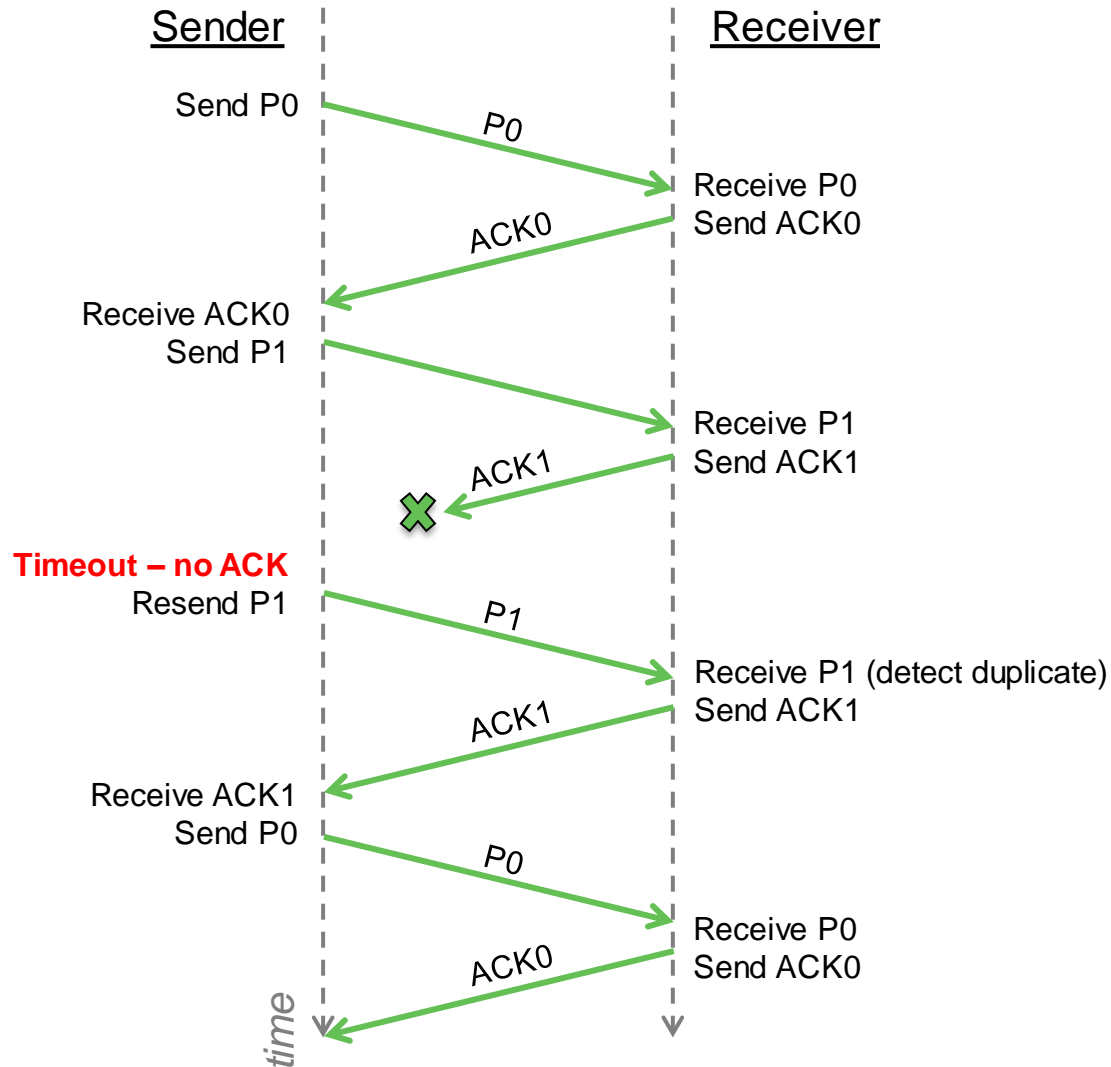




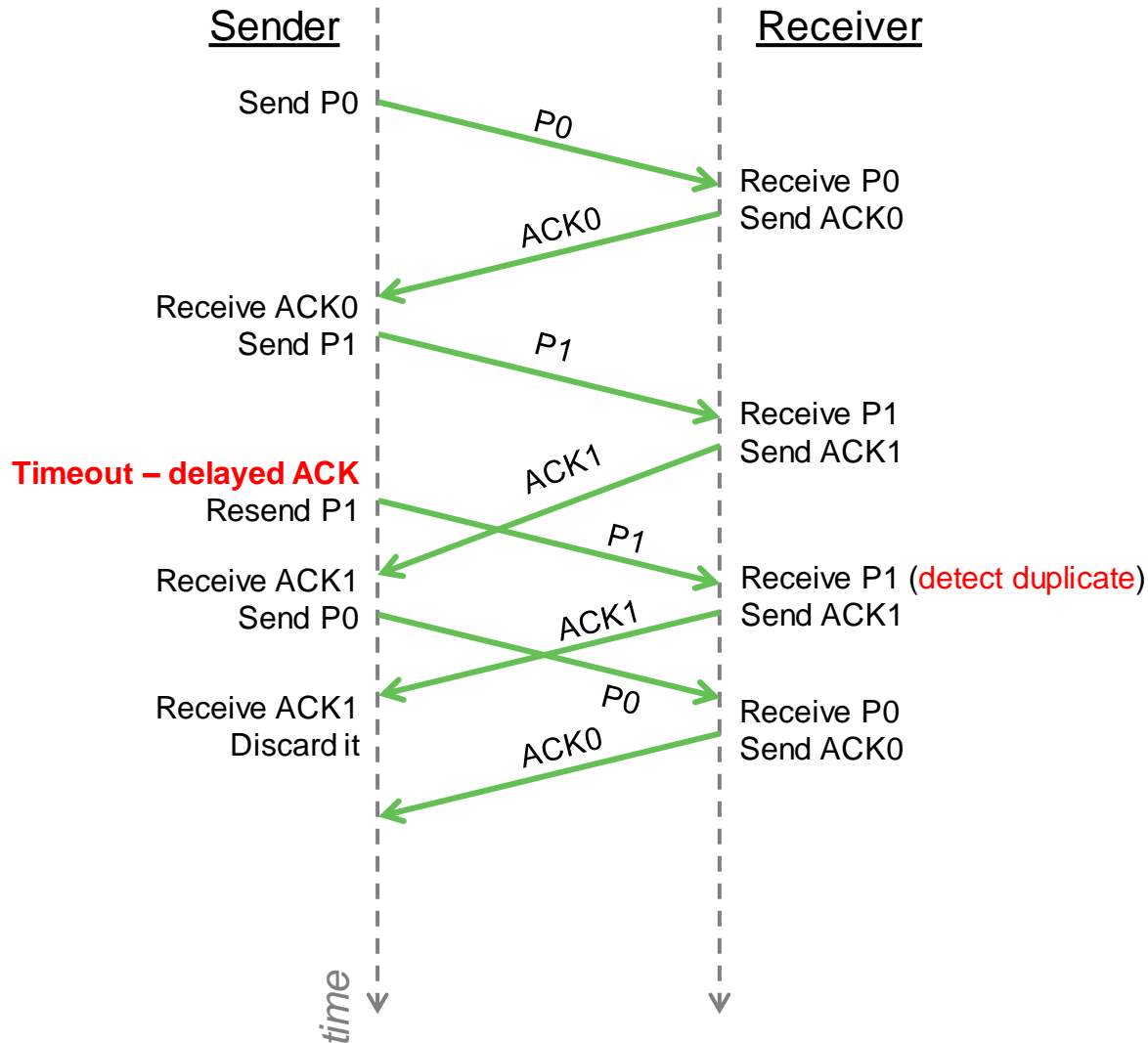
# RDT – Alternating Bit Protocol: lost Packet



# RDT – Alternating Bit Protocol: lost ACK



# RDT – Alternating Bit Protocol: early timeout



# Network utilization with stop-and-wait

- A **stop-and-wait** protocol gives us horrible **network utilization**
- Consider
  - Cross-country link  $\Rightarrow$  Round-trip propagation delay (RTT)  $\approx 30$  ms
  - Assume 1 Gbps link (ignore router delays),  $R = 10^9$  bits/second
  - Assume 1,000-byte packets ( $L = 8,000$  bits)
  - Time to transmit the packet:  $d_{trans} = L \div R = 8,000 \div 10^9 = 8 \mu\text{s}$
- With a stop-and-wait protocol
  - one-way delay =  $d_{trans} + d_{prop} = (30 \text{ ms} \div 2) + 8 \mu\text{s} = 15.008 \text{ ms}$
  - Assume ACK packets are tiny; one-way delay for ACK packet = 15 ms
    - ACK is received at  $15.008 + 15 = 30.008 \text{ ms}$
  - Next packet can be sent  $(15.008 + 15) = 30.008 \text{ ms}$  after the first one
  - **Utilization** = fraction of time sender is sending bits into the channel

$$U = \frac{L / R}{RTT + (L / R)} = \frac{0.008}{30.008} = 0.00027 = 0.027\%$$

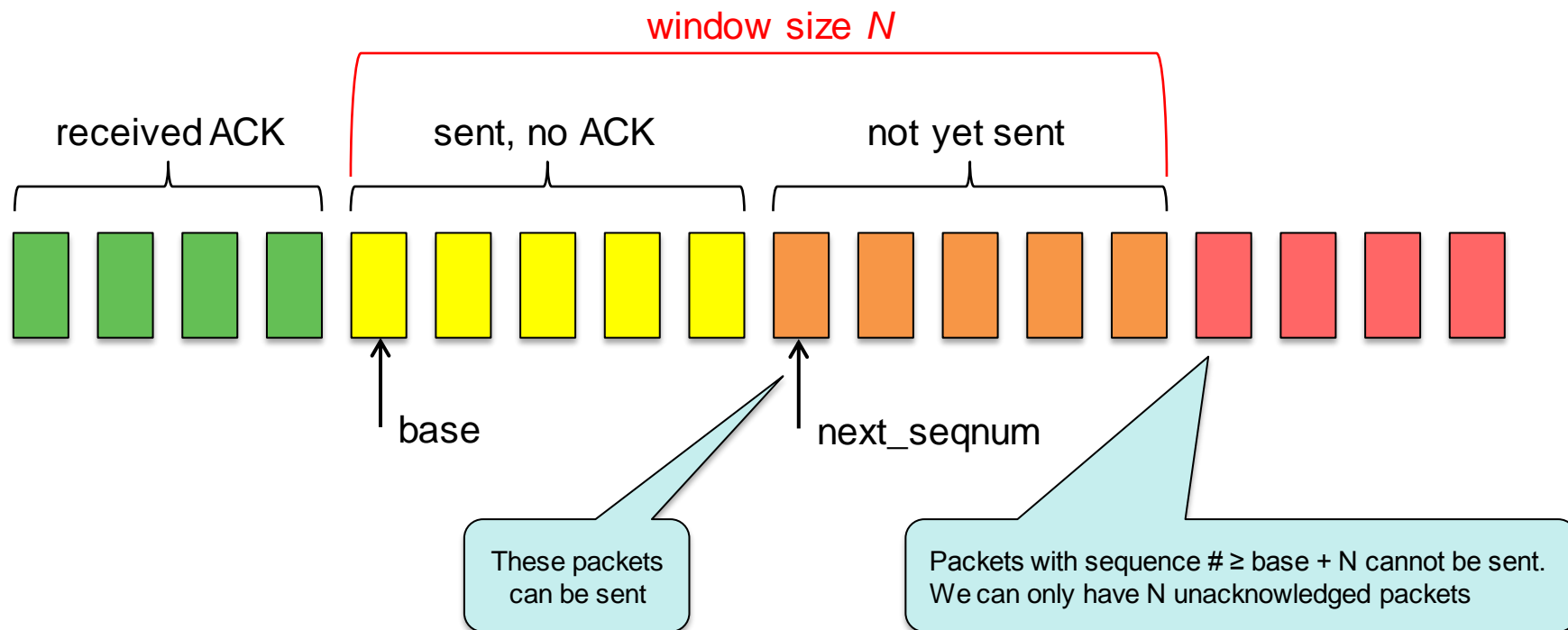
The sender can transmit 1,000 bytes in 30.008 ms: 267 kbps on a 1 Gbps link!

# Improve Network Utilization: Pipelining

- Don't wait for an acknowledgement before sending the next packet
- But then we need to
  1. Increase the range of sequence numbers
    - Each in-transit packet needs a unique number
  2. Hold on to unacknowledged packets at sender
  3. Hold on to out-of-sequence packets at receiver
- Two approaches for pipelined error recovery
  - **Go-Back-N**
  - **Selective Repeat**

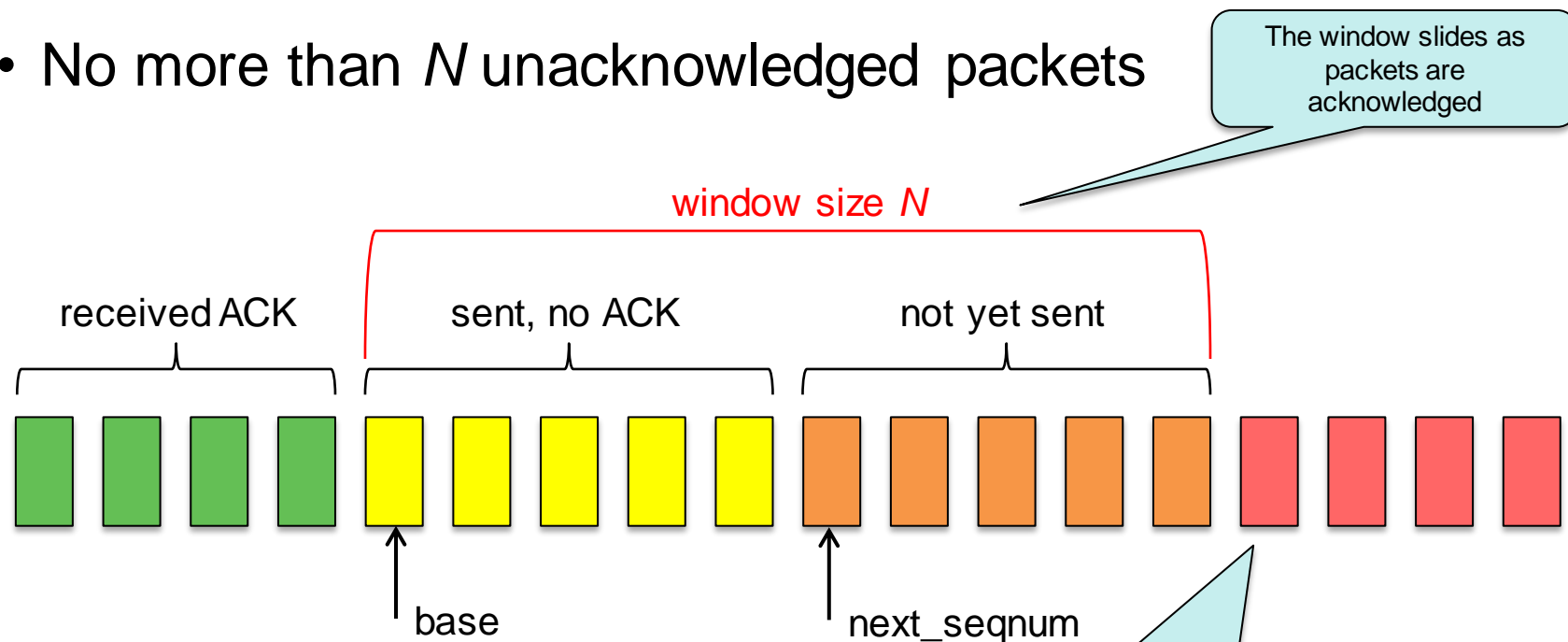
# Go-Back-N (GBN)

- Sender can send multiple packets without waiting for ACKs
- No more than  $N$  unacknowledged packets



# Go-Back-N (GBN)

- Sender can send multiple packets without waiting for ACKs
- No more than  $N$  unacknowledged packets

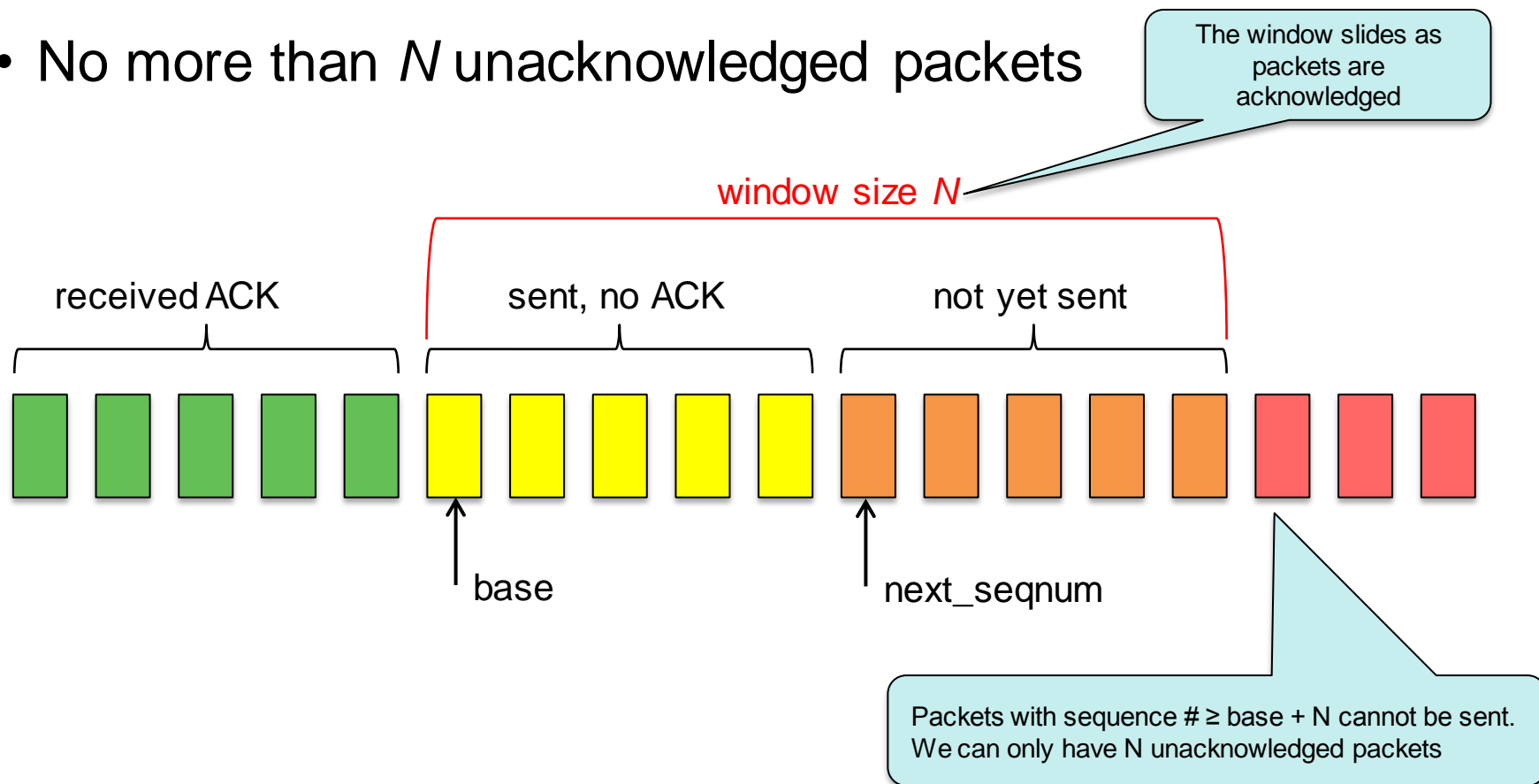


Packets with sequence #  $\geq$  base +  $N$  cannot be sent. We can only have  $N$  unacknowledged packets

**GBN = Sliding Window Protocol**

# Go-Back-N (GBN)

- Sender can send multiple packets without waiting for ACKs
- No more than  $N$  unacknowledged packets

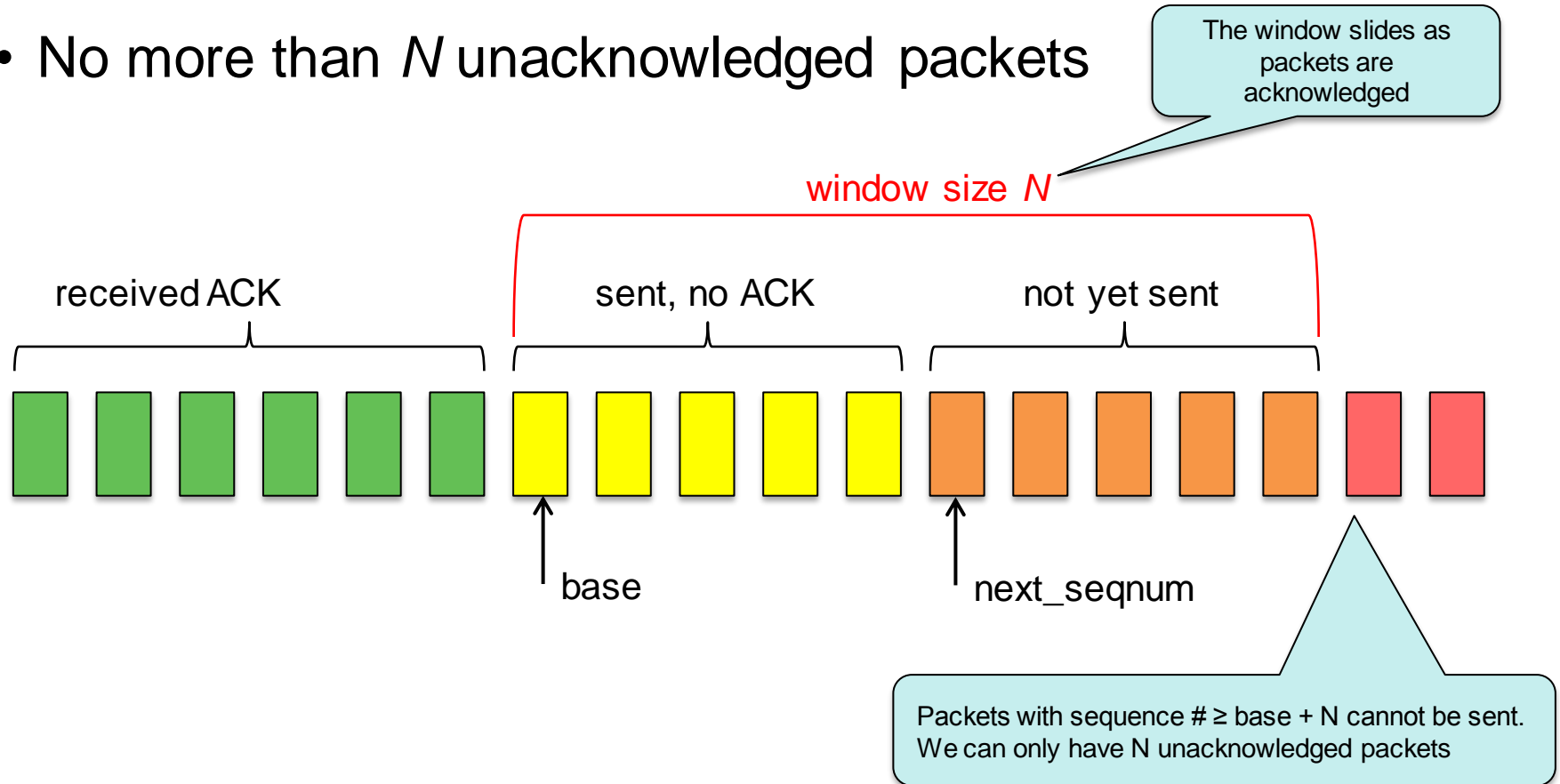


**GBN = Sliding Window Protocol**



# Go-Back-N (GBN)

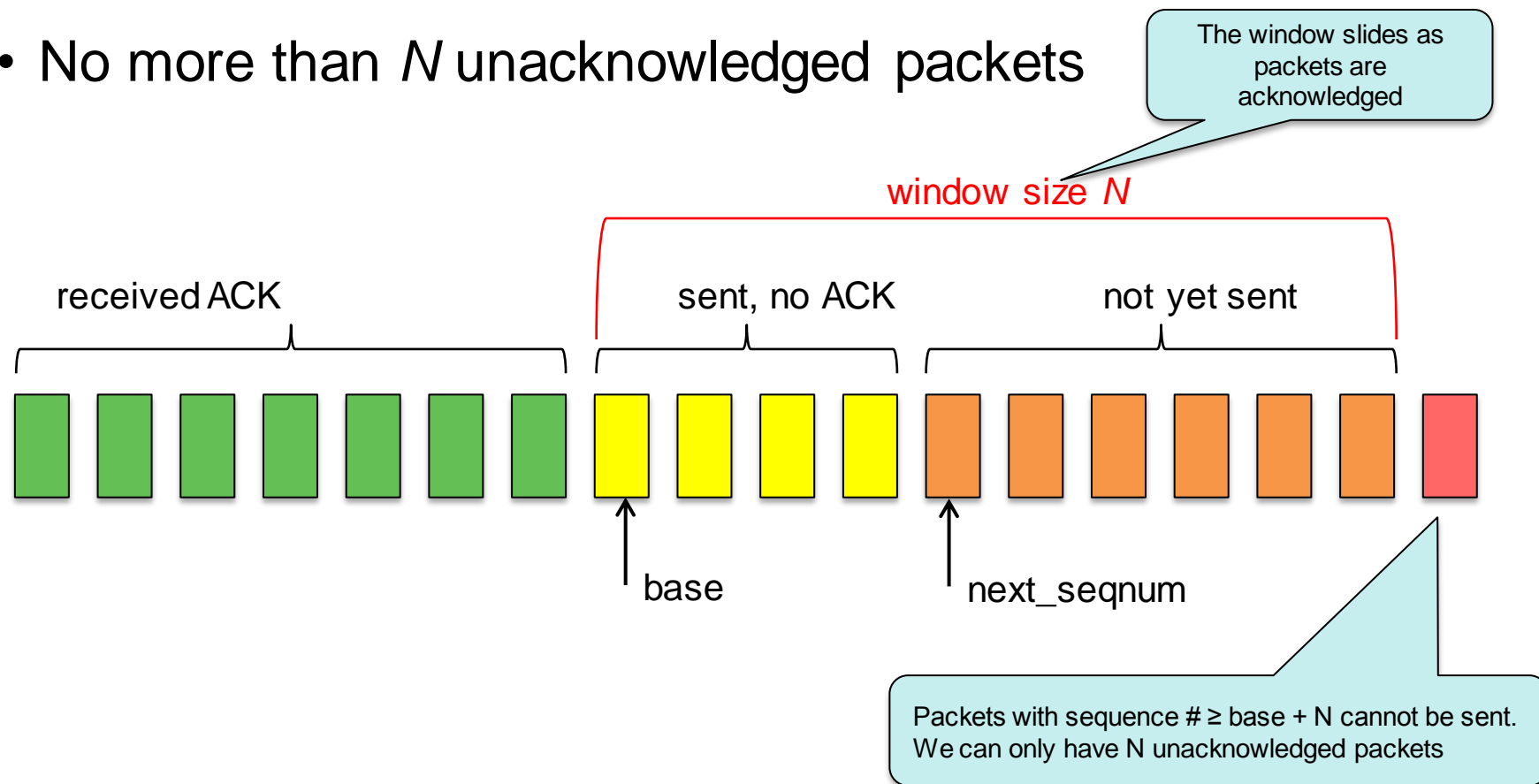
- Sender can send multiple packets without waiting for ACKs
- No more than  $N$  unacknowledged packets



**GBN = Sliding Window Protocol**

# Go-Back-N (GBN)

- Sender can send multiple packets without waiting for ACKs
- No more than  $N$  unacknowledged packets



**GBN = Sliding Window Protocol**

# Sequence numbers

A sequence number will take up a fixed #,  $k$ , of bits in the header

- Range of sequence numbers is  $0 \dots 2^k - 1$
- Modulo  $2^k$  arithmetic:  $2^k - 1$  increments to 0

# Extended FSM for a GBN sender

Sender

*Send data if it's in the window (we can have at most N unacknowledged packets)*

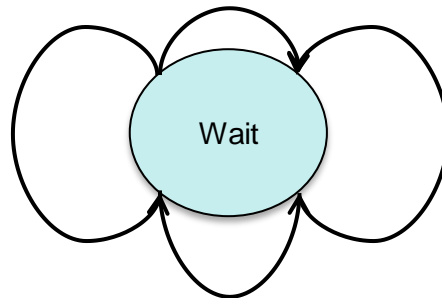
```
rdt_send(data)


---


if (next_seqnum < base+N) { // there's room in the window
    sndpkt[next_seqnum] = make_pkt(next_seqnum, data, checksum)
    udt_send(sndpkt[next_seqnum])
    if (base == next_seqnum)
        start_timer
    next_seqnum++
} else {
    refuse_data(data) // cannot send
```

*Ignore corrupted ACKs*

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
```



**Go Back N:**

Timeout means resend all unacknowledged packets

timeout

```
start_timer
for (i = base; i < next_seqnum; i++)
    udt_send(sndpkt[i])
```

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)
```

```
base = get_acknum(rcvpkt)+1
if (base == next_seqnum)
    stop_timer // we have the latest ACK
else
    start_timer // still waiting for ACKs
```

**Cumulative acknowledgement:**

Receipt of a sequence number  $n$  ACK means that all packets up to and including  $n$  have been received

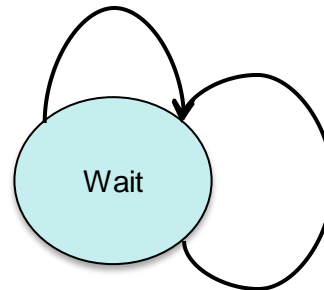
# Extended FSM for a GBN receiver

## Receiver

```
We received a good packet with the expected sequence number  
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt) && has_seqnum(rcvpkt, expected_seqnum)  
-----  
extract(rcvpkt, data)  
deliver(data) // give it to the app  
sndpkt = makepkt(expected_seqnum, ACK, checksum)  
udt_send(sndpkt) // send the ACK to the sender  
expected_seqnum++
```

### Initialize

```
expected_seqnum = 1  
sndpkt = makepkt(0, ACK, checksum)
```



*If we receive anything else, send the last ACK*

```
default  
-----  
udt_send(sndpkt)
```

The receiver discards out-of-order packets

If packet  $n$  is lost and  $n+1$  arrives, the receiver does not buffer packet  $n+1$ .  
The sender will retransmit all unacknowledged packets (go back N).

The receiver has to only keep track of the next sequence number.

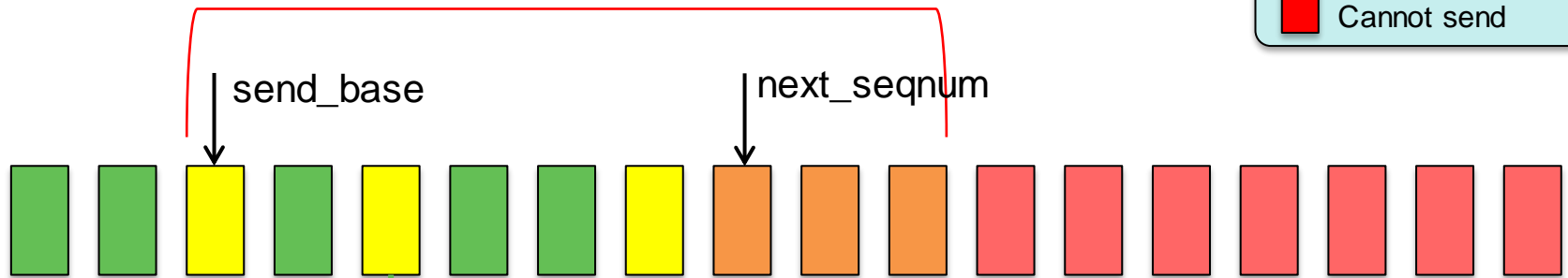
# Selective Repeat

- **Problem with Go-Back-N**
  - With a large window size and large delays, many packets can be in the pipeline
  - A single error can cause GBN to retransmit many packets (all that are unacknowledged)
  - If  $P(\text{channel error})$  increases, the pipeline can become filled with excess retransmissions
- **Selective Repeat Protocol**
  - Retransmit only those packets that were lost or corrupted
  - Receiver must acknowledge each correctly received packet
    - Even if it is out of order
    - Out of order packets must be buffered
  - Window size  $N$  = limit of number of outstanding packets
    - But some packets in the window may be acknowledged
    - The window slides when the earliest packet in the window is acknowledged

# Selective Repeat Windows

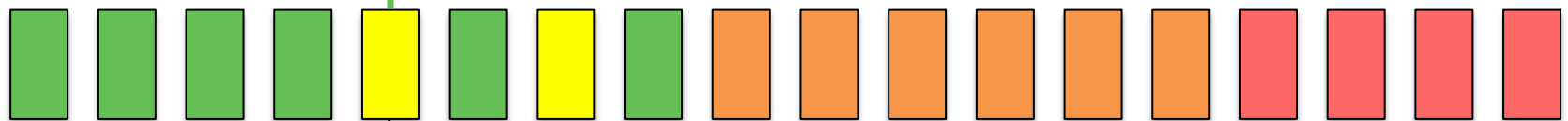
Sender's view of sequence numbers

window size  $N$



Green	Sent & got ACK
Yellow	Sent but no ACK
Orange	Ready to send
Red	Cannot send

Receiver's view



Green	Out of order, ACKed
Yellow	Expected
Orange	Acceptable
Red	Unusable

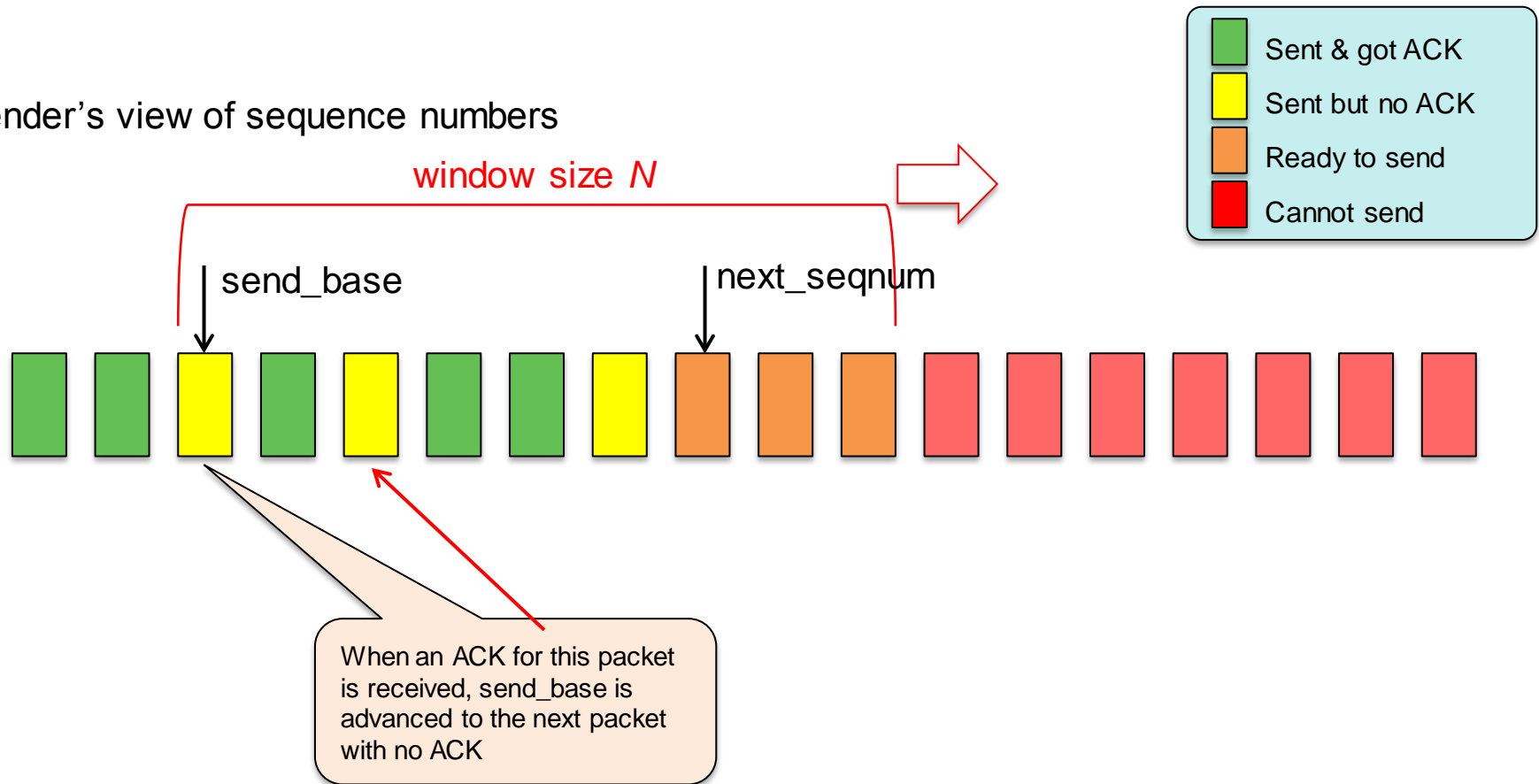
# Selective Repeat: sender operation

- **Send requests from application**
  - Check next available sequence #
  - If no room in window, reject (or buffer)
  - Else send the packet (with sequence #)
- **Timeout**
  - Each packet has its own timer
  - Retransmit only the specific packet on timeout
- **ACK received**
  - If packet is within window
    - Mark packet as received
    - If `sequence # == send_base`  
advance the base (start of window) to the next unacknowledged sequence number



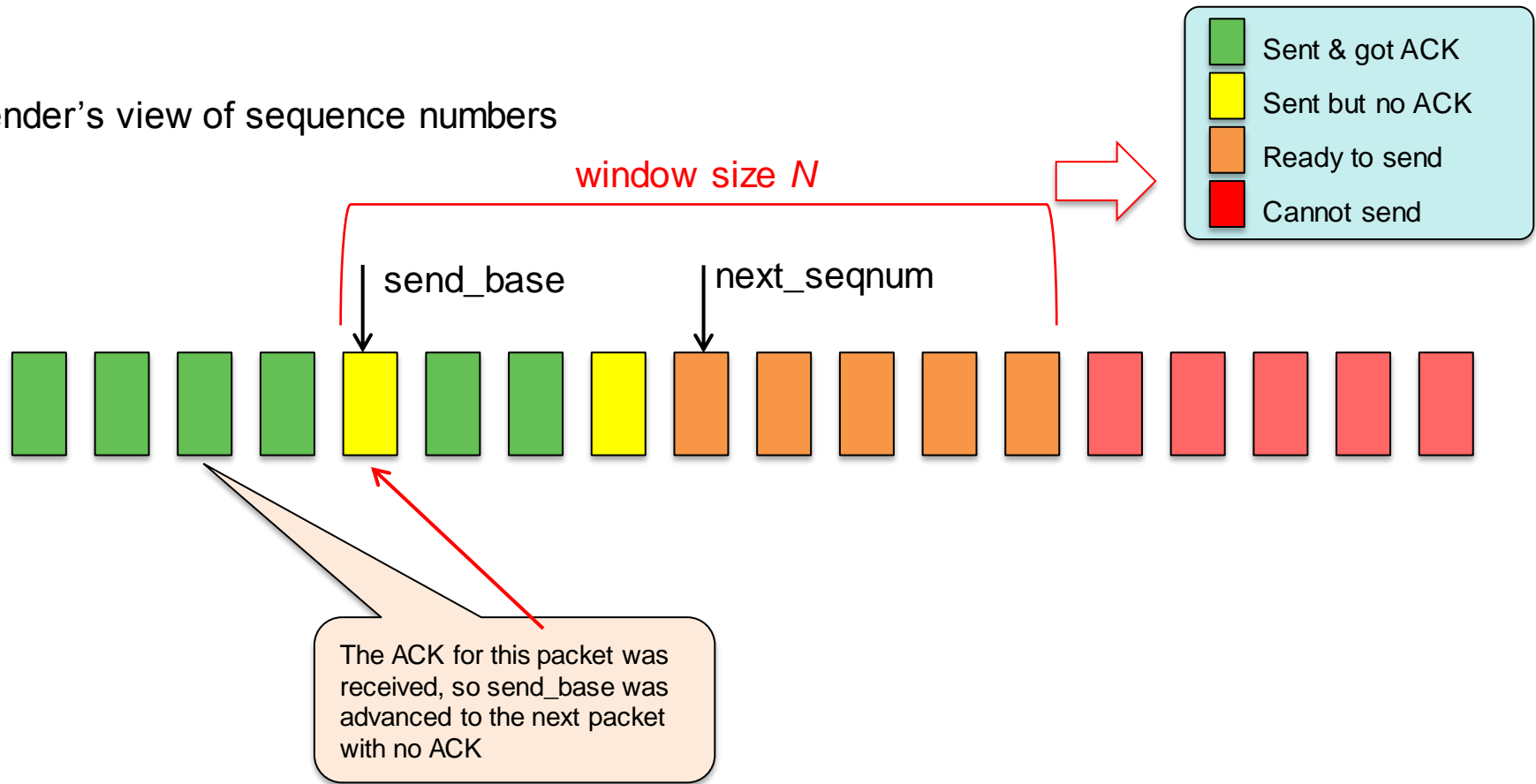
# Out-of-order ACKs

Sender's view of sequence numbers



# Out-of-order ACKs

Sender's view of sequence numbers

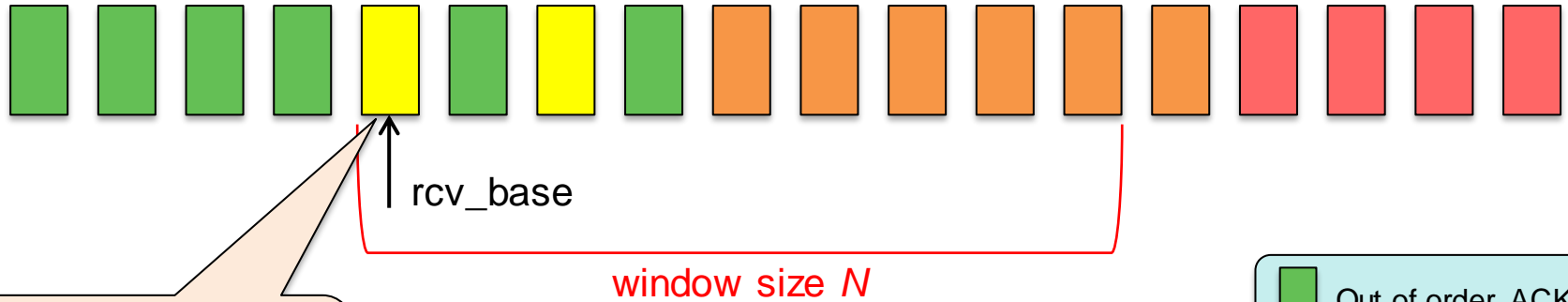


# Selective Repeat: receiver operation





- **Good packet with seq # in  $[rcv\_base, rcv\_base+N-1]$** 
  - Packet is within the receiver's window
  - Send ACK for that sequence #
  - If sequence # ==  $rcv\_base$ 
    - Deliver packet to app and deliver all successive packets that have been received
    - Adjust start of window ( $rcv\_base$ )
- **Good packet with seq # in  $[rcv\_base-N, rcv\_base-1]$** 
  - Packet is within the *before* receiver's window
  - We already saw it – but send ACK anyway
- **Anything else**
  - Ignore the packet

# Selective Repeat: receiving packets

Receiver's view

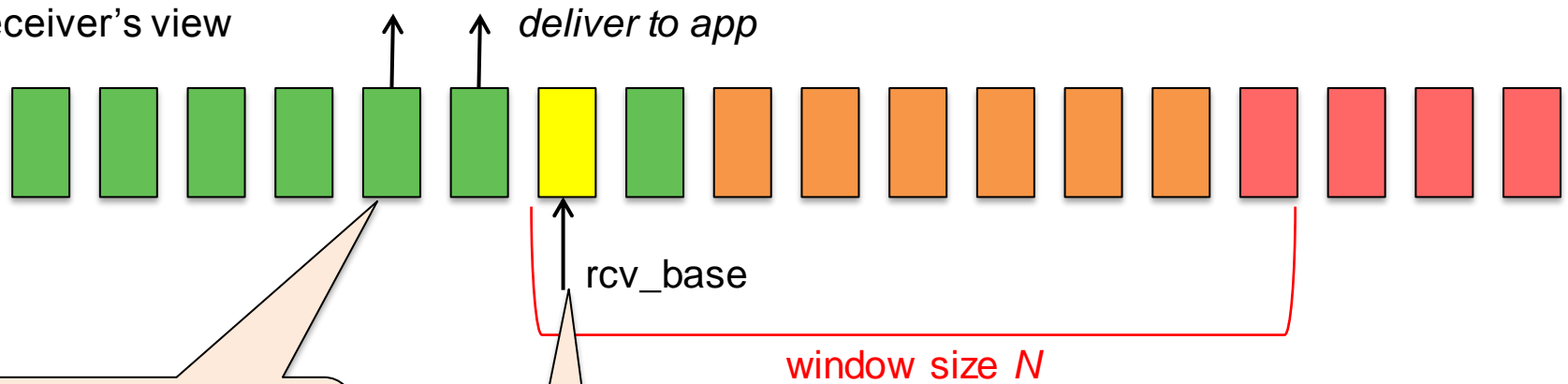


When this packet is received, we can deliver it to the app and deliver all received packets immediately after it

	Out of order, ACKed
	Expected
	Acceptable
	Unusable

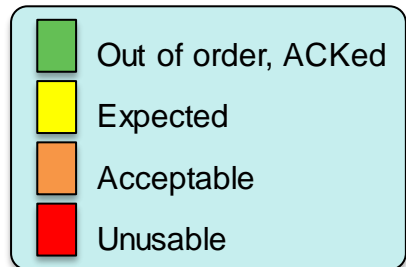
# Selective Repeat: receiving packets

Receiver's view



This packet was received, so we delivered it and all received packets immediately after it.

The start of the window (base) is moved to the first missing packet. The start of the window on the receiver is not always the same as the start of the window on the sender.



The end