CSC 4350 – Computer Networks

Lecture 11 – UDP, Reliable Data Transfer

October 3, 2024

Note

- Material in this lecture is heavily borrowed from Kurose & Ross' "Computer Networking: A Top Down Approach, 8th Edition"

Connectionless Transport – UDP

- UDP Segment Structure
- UDP Checksum

UDP

- Does just about as little as a transport protocol can do
  - Only multiplexing/demultiplexing and a little bit of error checking
- If chosen over TCP, then the application is almost directly talking with IP
- Takes messages from the application process, attaches source and destination port number fields for multiplexing/demultiplexing service, adds two other small fields and passes resulting segment to the network layer
- Network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt at delivery
- If segment arrives at the receiving host, UDP uses the destination port number to deliver the segment’s data to the correct application process
- No handshaking between sender, receiver, so deemed as connectionless

UDP

- DNS is an example of an application-layer protocol that uses UDP
- When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to UDP
- UDP adds header fields to the message and passes resulting segment to the network layer
- Network layer – encapsulates segment into a datagram, sends datagram to a name server
- DNS application at the querying host then waits for a reply to its query
  - If no reply, it might try resending the query or try sending the query to another name server, or let invoking application that it can’t get a reply

UDP Better Suited Applications?

- Finer application-level control over what is sent and when
  - As soon as an application process passes data to UDP, UDP will package the data inside a segment and immediately pass segment to network layer
  - TCP: congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between source and destination hosts become congested
  - TCP will also continue to resend a segment until receipt acknowledged by destination
  - Real-time applications often require a minimum sending rate
    - Can tolerate some data loss
    - TCP isn’t a good match
    - Applications can use UDP and implement any additional functionality that is needed beyond UDP’s delivery service

UDP Better-Suited Applications?

- No connection establishment
  - UDP does not introduce delay to establish a connection like TCP
  - Probably why DNS runs over UDP and not TCP (slow-down)
  - http uses TCP since reliability is critical; connection-establishment delay is a contribution to delays associated with downloading web documents
- No connection state
  - TCP maintains connection state in the end systems
    - Receive/send buffers
    - Congestion-control parameters
    - Sequence and acknowledgment number parameters
  - UDP – does not
  - A server devoted to a particular application can typically support many more active clients when it runs over UDP
- Small packet overhead
  - TCP segment – 20 bytes of header overhead in every segment
  - UDP – 8 bytes of overhead
- Note: it is possible for an application to have reliable data transfer with UDP, if reliability is built into the application itself (QUIC, being one)

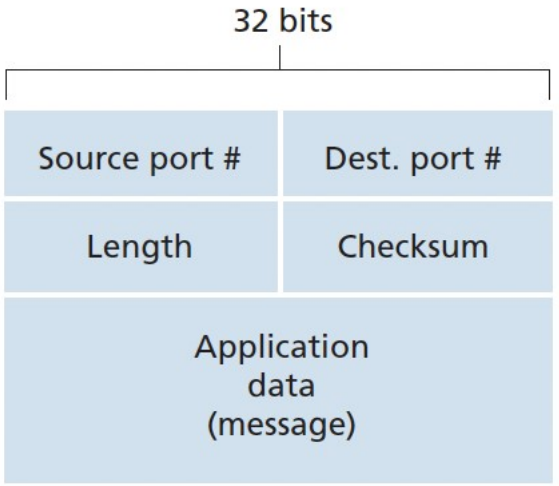
Figure 3.6 – Popular Internet Applications and Their Underlying Transport Protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Secure remote terminal access	SSH	TCP
Web	HTTP, HTTP/3	TCP (for HTTP), UDP (for HTTP/3)
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	DASH	TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

UDP Segment Structure

- Figure 3.7 – UDP segment structure
  - Defined in RFC 768
  - Application data occupies the data field of the UDP segment
  - DNS – data field contains either a query message or response message
- UDP header has only four fields, each consisting of two bytes
  - Port numbers allow the destination host to pass the application data to the correct process running on the destination end system
  - Length – number of bytes in the UDP segment; length value is needed since the size of the data field may differ from one UDP segment to the next
  - Checksum – used by receiving host to check whether errors have been introduced into the segment
    - Also calculated over a few fields in the IP header

Figure 3.7 – UDP Segment Structure



UDP Checksum

- Used to determine whether bits within the UDP segment have been altered (like from noise in the links or while stored in a router)
- UDP at the sender side performs 1s complement of the sum of all the 16-bit words in the segment
  - Wraps around any overflow
- Book Example
- Why bother?
  - No guarantee that all the links between source and destination provide error checking; one of the links may use a link-layer protocol that does not provide error checking
  - Possible that errors are introduced when a segment is stored in a router’s memory

UDP Checksum

- UDP must provide error detection at the transport layer on an end-to-end basis, if the end-end data transfer service is to provide error detection
- Introduction to end-end principle in system design
  - Certain functionality must be implemented on an end-end basis
    - Functions placed at lower levels may be redundant or of little value when compared to the cost of providing them at the higher level
- IP is supposed to run over just about any layer-2 protocol – useful for the transport layer to provide error checking as a safety measure
- UDP provides error checking, but does not do anything to recover from an error
  - Some implementations discard the damaged segment
  - Other implementations pass damaged segment with a warning

Principles of Reliable Data Transfer

- Probably of highest priority; problems could also occur at other layers preventing this
- Responsibility: implement this service abstraction
  - Made difficult because it’s possible the layer below the reliable transfer protocol is implemented on top of an unreliable end-to-end network (IP)
  - We can view this lower layer as unreliable point-to-point channel
- Building a Reliable Data Transfer Protocol
- Pipelined Reliable Data Transfer Protocols
- Go-Back-N (GBN)
- Selective Repeat (SR)

Figure 3.8 – Reliable Data Transfer – Service Model and Service Implementation

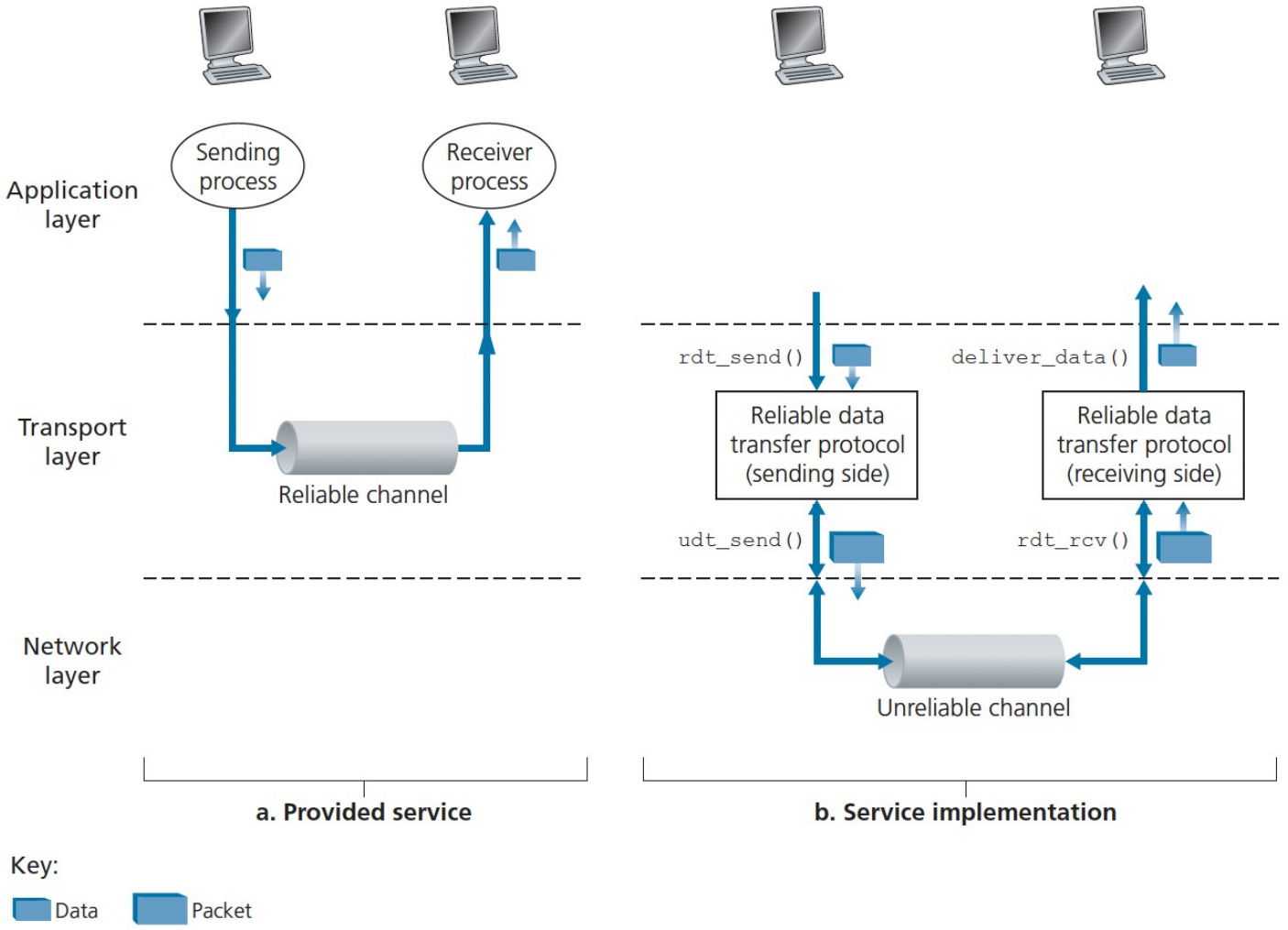


Figure 3.8 – Explained

- Service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred
- Reliable channel – no transferred data bits are corrupted (flipped to its complement) or lost
  - All are delivered in the order in which they were sent
- This is what TCP does (Figure 3.8a)
- Figure 3.8b illustrates data transfer protocol for our data transfer protocol
  - Sending side of the data transfer protocol will be invoked from above by a a call to rdt_send()
  - It will pass the data to be delivered to the upper layer of the receiving side
  - Receiving side – rdt_rcv() will be called when a packet arrives from the receiving side of the channel
  - When it wants to deliver data to the upper layer, it will do so by calling deliver_data()
- For discussions, instead of referring to the transport-layer information as a segment, it will be referred to as a packet
- Focus: unidirectional data transfer
  - Bidirectional is possible, but skipped in your book (more tedious to explain)
  - Unidirectional: data transfer from the sending to the receiving side

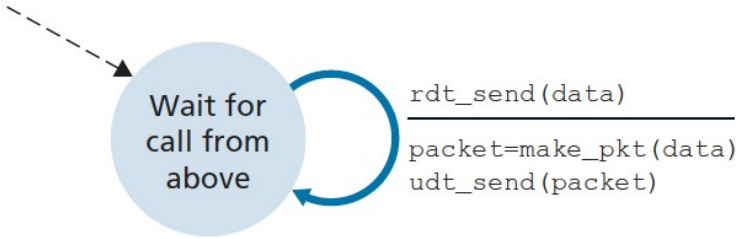
Building a Reliable Data Transfer Protocol

- Reliable Data Transfer over Reliable Channel: rdt1.0
- Reliable Data Transfer over a Channel with Bit Errors: rdt2.0
- Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

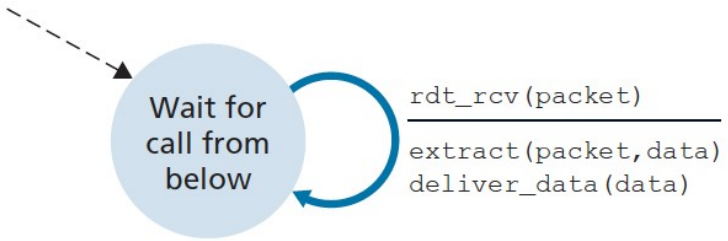
rdt1.0 - Reliable Data Transfer over a Reliable Channel

- Finite state machine (FSM) definitions for rdt1.0 sender and receiver are shown in Figure 3.9
- Important to note that there will be different FSMs for sender and receiver
  - Figure 3.9 – each have just one state
  - Arrows – transition of the protocol from one state to another
    - Figure 3.9 – one state – transition necessary from the one state back to itself
- More notation to know…
  - Event causing transition is shown above the horizontal line labeling the transition
  - Actions taken when event occurs – beneath the line
- One other important note: usually FSMs are more complicated than that from 3.9; have multiple states, so it’s important to determine the initial state of each FSM

Figure 3.9 – rdt1.0 – A Protocol for a Completely Reliable Channel



a. rdt1.0: sending side



b. rdt1.0: receiving side

rtd1.0 - Continued

- Sending side
  - Accepts data from the upper layer via rdt_send(data) event
  - Creates a packet containing the data via make_pkt(data)
  - Sends the packet into the channel
  - In reality – would result from a procedure call by the upper-layer application
- Receiving side
  - Receives a packet from the underlying channel via rdt_rcv(packet) event
  - Removes data from the packet via extract(packet, data)
  - Passes the data up to the upper layer via deliver_data(data)
- No difference between a unit of data and a packet
- All packet flow is from sender to receiver; no need for receiver to send feedback to sender
- Assume receiver is able to receive data as fast as the sender sends

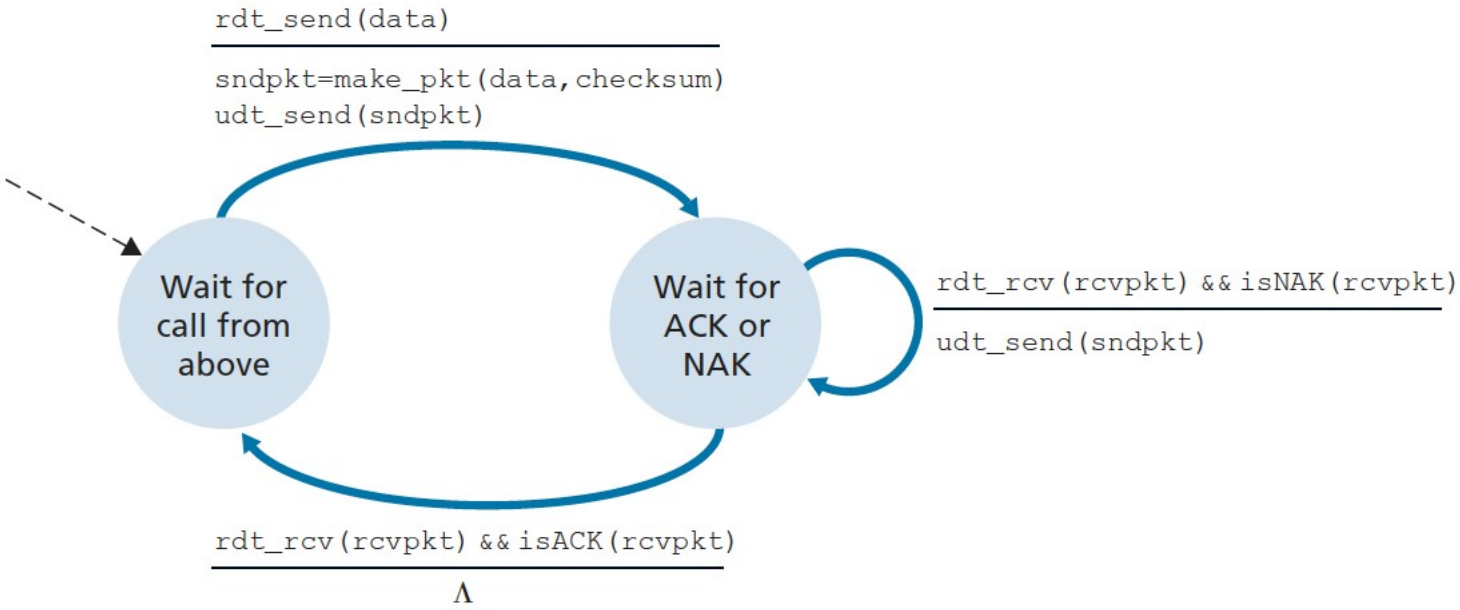
rdt2.0 – Reliable Data Transfer Over a Channel with Bit Errors

- More realistic: bits in a packet may be corrupted; usually occurs in the physical components of a network as packet is traverses to the receiver
- Continue assuming that all transmitted packets are received (bits could be corrupted) in the order in which they were sent
- Example: being on the phone with someone; after sentence, either the receiver will say OK (positive acknowledgments) or “Please repeat that” (negative acknowledgments)
- In networking – ARQ protocols
  - ARQ = Automatic Repeat request
- Three additional protocol capabilities required in ARQ protocols to handle bit errors
  - Error detection
  - Receiver feedback
  - Retransmission

ARQ Protocols

- Error detection
  - Needed to allow the receiver to detect when bit errors occur
    - UDP – uses checksum field
  - For our purposes at the moment – require extra bits, beyond the bits of original data to be transferred) be sent from sender to receiver
  - Bits will be gathered into the packet checksum field of the rdt2.0 data packet
- Receiver feedback
  - Since sender and receiver execute on different end systems, the only way for the sender to learn of the receiver's view of the packet is for the receiver to provide explicit feedback to the sender
  - Positive (ACK) and negative (NAK) acknowledgments are examples of feedback
  - Packets need only be one bit long; 0 – NAK, 1 – ACK (maybe…this is the suggestion)
- Retransmission
  - Packet that is received in error at the receiver will be retransmitted by the sender

Figure 3.10a – rdt2.0 – Protocol for a Channel with Bit Errors – Sending Side

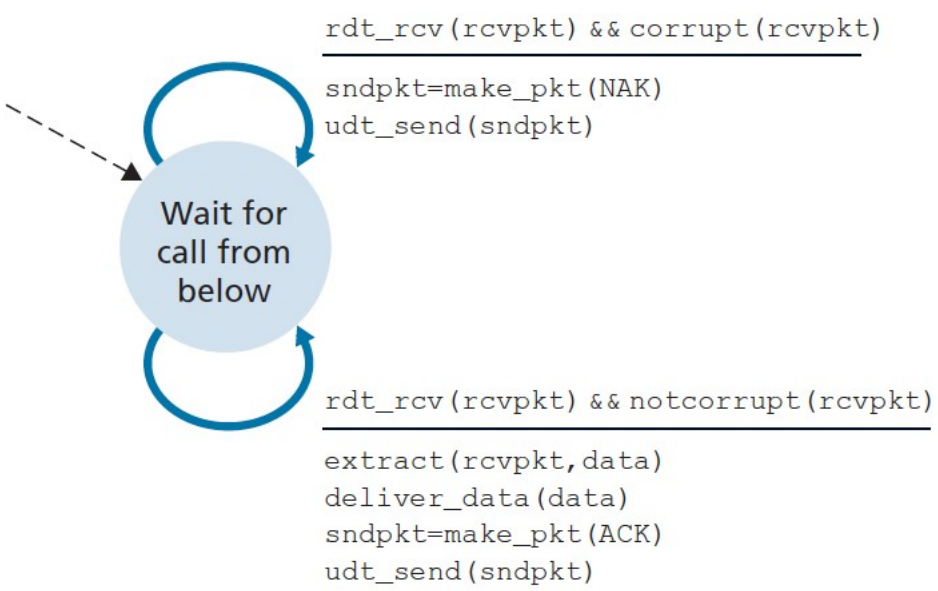


a. rdt2.0: sending side

Figure 3.10a - Sender

- Two states
  - Leftmost state – send-side protocol is waiting for data to be passed down from the upper layer
    - When rdt_send(data) event occurs, sender will create a packet (sndpkt) containing the data to be sent, along with a packet checksum
    - Will then send the packet via the udt_send(sndpkt) operation
  - Rightmost state – sender is waiting for an ACK or NAK packet from the receiver
    - If ACK – sender knows that the most recently transmitted packet has been received correctly; returns to normal waiting state
    - If NAK – protocol retransmits the last packet and waits for ACK or NAK
    - While in this state, the sender cannot get more data from the upper layer; only occurs once the sender returns to wait for call state
  - Known as stop-and-wait protocols

Figure 3.10b – rdt2.0 – Protocol for a Channel with Bit Errors – Receiving Side



b. rdt2.0: receiving side

Figure 3.10b - Receiver

- Single state
  - On packet arrival, receiver replies with either ACK or NAK
  - Notation rdt_rcv(rcvpkt) && corrupt(rcvpkt) – when the packet is found to be in error
- Beyond…
  - Haven’t accounted for the possibility that the ACK/NAK packet could be corrupted
  - If corrupted – sender has no way of knowing whether or not the receiver correctly received the last piece of transmitted data

Possibilities for Handling Corrupted ACK/NAK

- Equivalent to speaker's "what did you say?" from earlier example is corrupted?
  - Receiver – having no idea whether garbled sentence was part of the dictation or request to repeat the last reply would probably respond with "what did you say?"
  - Response might be garbled
- Add enough checksum bits to allow the sender not only to detect but also recover from bit errors
  - Solves the immediate problem for a channel that can corrupt packets but not lose them
- Third route – sender simply resends the current data packet when it receives a garbled ACK/NAK
  - Introduces duplicate packets into the sender-to-receiver channel
  - Problem: receiver doesn't know whether the ACK/NAK it last sent was received correctly at the sender

Corrupted Data Solution – Sequence Number?

- Add a new field to the data packet and have the sender number its data packets by putting a sequence number into this field
- Receiver checks this number to determine whether/not received packet is a retransmission
  - For stop-and-wait – 1-bit sequence number will work
    - New packet or retransmitted packet?
    - Go back and forth between 0 and 1 for packet numbers, since sender can only send one packet at a time
  - ACK/NAK packets do not need to indicate the sequence number of the packet they are acknowledging
  - Sender knows that a received ACK/NAK was generated in response to its most recently transmitted packet
- Figures 3.11 and 3.12 – fixed version of rdt2.0 as rdt2.1

Considering rdt2.1

- Sender and receiver have twice as many states as before
  - Protocol state must now reflect whether the packet currently being sent (by sender) or expected (at the receiver) should have a sequence number of 0 or 1
  - Actions in those states where a 0-numbered packet is being sent/expected are mirrored to those where a 1-numbered packet is sent or expected
    - Only difference – sequence number
- Uses both positive and negative acknowledgments from receiver to sender
  - When an out-of-order packet is received, the receiver sends a positive acknowledgment for the packet it has received
  - When a corrupted packet is received, the receiver sends a negative acknowledgment
  - Similar for NAK
    - Sender that receives two ACKs for the same packet knows that the receiver did not correctly receive the packet since the packet was ACKed twice

Figure 3.11 – rdt2.1 Sender

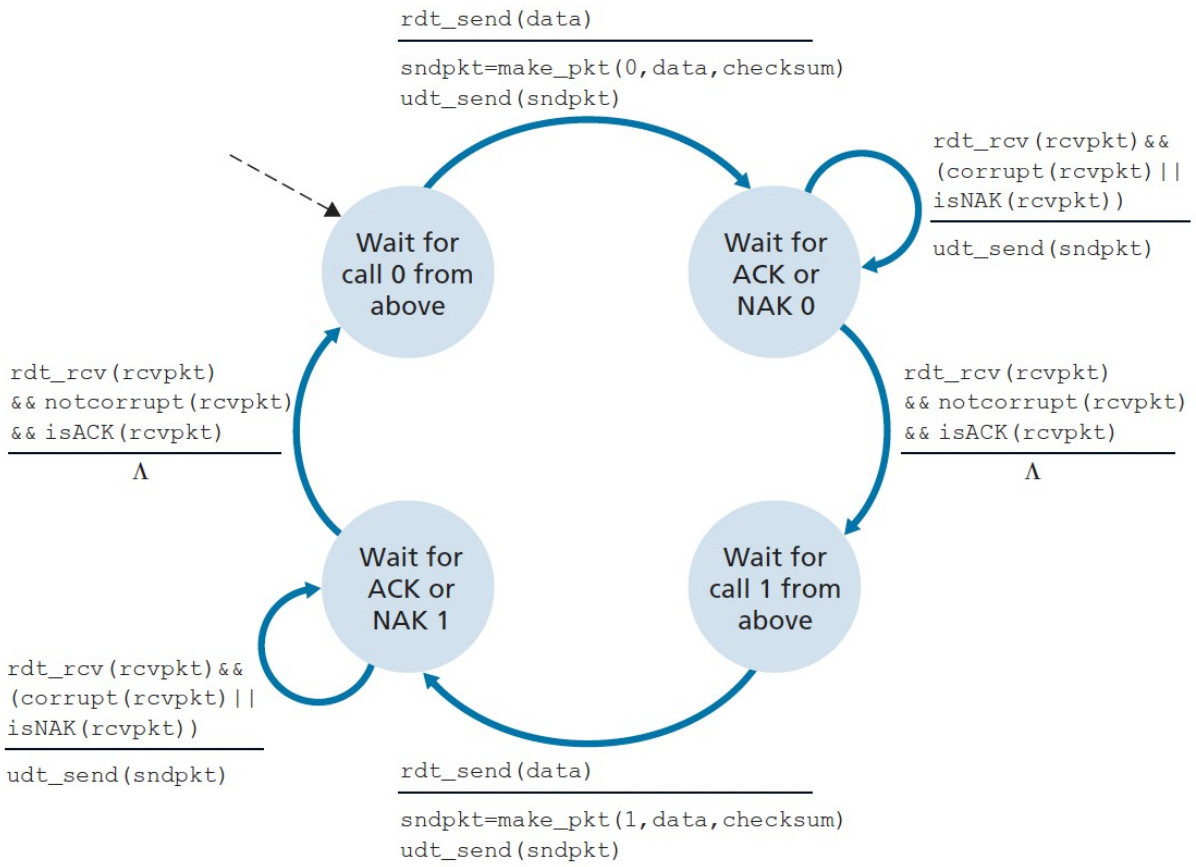
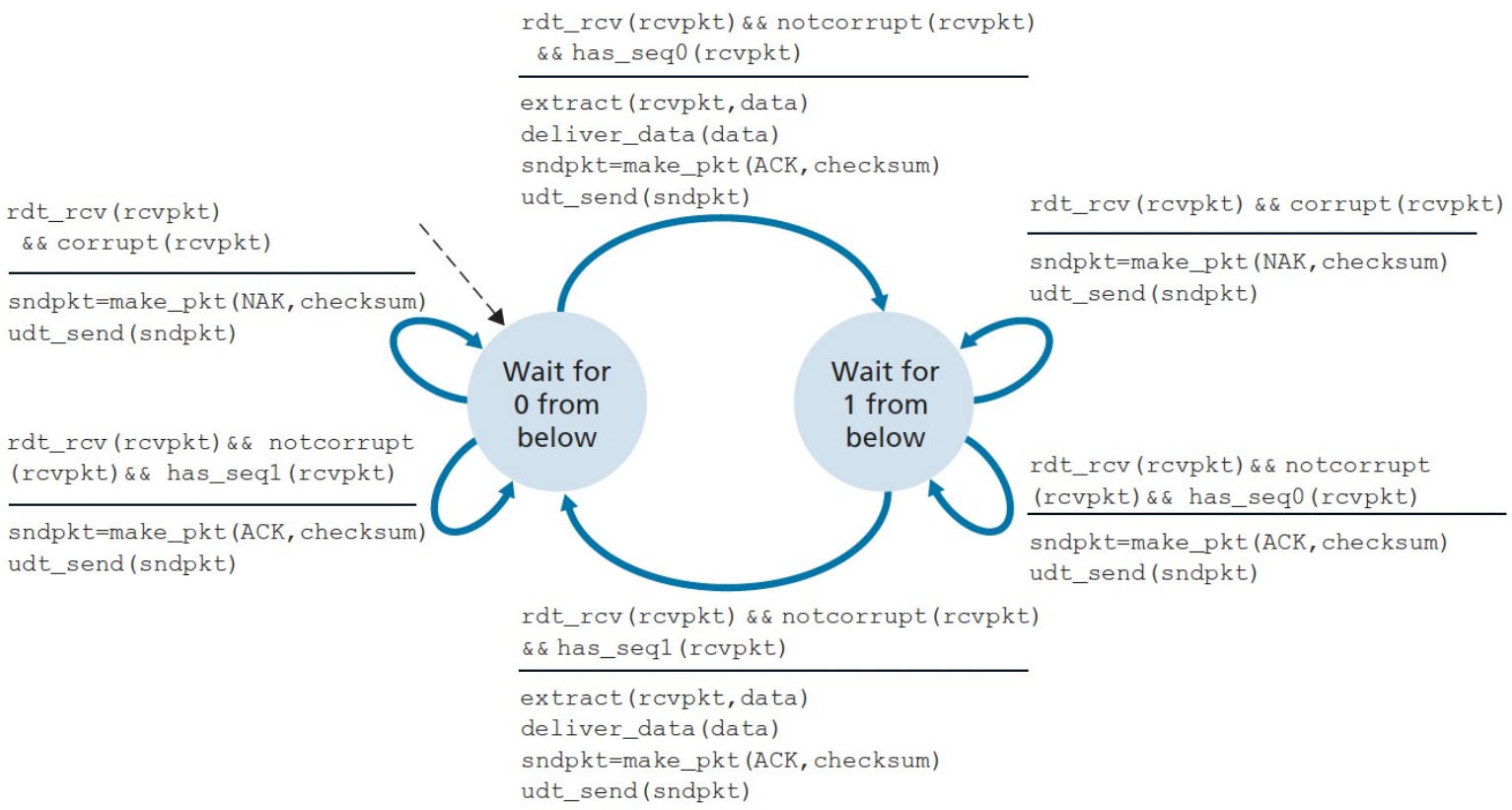


Figure 3.12 – rdt2.1 Receiver



Slight modifications for rdt2.2

- Nak-free reliable data transfer protocol for a channel with bit errors
- Figure 3.13 and 3.14
- Receiver must now include the sequence number of the packet being acknowledged by an ACK message
- Sender must now check the sequence number of the packet being acknowledged by a received ACK message
  - Include 0 or 1 argument in isACK() in the sender FSM

Figure 3.13 – rdt2.2 Sender

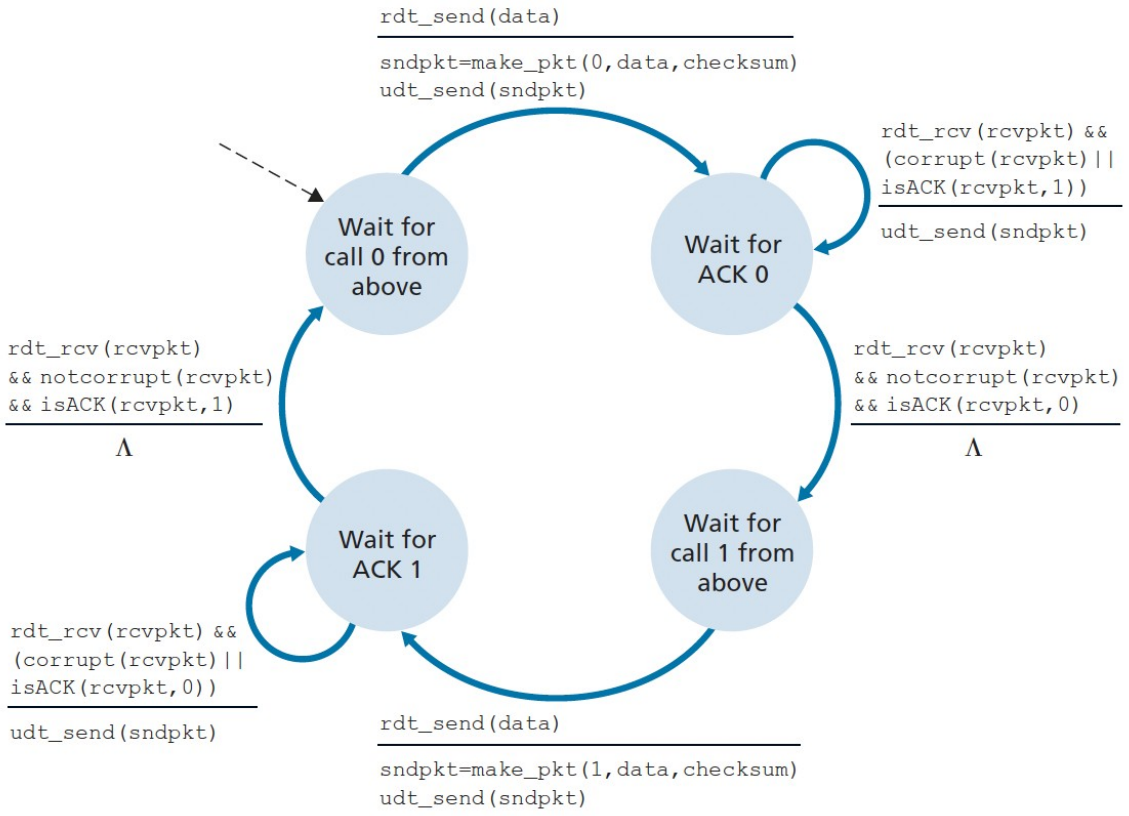
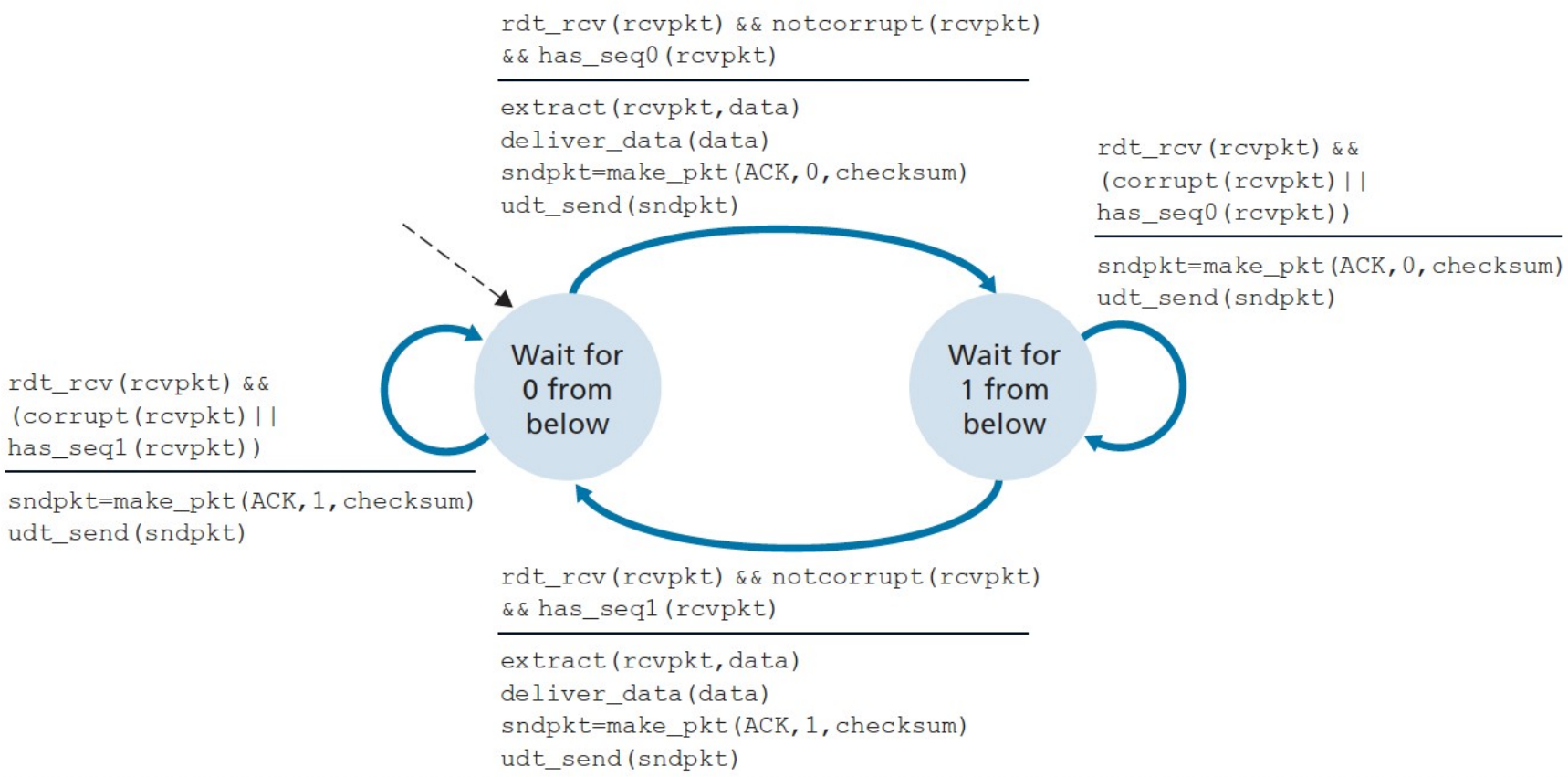


Figure 3.14 – rdt2.2 Receiver



rdt3.0 – Reliable Data Transfer Over Lossy Channel with Bit Errors

- Previous version: corrupting bits
- Underlying channel can lose packets as well as getting corrupted packets
- Additional concerns
  - How to detect packet loss
  - What to do when packet loss occurs
- Handling in rdt2.2 will help answer packet loss occurring
- First concern – new protocol mechanism

A Closer Look – rdt3.0

- Possible approaches for dealing with packet loss
  - Put the burden of detecting and recovering from lost packets on the sender
  - Suppose that the sender transmits a data packet and either that packet or receiver's ACK of that packet gets lost
  - No reply is going back to the sender from receiver
  - If sender is willing to wait long enough so it is certain that a packet has been lost, it can retransmit the packet
- How long must the sender wait before assuming something is lost?
  - At least as long as a round-trip delay between the sender and receiver plus whatever amount of time is needed to process a packet at the receiver
  - Networks – worst-case max delay is difficult to estimate, or even know its correctness
  - Protocol should ideally recover from packet loss as soon as possible
    - Waiting for a worst-case delay could mean a long wait until error recovery is initiated
  - Sender "judiciously" chooses a time value such that packet loss is likely, although not guaranteed
    - If an ACK is not received – packet is retransmitted
    - Extra fun: packet experiences large delay – sender may retransmit the packet even though neither the packet or the ACK have yet been lost; so, duplicate packets again

Sender for rdt3.0

- Does not know whether a data packet was lost, ACK was lost, or if there is some kind of big delay
- The action remains the same no matter the situation – retransmit
- Implementing a time-based retransmission mechanism requires a countdown timer that can interrupt the sender after a given amount of time has expired
- Will need to be able to
  - Start the timer each time a packet is sent
  - Respond to a timer interrupt
  - Stop the timer
- Figure 3.15 shows sender FSM for rdt3.0 protocol that transfers data over a channel that can corrupt or lose packets
- Figure 3.16 – time moves forward from the top of the diagram toward the bottom of the diagram
  - Receive time for a packet is later than the send time for a packet as a result of transmission and propagation delays
  - (b)-(d) – send-side brackets indicate times at which a timer is set and later times out
- Because packet sequence numbers alternate between 0 and 1 – rdt3.0 is known as the alternating-bit protocol

Figure 3.15 – rdt3.0 Sender

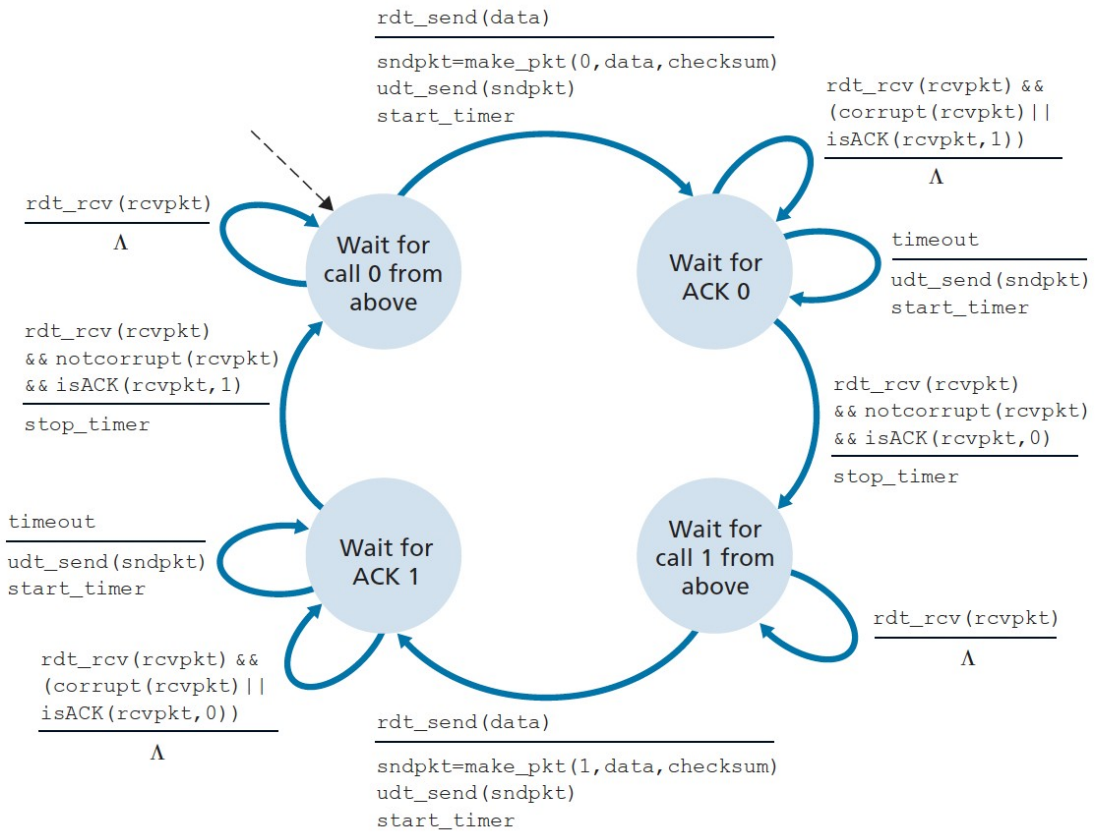
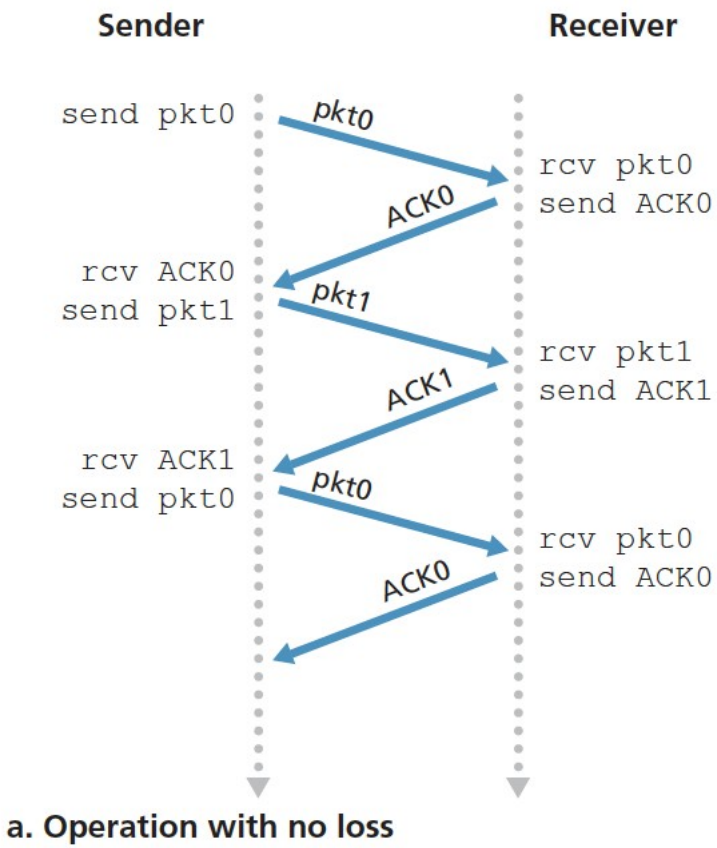


Figure 3.16a – Operation of rdt3.0, The Alternating-Bit Protocol – Operation with No Loss



a. Operation with no loss

Figure 3.16b – Operation of rdt3.0, The Alternating-Bit Protocol – Lost Packet

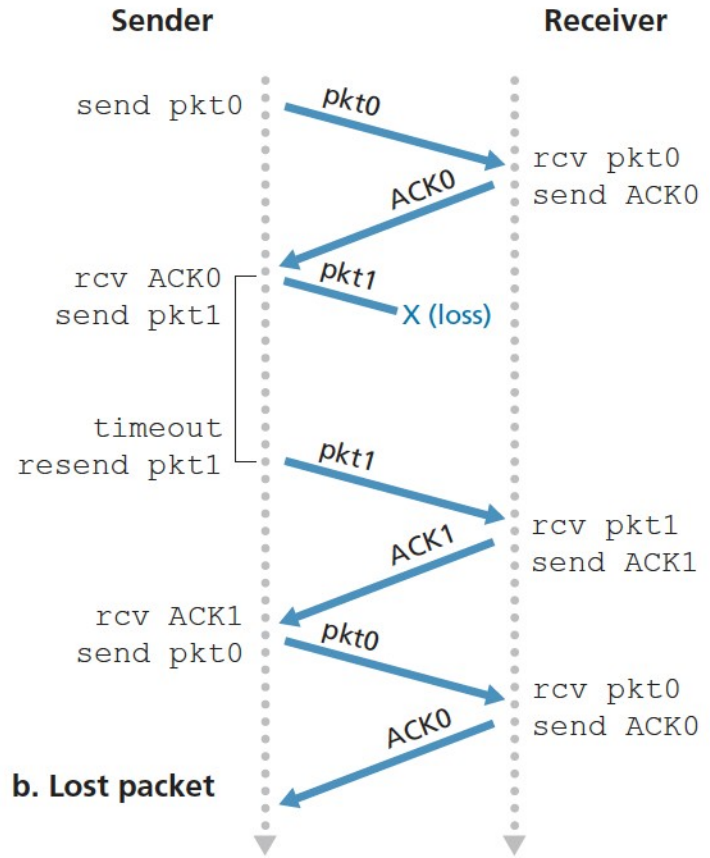
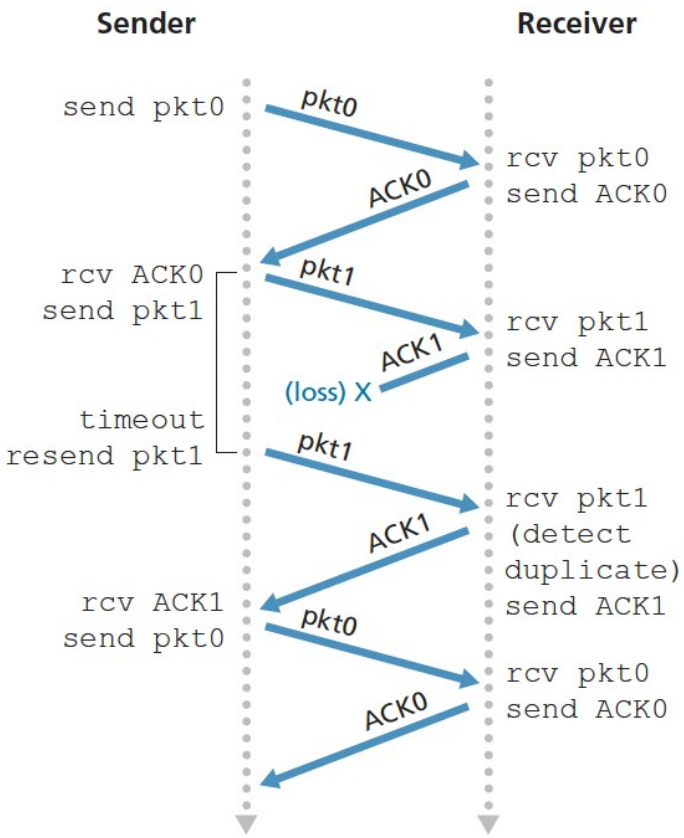
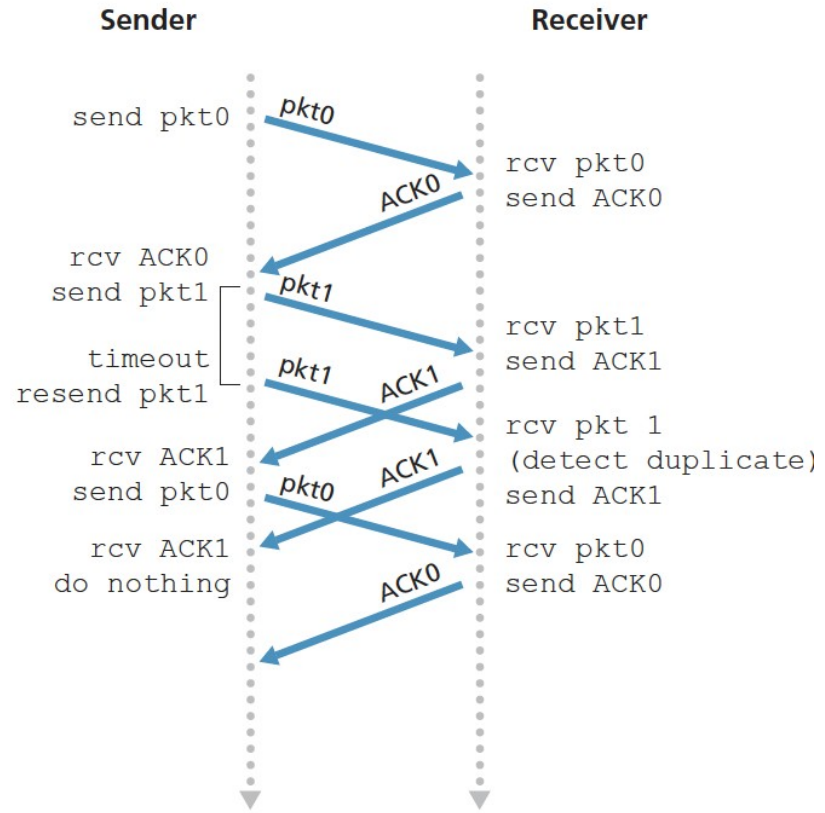


Figure 3.16c – Operation of rdt3.0, The Alternating-Bit Protocol – Lost ACK



c. Lost ACK

Figure 3.16d – Operation of rdt3.0, The Alternating-Bit Protocol – Premature Timeout

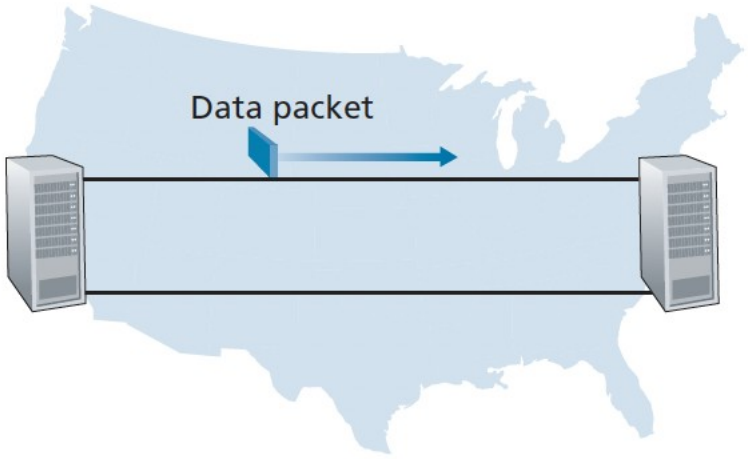


d. Premature timeout

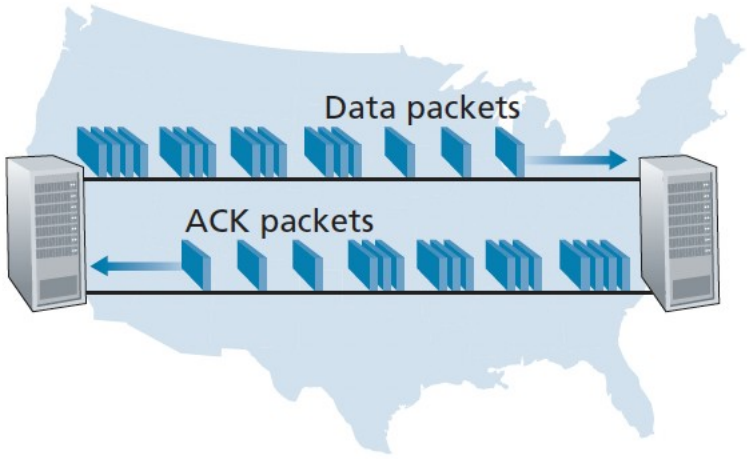
Pipelined Reliable Data Transfer Protocols

- The proposed rdt3.0 is functionally correct, but unlikely anyone would be happy with performance
  - Mainly – being a stop-and-wait protocol
- Figure 3.17 – two hosts – one on East Coast; other on West Coast
- Speed of light round-trip propagation delay between these two end systems, RTT, is roughly 30ms
- Suppose that they are connected by a channel with a transmission rate R of 1 Gbps
- With packet size of L, 1000 bytes (8000 bits) per packet, including both header fields and data – time needed to actually transmit the packet into the link is:
  - L/R = 8000 bits/1 billion bits per second = 8 microseconds

Figure 3.17 – Stop-and-Wait v. Pipelined Protocol



a. A stop-and-wait protocol in operation



b. A pipelined protocol in operation

Pipelined

- Figure 3.18a shows that with the stop-and-wait protocol, if the sender begins sending the packet at t = 0, then at t = L/R = 8 microseconds, the last bit enters the channel at the sender side
- Packet makes its 15ms cross-country trip
- Last bit of the packet emerges at the receiver at t = RTT/2 + L/R = 15.008 ms.
- Assuming ACK packets are extremely small (ignore transmission time) and receiver can send an ACK as soon as the last bit of a packet arrives, the ACK arrives back at the sender at t = RTT + L/R = 30.008ms
- Sender can then transmit next message
- In that 30.008ms, the sender was sending for only 0.008ms
- Utilization of the sender – fraction of time the sender is actually busy sending bits into the channel, Figure 3.18(a) shows stop-and-wait protocol has a sender utilization of 0.00027

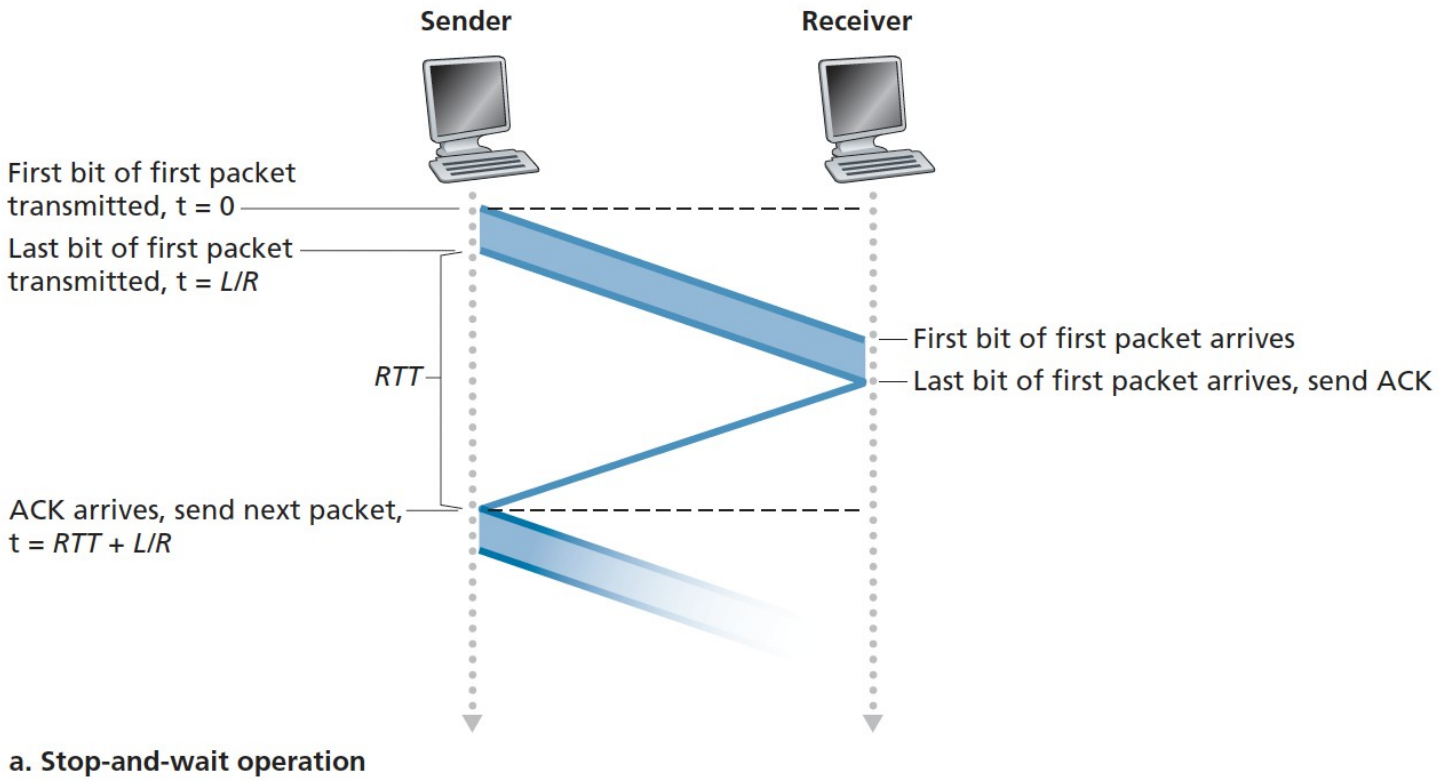
Pipelined

- Sender was busy only 2.7 hundredths of one percent of the time
  - Not good
- Alternative
  - Instead of stop-and-wait – sender is allowed to send multiple packets without waiting for acknowledgments
    - See Figure 3.17b
  - Figure 3.18b shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, utilization of the sender is tripled
  - Since many in-transit sender-to-receiver packets can be visualized as filling a pipeline, hence the name

Pipeline - Consequences

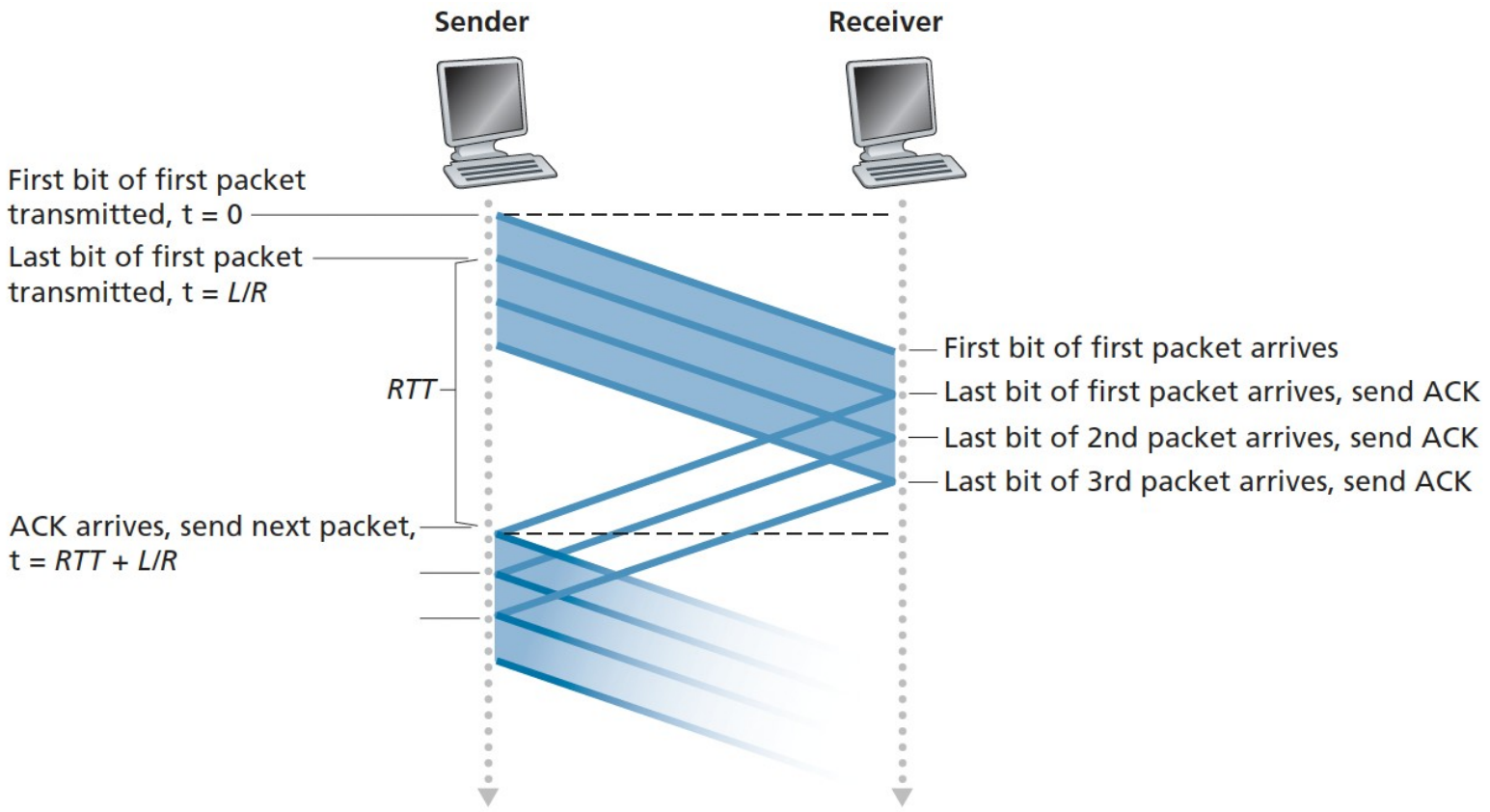
- The range of sequence numbers must be increased, since each in-transit packet must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets
  - Not counting retransmissions
- Sender and receiver sides of the protocols may have to buffer more than one packet.
  - The sender will have to buffer packets that have been transmitted but not yet acknowledged
  - Buffering of correctly-received packets may also be needed at the receiver
- Range of sequence numbers needed and buffering requirements will depend on the manner in which a data transfer protocol responds to lost/corrupted/overly-delayed packets
- Two approaches
  - Go-Back-N (GBN)
  - Selective Repeat

Figure 3.18a – Stop-and-Wait Sending



a. Stop-and-wait operation

Figure 3.18b – Pipelined Sending



b. Pipelined operation