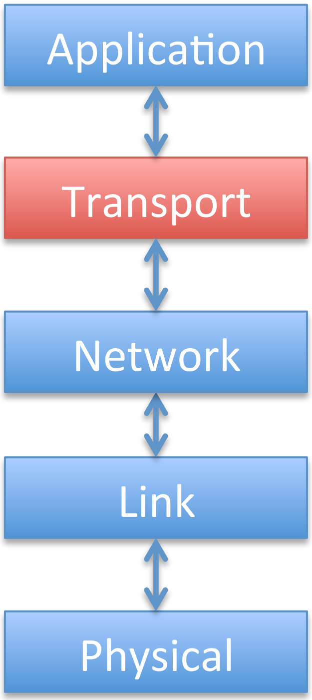
Computer Networks and Communication

Lecture 5

Transport Layer,
UDP Protocol,
Reliable Data Transfer

Transport Layer

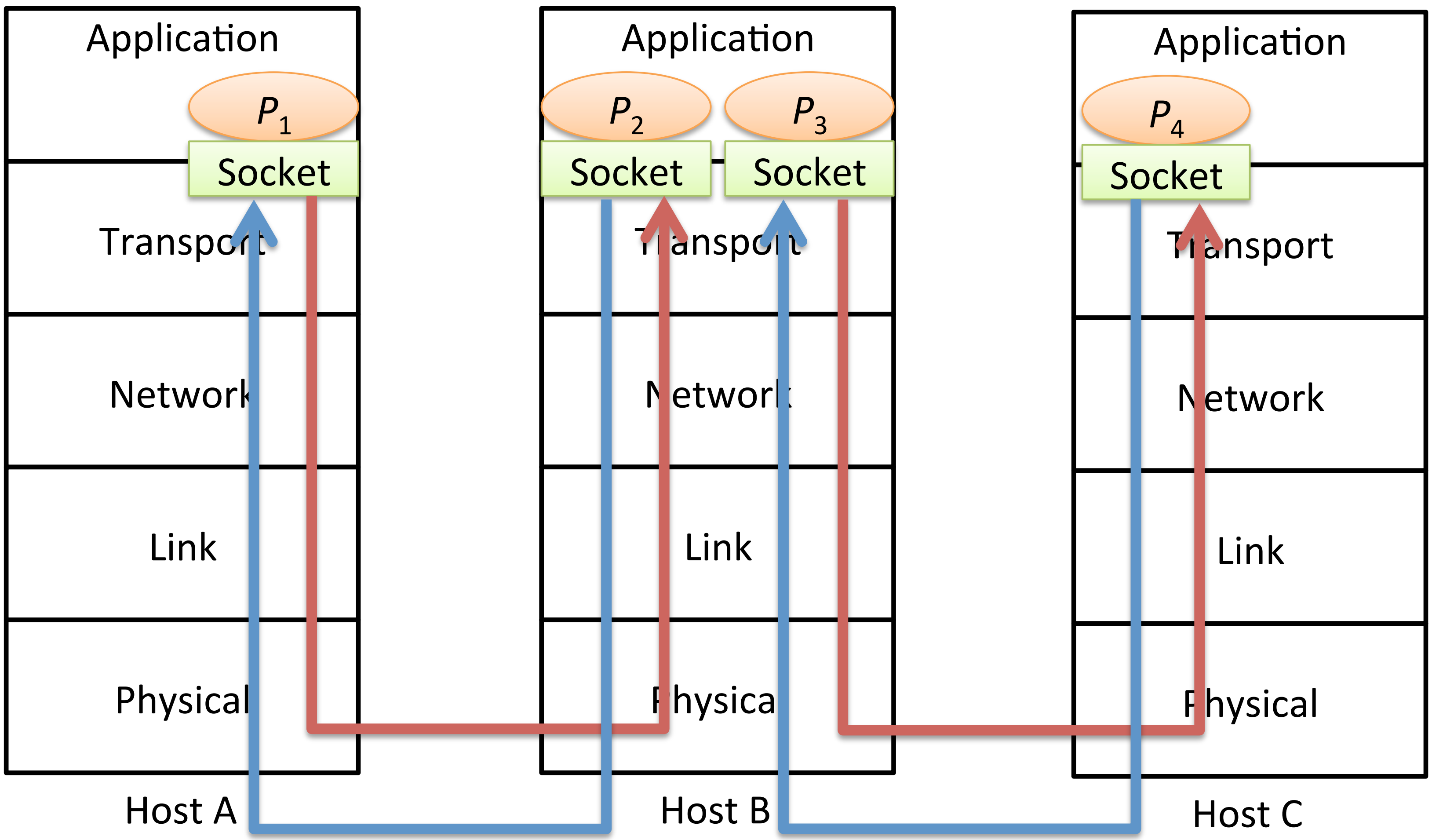
- Resides between the application layer and the network layer
- Provides for **logical communication** between processes on different hosts
- Packets in transport layer are called **segments**
- TCP and UDP operate in this layer



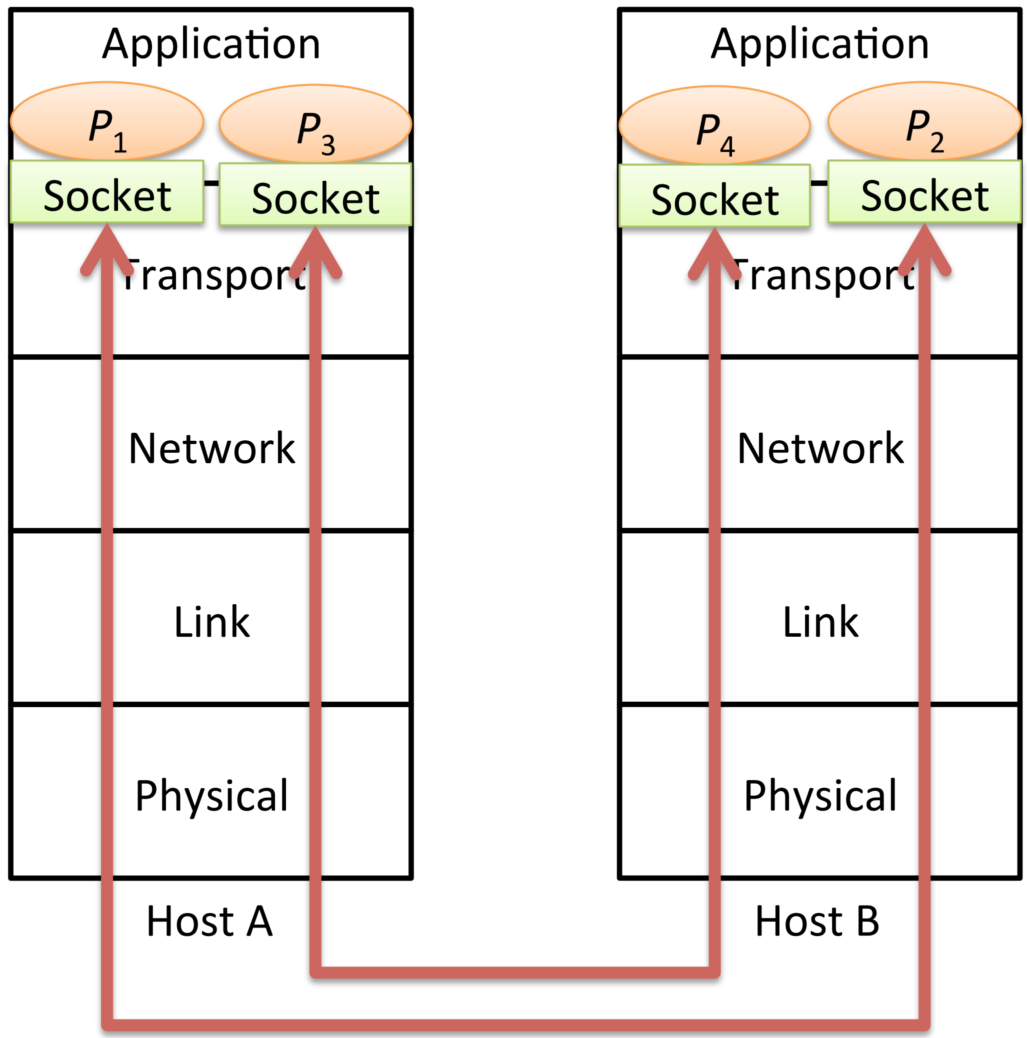
Transport Layer (2)

- There can be many processes running on a single host
- Hence, if a process P_1 in host A wants to communicate with a process P_2 in host B
  - P_1 has to know both IP address of B and the port number associated to P_2
  - P_2 has to know IP address and port number of P_1 as well
- **Process-to-process data delivery** is the main service of transport layer
  - **Multiplex** / **Demultiplex**

Process-to-Process Communication



Process-to-Process Communication (2)



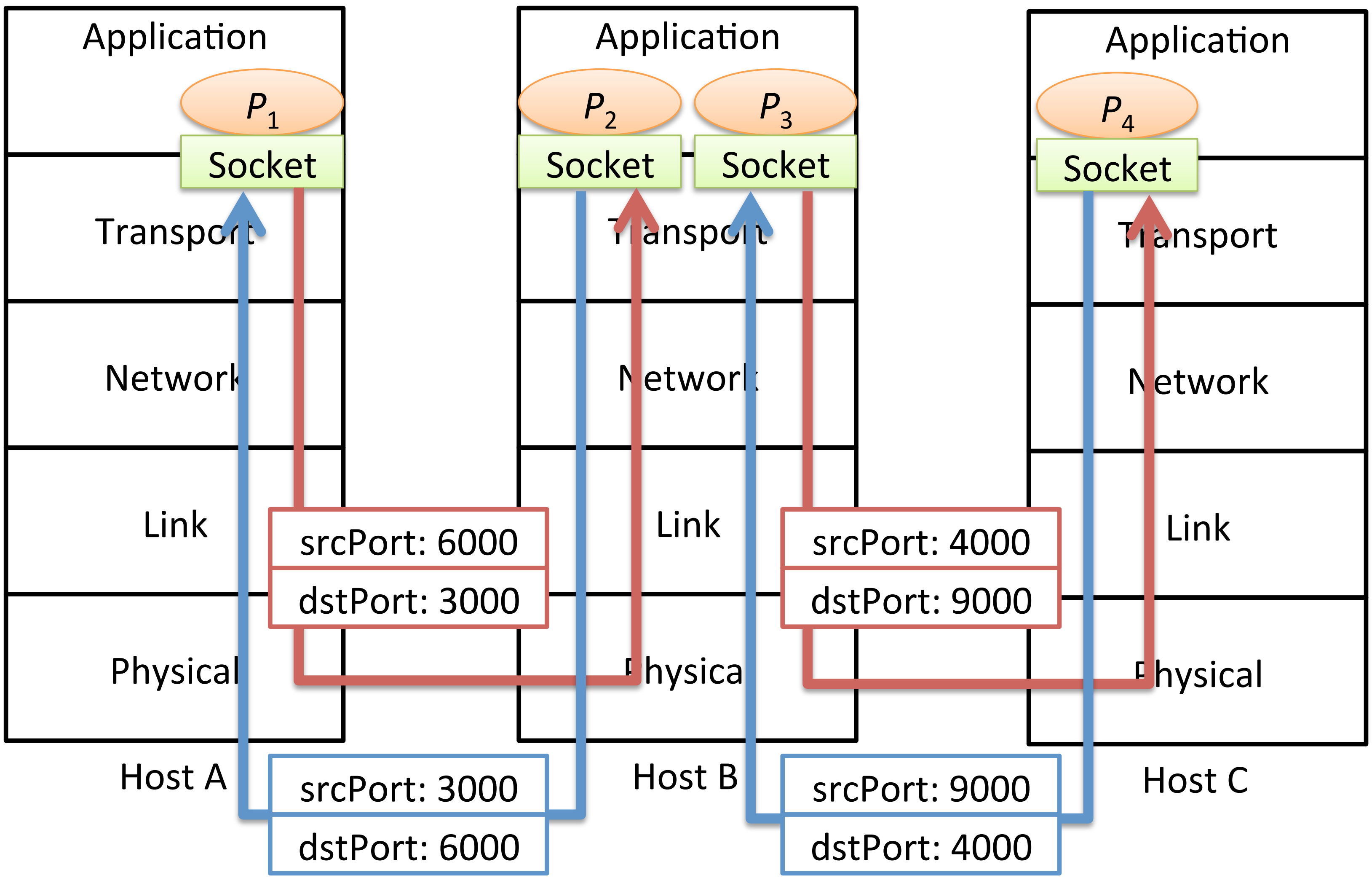
The 5-Tuple

- Process-to-process communication can be distinguished by
  - Source IP (SrcIP)
  - Source port (SrcPort)
  - Destination IP (DstIP)
  - Destination Port (DstPort)
  - Transport protocol (e.g. TCP and UDP)
- Packet sender and receiver can identify each other using these attributes
- We call the these attributes together the **5-tuple**

Specified in network-layer header

Specified in transport-layer header

Source and Destination Ports



Well-Known Ports

- With port numbers, we can specify which process we want to communicate with
- But how do we know which port numbers are associated to which processes in the distant host?
- To this end, some important applications have specific port numbers assigned to them
  - We call those port numbers **well-known ports**
  - Standardized in **RFC 1700** by **Internet Assigned Numbers Authority (IANA)**

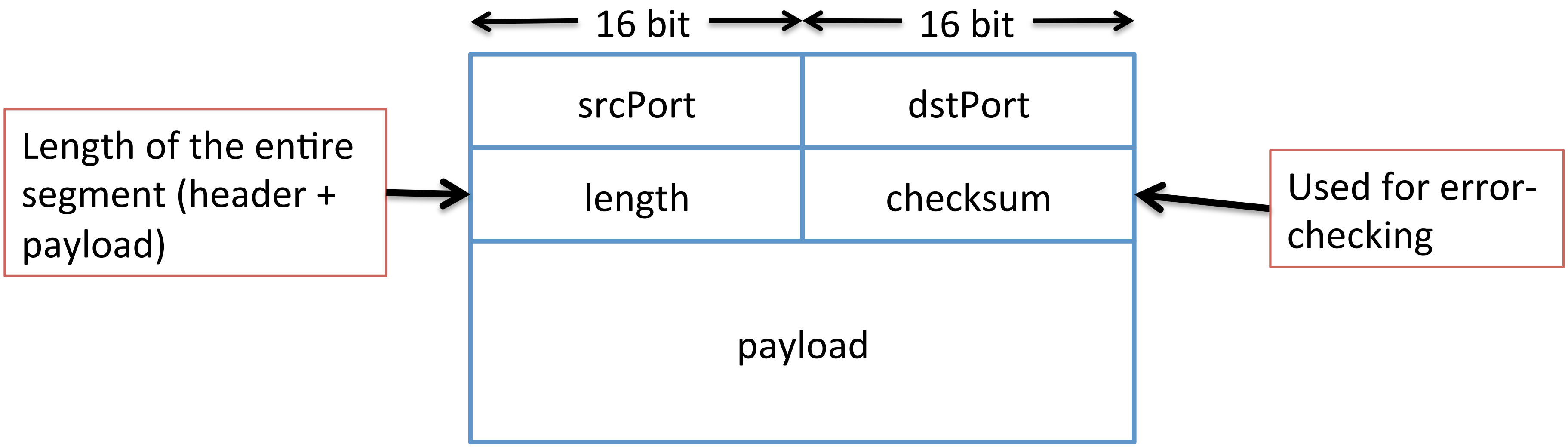
Well-Known Ports (2)

- Highest port number is: **65535** (Why?)
- Standardized well-known ports are ranged from ports **0 to 1024**
- Well known ports example:
  - **7**: ECHO
  - **20** and **21**: FTP data and control respectively
  - **22**: SSH
  - **53**: DNS
  - **80**: HTTP
  - **110**: POP3
  - **547**: DHCP Server
- There are other well-known ports above 1024 too but they are not specified in the standard

Data Transfer with UDP

- Application can control packet-sending speed
  - No congestion control
  - No packet-retransmission
- Fast
  - No handshaking / connection establishment
  - Small protocol header
- Provides simple error-detection
- Example applications:
  - DNS
  - Videoconference software
  - First-person shooting games

UDP Header



- Header size: 8 byte
- Payload size:
  - Min: 0 byte
  - Max: 65,527 bytes

UDP Checksum

- Checksum is a simple error-detection mechanism
- In UDP, checksum is **optional**

Sender

- Divide the **entire segment** into a sequence of 16-bit **words**
- Compute the sum of all words
  – Add the words to each other
- Perform 1s complement of the sum
- If the sum is 0xFFFF, then ignore the 1's complement (which is 0x0000)
- The result is then stored in the checksum field

Receiver

- Compute the sum of the received segment
- Compare the computed checksum and the one in the checksum field
  – They are equal: No error
  – Not equal: Error detected

UDP Checksum (2)

1110011001100110	1101010101010101
0000000000001000	

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0	16 bit word
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	16 bit word
1	1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1	sum
	1	wraparound
	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0	

UDP Checksum (3)

- Exercise: Compute the sum

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0	16 bit word
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	16 bit word
1	1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1	sum
	1	wraparound
	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0	sum
	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	16 bit word
	1 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0	sum
	0 1 0 0 0 1 0 0 0 0 1 1 1 0 1 1	**checksum**

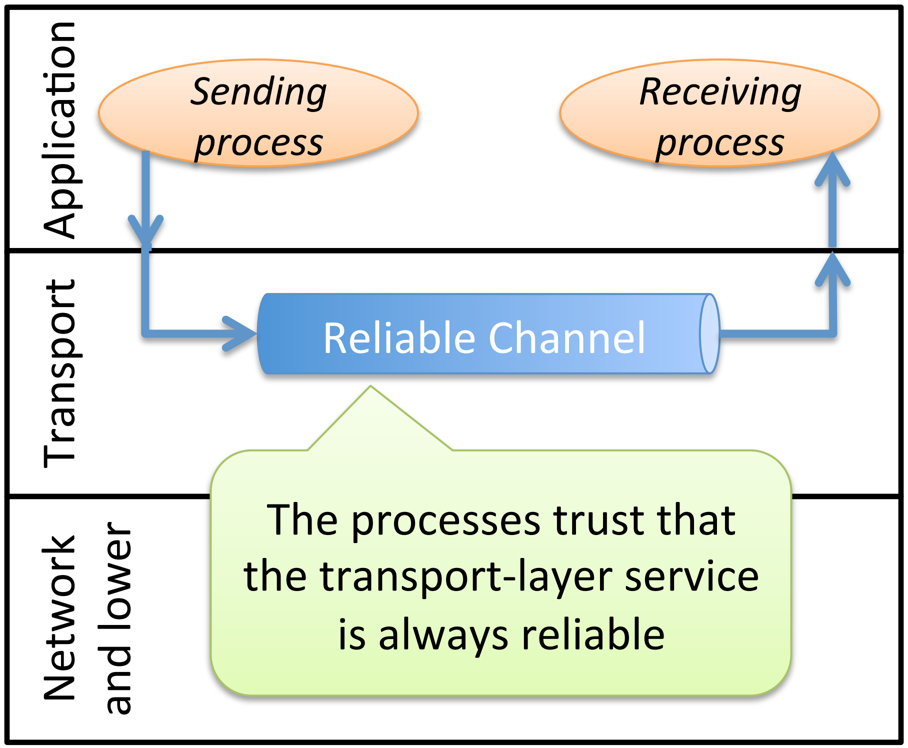
1's compliment

Checking the Checksum

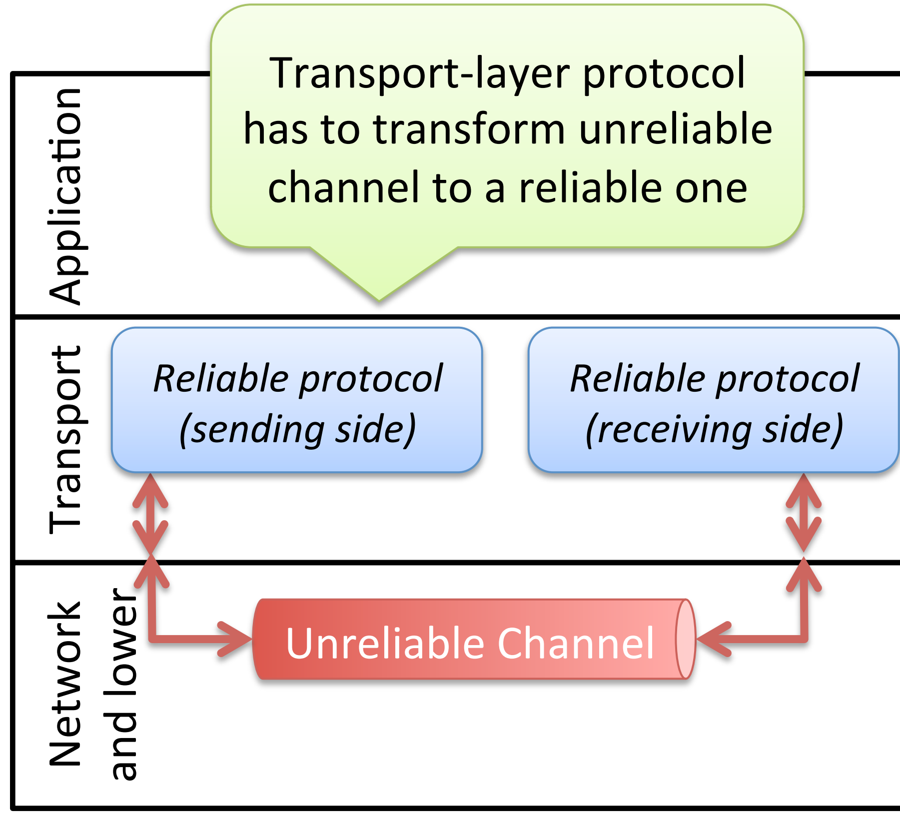
1110011001100110	1101010101010101
0000000000001000	0100010000111011

- What is the sum of all words (with wraparound)?:
  - 1110011001100110
  - 1101010101010101 Segment words
  - 0000000000001000
  - 0100010000111011 Checksum
- It is: 1111111111111111 **Why?**
- With checksum, can we detect all possible errors?
- Can UDP detect packet lost or out-of-order?
- UDP Checksum is optional. Why it is so?

Reliable Data Transfer



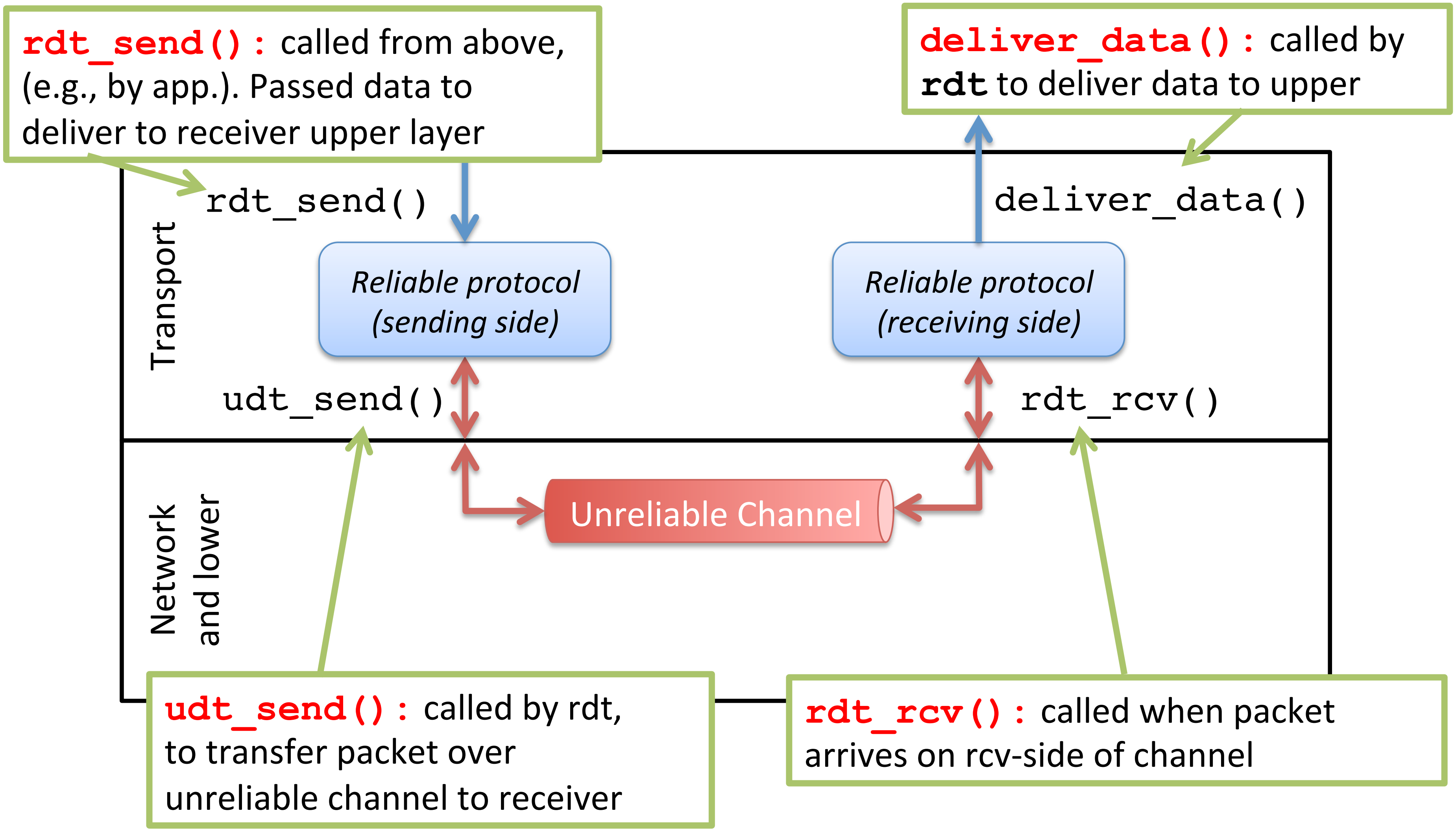
Provided service



Service implementation

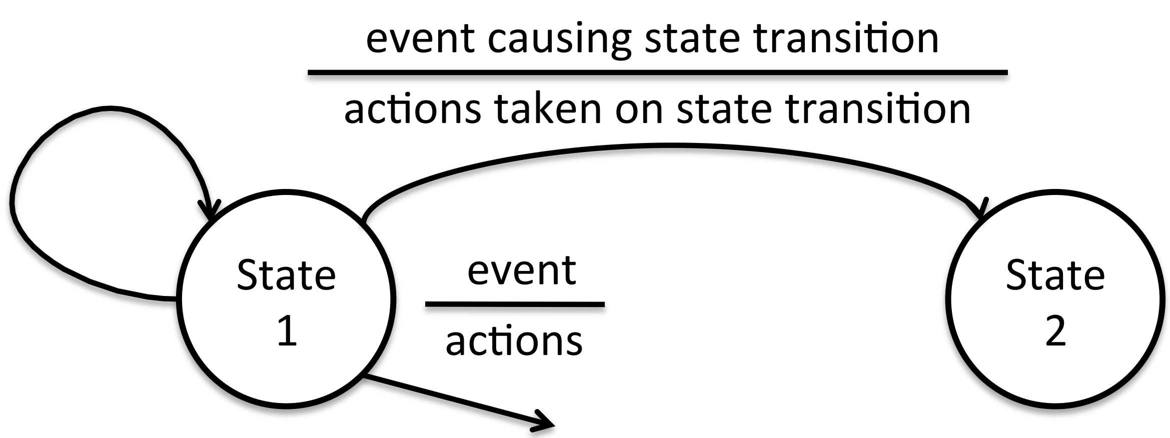
- We are going to build a **reliable data transfer protocol (rdt)**

Reliable Data Transfer (2)



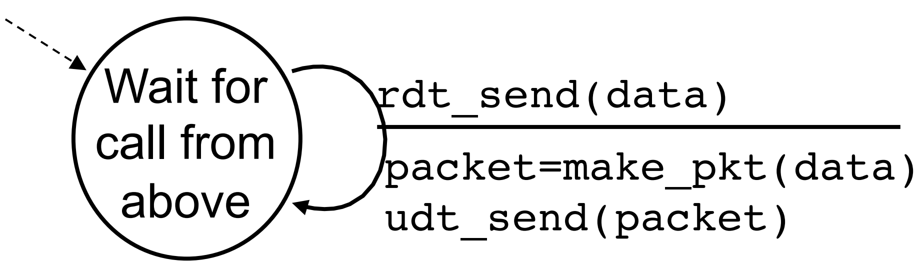
Reliable Data Transfer (3)

- We will incrementally develop sender and receiver sides of the rdt protocol
- The data transfer will be **unidirectional**
  - Application data will be transferred one-way
  - Control data will be transferred in both direction
- We will use **finite state machine (FSM)** to model the operations in both sides

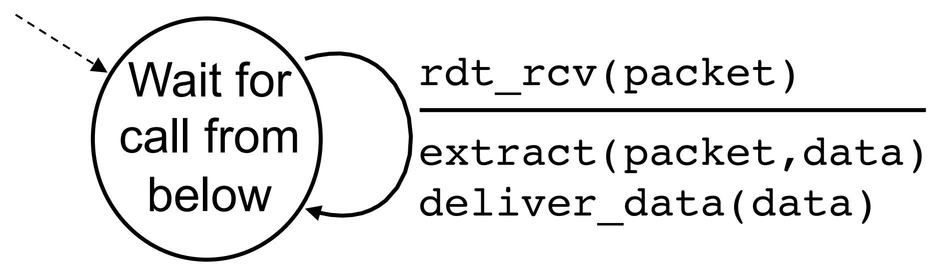


rdt 1.0

- Reliable transfer over a ***reliable*** channel
- Underlying channel is reliable
  - No errors
  - No packet loss
- Separate FSM for sender and receiver
  - Sender keep sending the data
  - Receiver keep receiving data



sender

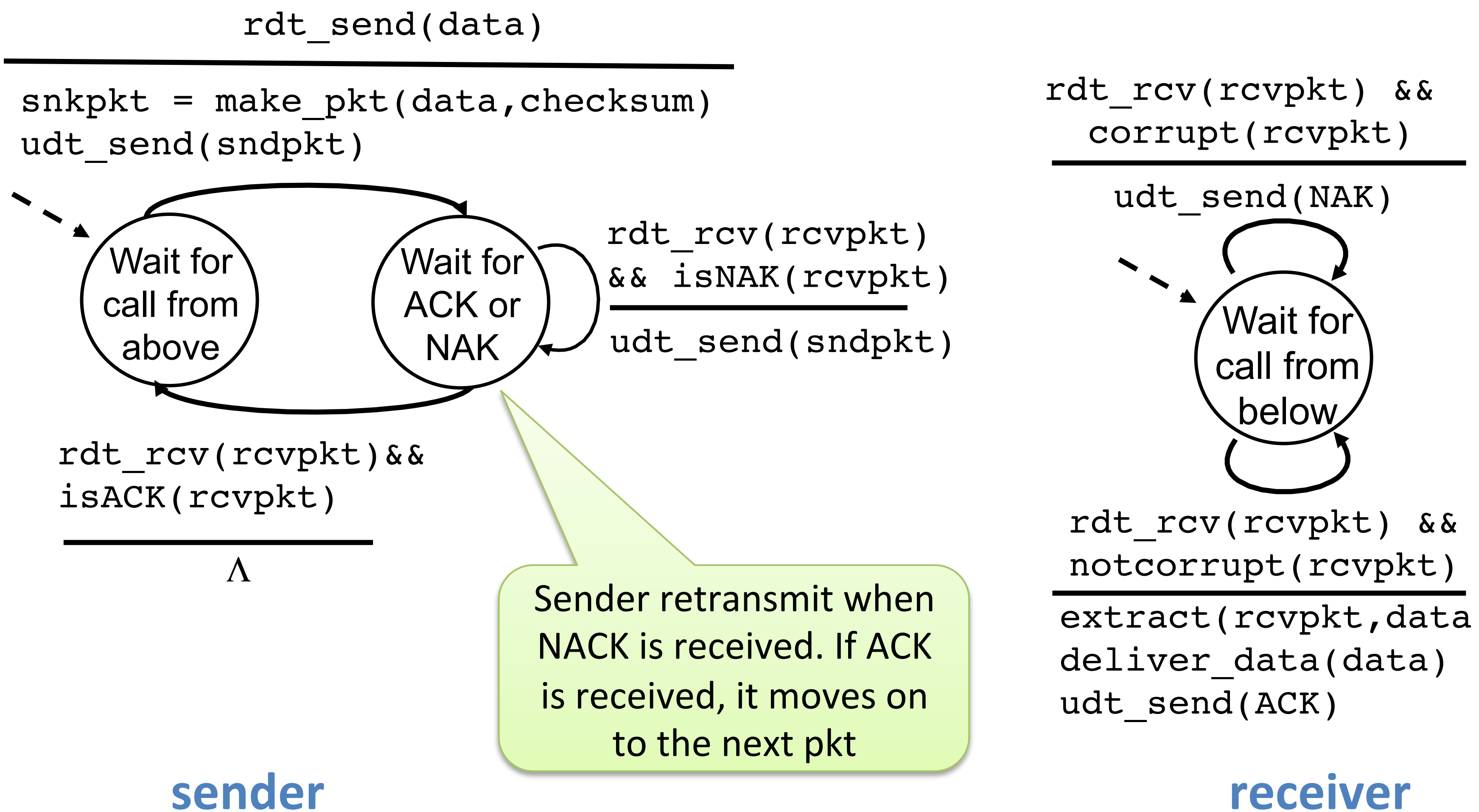


receiver

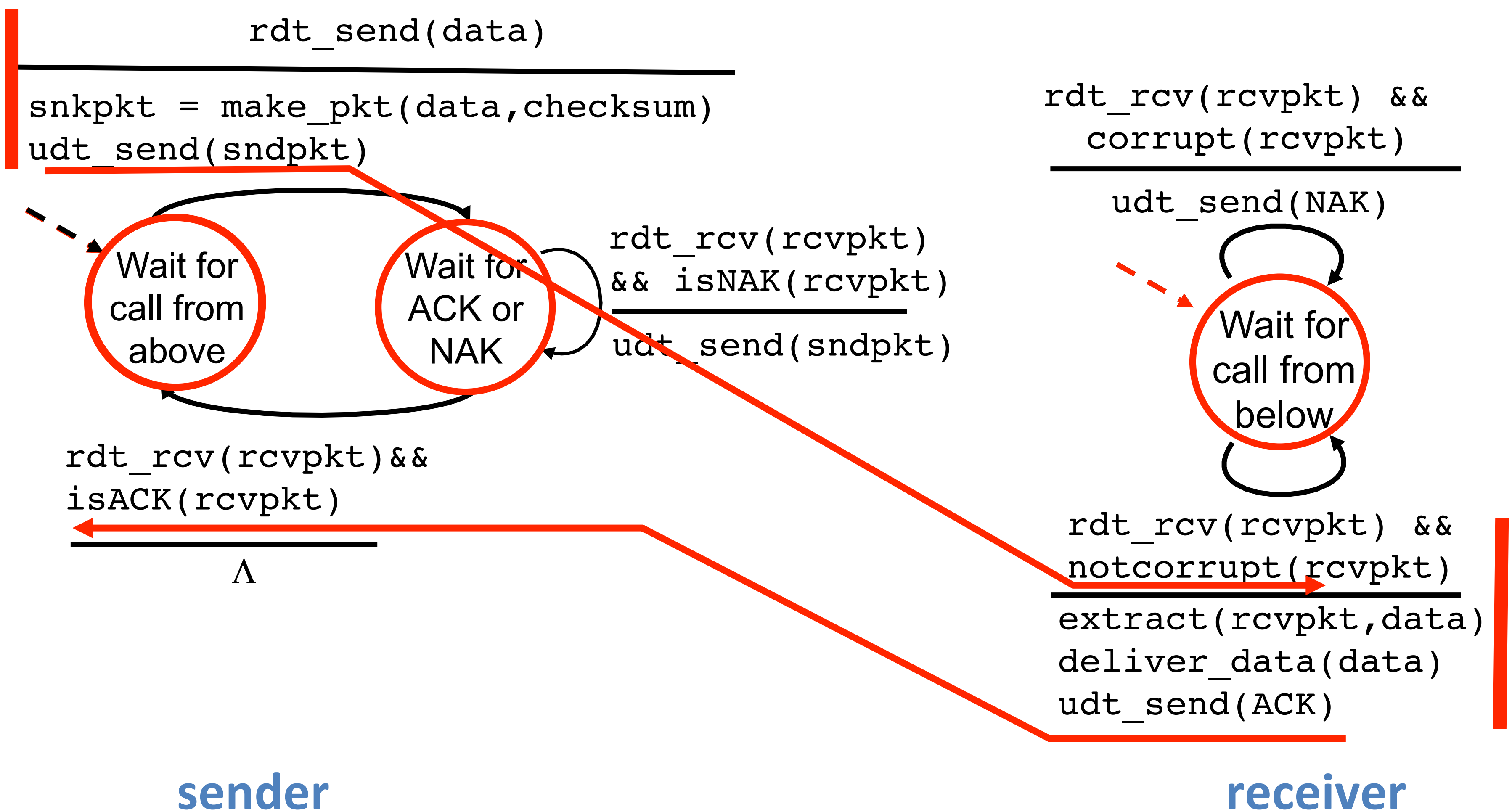
rdt 2.0

- In reality, underlying channel is ***not*** reliable
  - We can use the **checksum** to detect errors
- Error recovery
  - Acknowledgement (**ACK**): The receiver tells the sender that the packet is correctly received (OK)
  - Negative ACK (**NACK**): The receiver informs that the packet had errors
  - Sender **retransmit** the packet after hearing NACK
- rdt 2.0 improvements over rdt 1.0
  - Error detection (at the receiver side)
  - Receiver feedback (ACK / NACK)

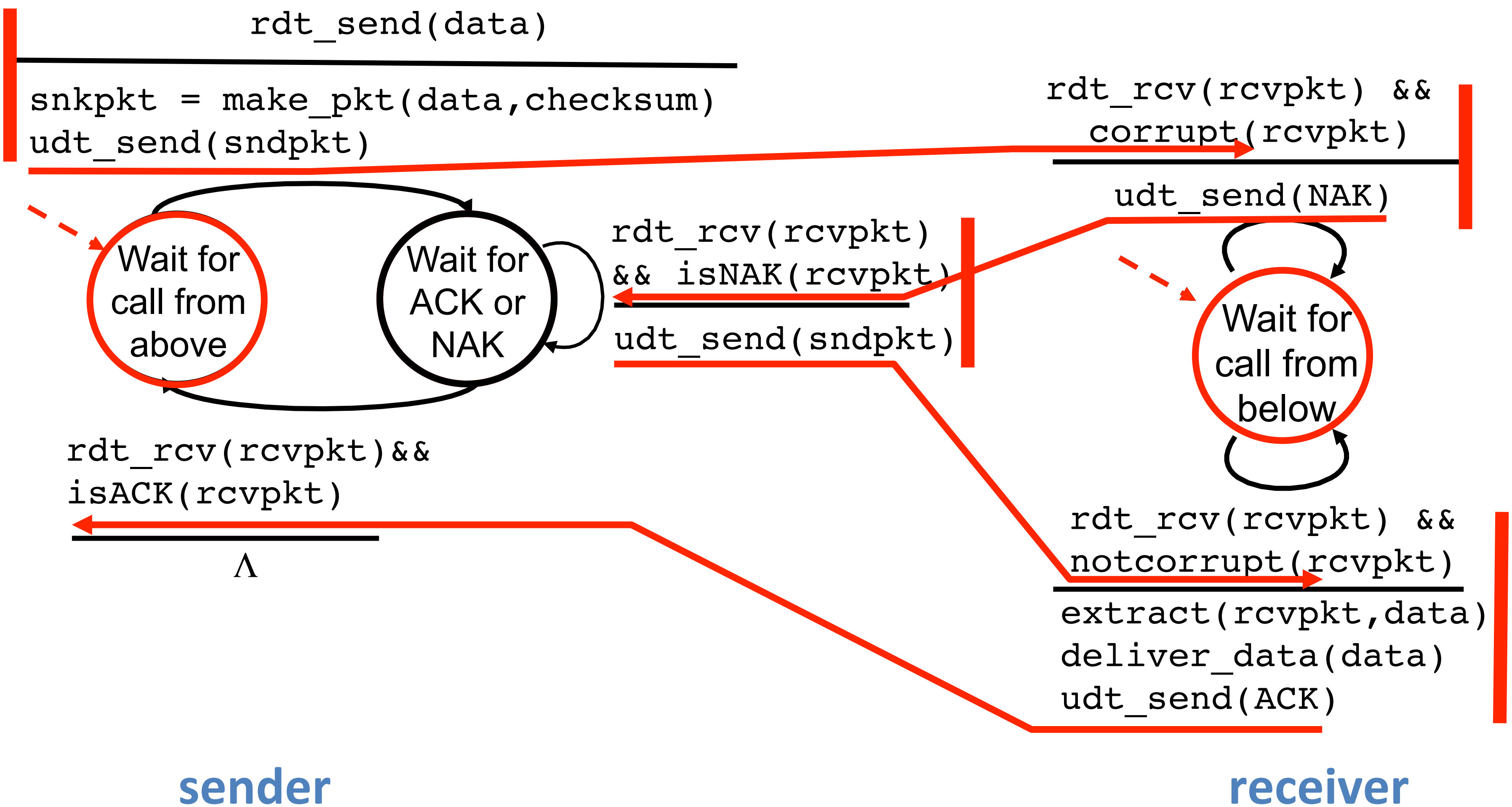
rdt 2.0 - FSM


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
Sender retransmit when NACK is received. If ACK is received, it moves on to the next pkt
sender
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)
receiver

rdt 2.0 – FSM without Errors



rdt 2.0 – FSM with Errors



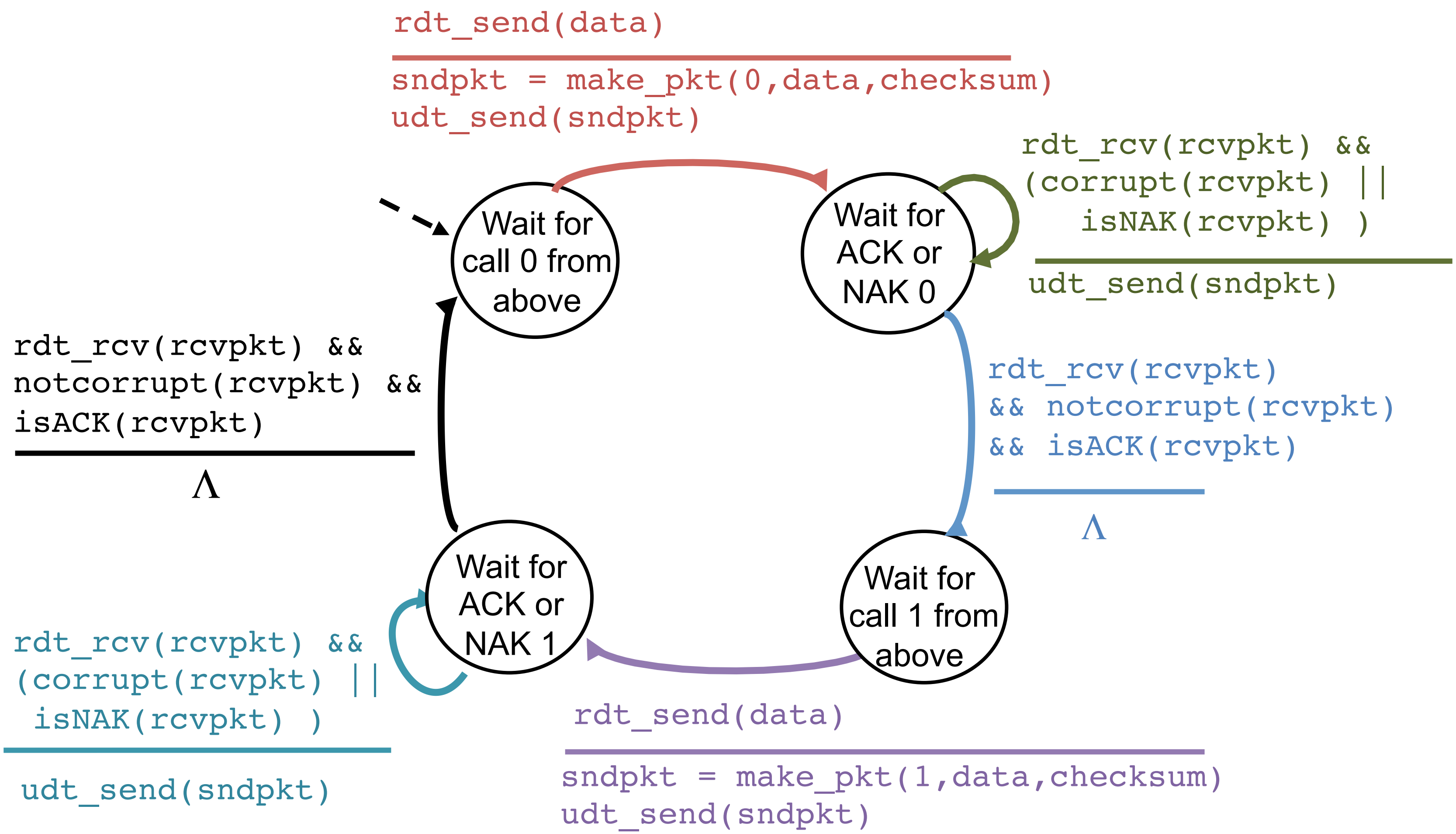
rdt 2.0 - Discussion

- Sender always wait for feedback from receiver
  - Feedback: **ACK**/**NACK**
  - **Stop-and-wait** protocol
- Receiver detects errors using **checksum**
- Problems:
  - What if ACK/NACK got lost or corrupted?
  - Can the sender still know if the packet is received correctly?
  - Any idea?

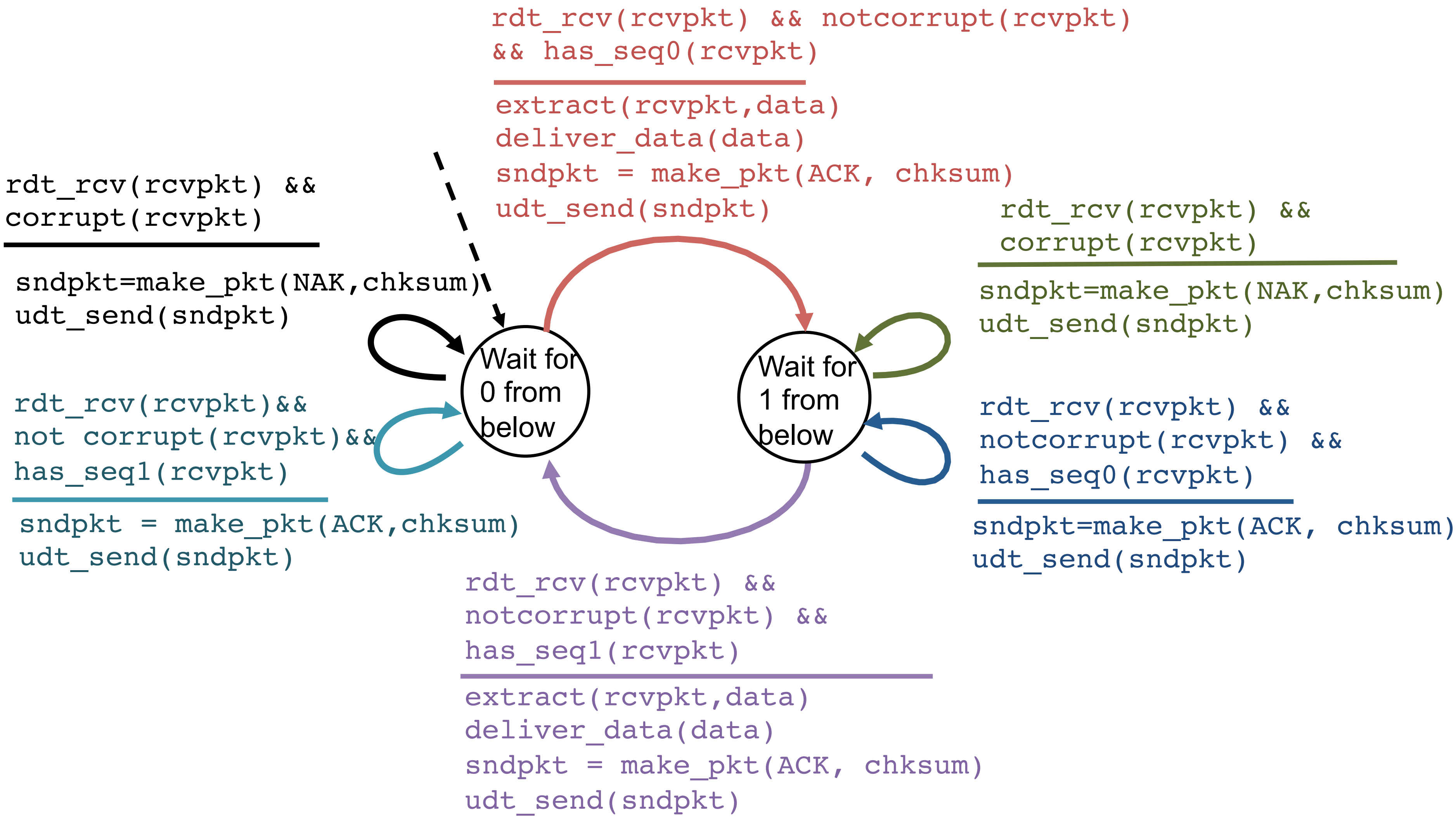
rdt 2.0 – Discussion (2)

- Possible solutions:
  - The sender keeps asking for ACK
    - Receiver might get confused
  - Use extra info (more than checksum), so that the sender can reconstruct correct feedback
    - Extra overhead
  - Sender simply resend the packet if the ACK is not received
    - **Duplicate**: Receiver might not know if the resent packet is a retransmitted packet or a new packet
- Another solution: Add **sequence numbers** into data packets

rdt 2.1 – FSM: Sender Side



rdt 2.1 – FSM: Receiver Side



rdt 2.1 – Discussion

- Sender
  - Added seq# to packets
  - Two sequence numbers, 0 and 1 will suffice. Why?
  - Must check if received ACK/NAK is corrupted
  - Number of states are twice more than rdt 2.0
  - State must remember whether current packet has 0 or 1 seq#
- Receiver
  - Must check if the received packet is duplicate
  - State specifies expected packet seq#
  - Receiver cannot know if the ACK/NACK is received correctly by the sender

rdt 2.1 – Discussion (2)

- Sender always wait for feedback from receiver
  - Feedback: **ACK**/**NACK**
- Receiver detects errors using **checksum**
- Receiver determines if the incoming packet is a retransmission or a new packet using **sequence number**
  - Solution to duplicate-packets problem
- Problem:
  - Sending both NACK and ACK brings additional overhead
  - What if the ACK or NACK is lost along the way?

rdt 2.2 – NAK-free Protocol

- Same functionality as rdt 2.1 but using only ACKs
- The receiver adds seq# to ACK, indicating which packet is corresponding to this ACK
  - e.g.: ACK 1 is an acknowledgement for packet with seq# 1
- **Duplicate ACK** (e.g. "ACK 1" twice) would result in the same action as "NAK"
- Like rdt 2.1, it does not work properly if the underlying channel can lose packets

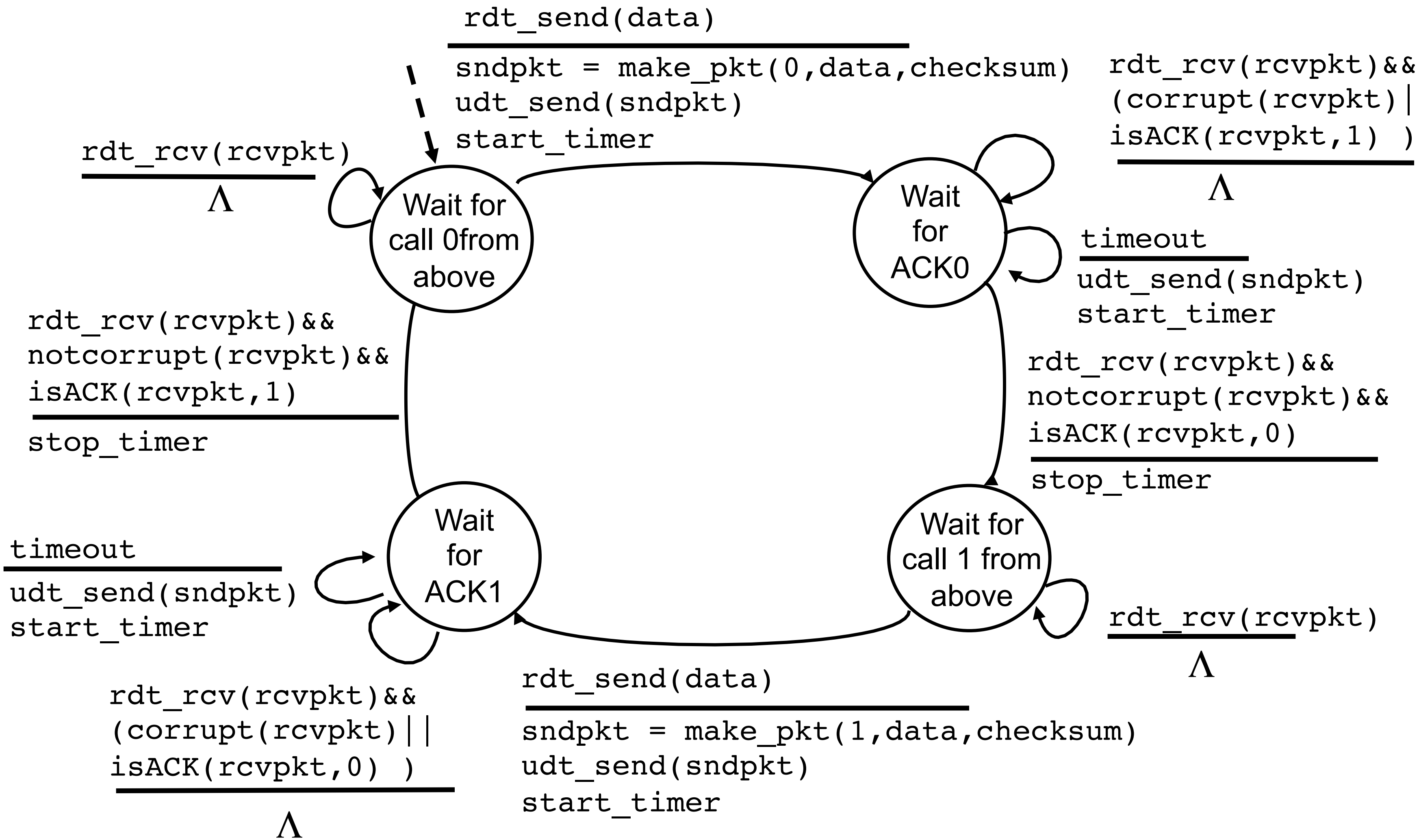
rdt 3.0

- Underlying channel may cause errors and can lose packets
  - **Checksum**: Detect errors
  - **Retransmission**: correct errors
  - **Seq#**: Detect duplicates
  - None of those can detect packet loss
- What would you do if N'Toey does not return your mails?
  - Your mail might be lost?
  - Her mail might be lost?
  - You should get lost?

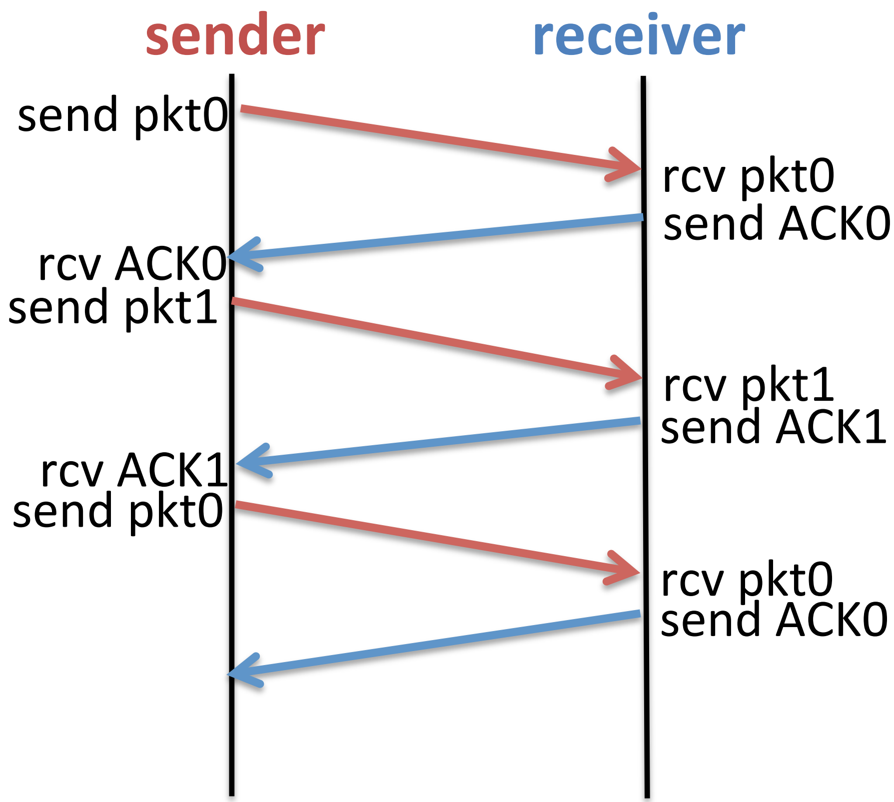
Handling Packet Loss

- Sender waits for "reasonable" amount of time for ACK
  - The transmitted packet might be lost
  - The ACK might be lost
- It retransmits if no ACK arrives in this time
- If the packet or ACK is delayed, the retransmitted packet would be duplicate
  - Seq# already handles this
  - Receiver must specify seq# in the ACK
- This approach requires countdown timer

rdt 3.0 – FSM: Sender Side

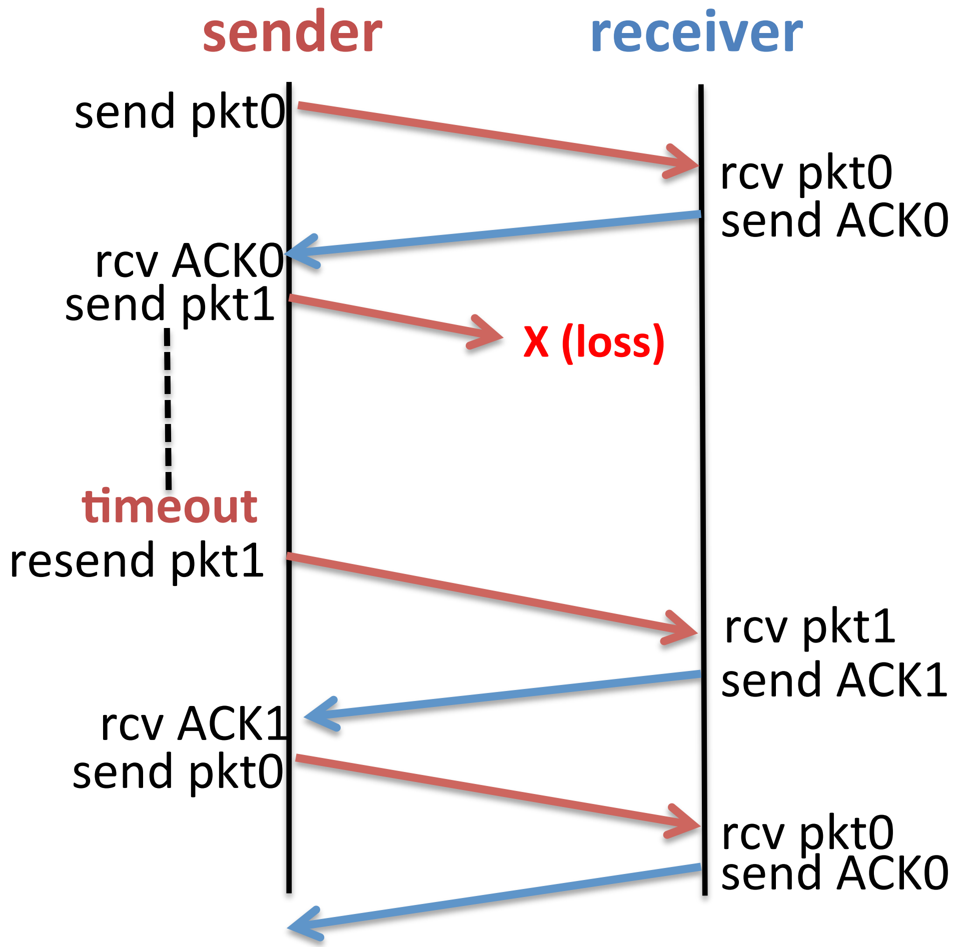


rdt 3.0 in Action



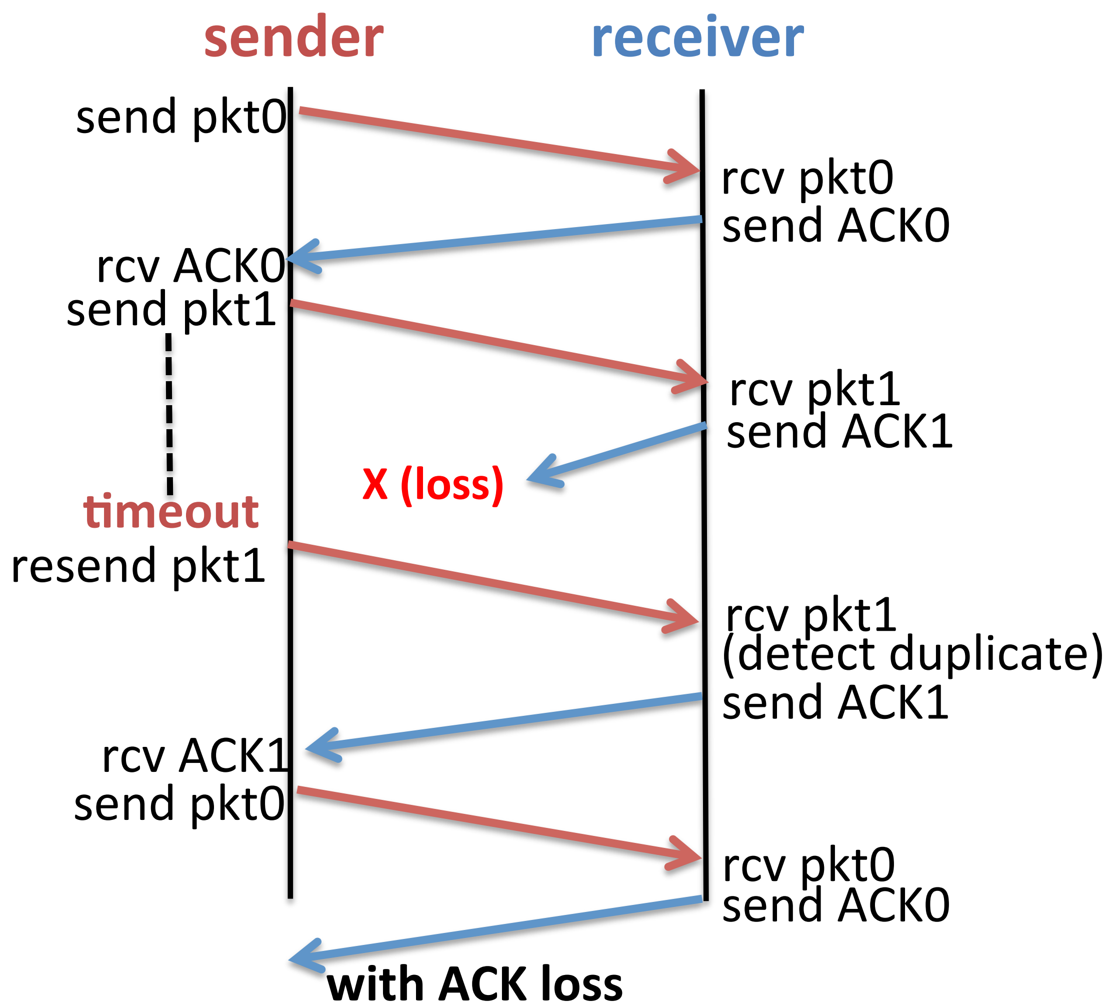
no packet loss

rdt 3.0 with Packet Loss

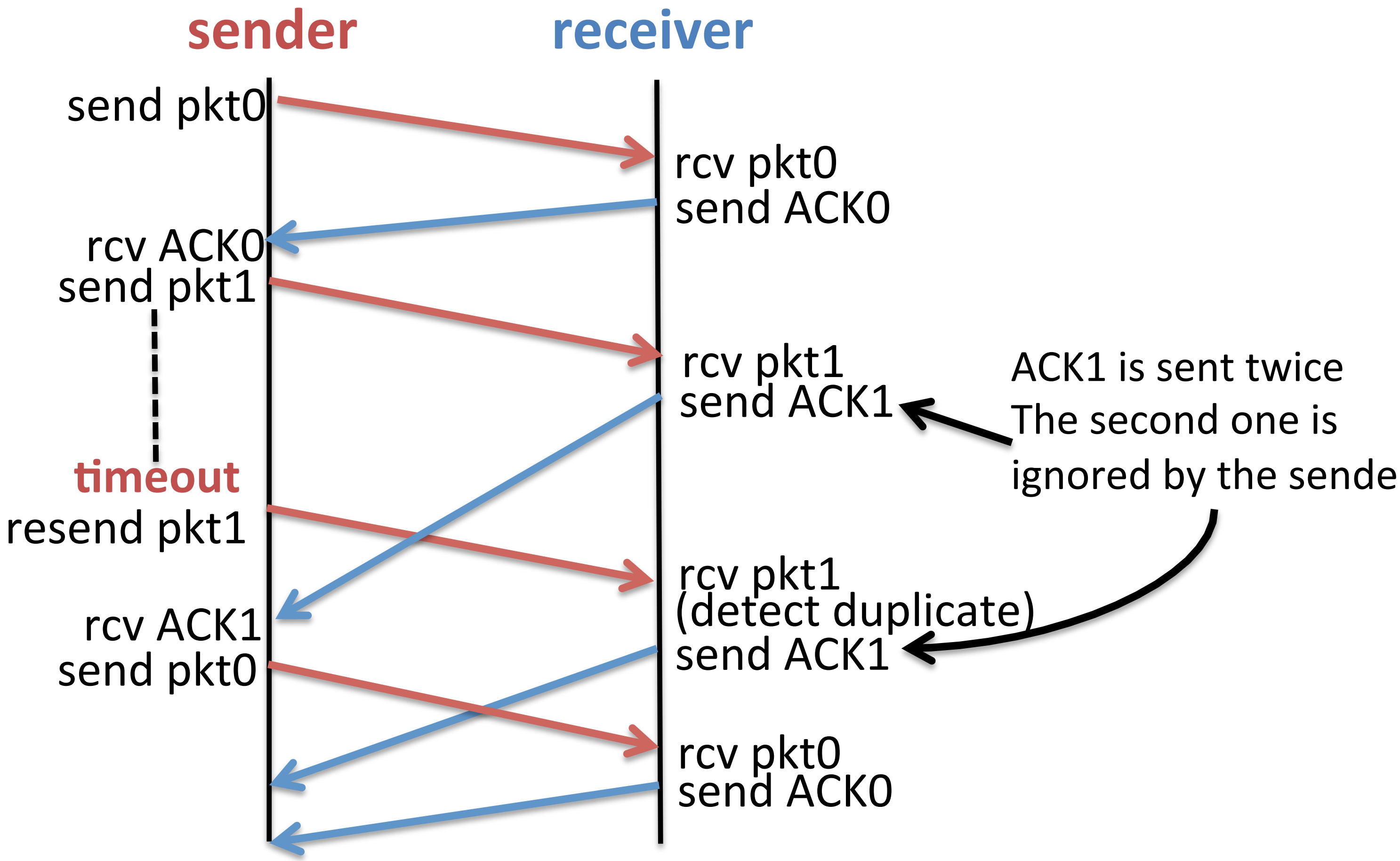


with packet loss

rdt 3.0 with ACK Loss



rdt 3.0 with Premature Timeout



with too short time out

rdt 3.0 – Discussion

- rdt 3.0 would work in general
- It detects error, duplicates and packet loss
- However, it is extremely slow
  - Stop-and-wait protocol
  - Every time it is going to send a packet, it has to wait for the ACK of previous packet
  - Round trip time (RTT) between sent packet and the ACK is the culprit
- Solution: **Pipelining**