8. Principles of Reliable Data Transport

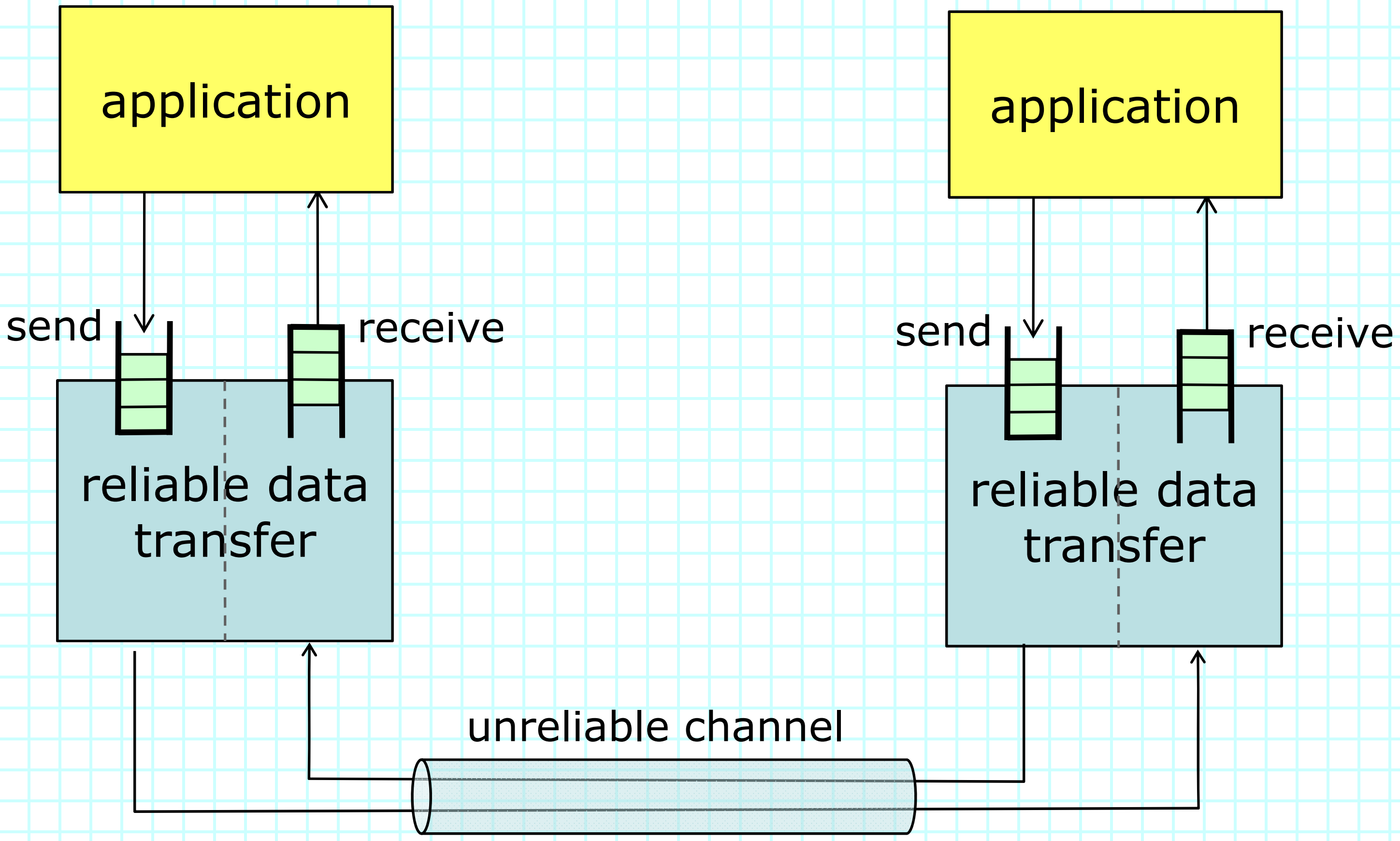
- Basic concepts
- Stop-and-wait protocol
- Go-Back-*N* protocol
- Selective Repeat protocol

Adapter from Jon Turner and Kurose and Ross

Preliminaries

- Physical channels are never completely reliable
  - » wireless links subject to interference
  - » transmission noise can introduce bit errors
  - » routers & switches may drop packets due to buffer overflow
- Reliable communication relies on (1) detection (of lost or corrupted packets) and (2) retransmissions as necessary
  - » sender retains copy until it knows packet has been received
  - » receiver notifies sender and delivers (to application) each piece of data once, and only once, preferably in order
- Protocol design specifies coordination between sender and receiver, **and** depends on nature of "channel"
  - » may corrupt packets, but never lose them
  - » may corrupt and/or lose packets but never re-order them
  - » may corrupt, lose or re-order packets

Model for Reliable Communication



- Reliable data transfer layer provides send/receive methods used by applications to transfer packets
- Reliable data transfer layer uses fields in packet header to coordinate with peer – this is transparent to application

Basic Concepts

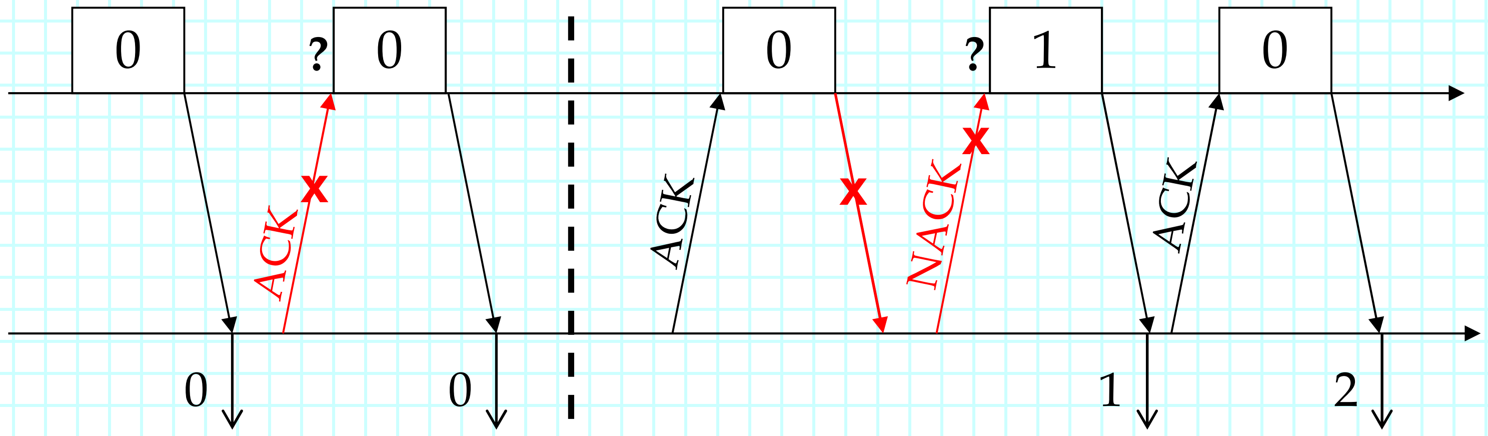
- focus separately on *sender* and *receiver* behavior
- receiver *acknowledges* packets from sender
  - » positive acks and/or negative acks (nack)
- sender *transmits* and *retransmits* based on information provided by the received
  - » time-out to trigger action if ack/nack is not received within a certain time-frame
  - » both packets and acks can get lost
- sequence numbers in each packet
  - » sequence number in ack identifies received packet(s)
  - » size of seq. num. field determines # packets that can be sent before acknowledgments must be received
    - • simplest protocol uses 1 bit sequence number
      - – sends one packet and waits for ack before sending another

Basic Protocol – RDT 2.x

- Assume channel may corrupt but never lose packets
  - » receiver can always detect if packet has been corrupted
- Simple protocol for such a channel uses acks and nacks
  - » sender transmits one packet at a time and waits for ack/nack or erroneous packet
  - » receiver sends ack when it receives packet correctly, sends nack when it receives a packet with an error
- Is this enough?
  - » packets, acks and nacks can all get corrupted

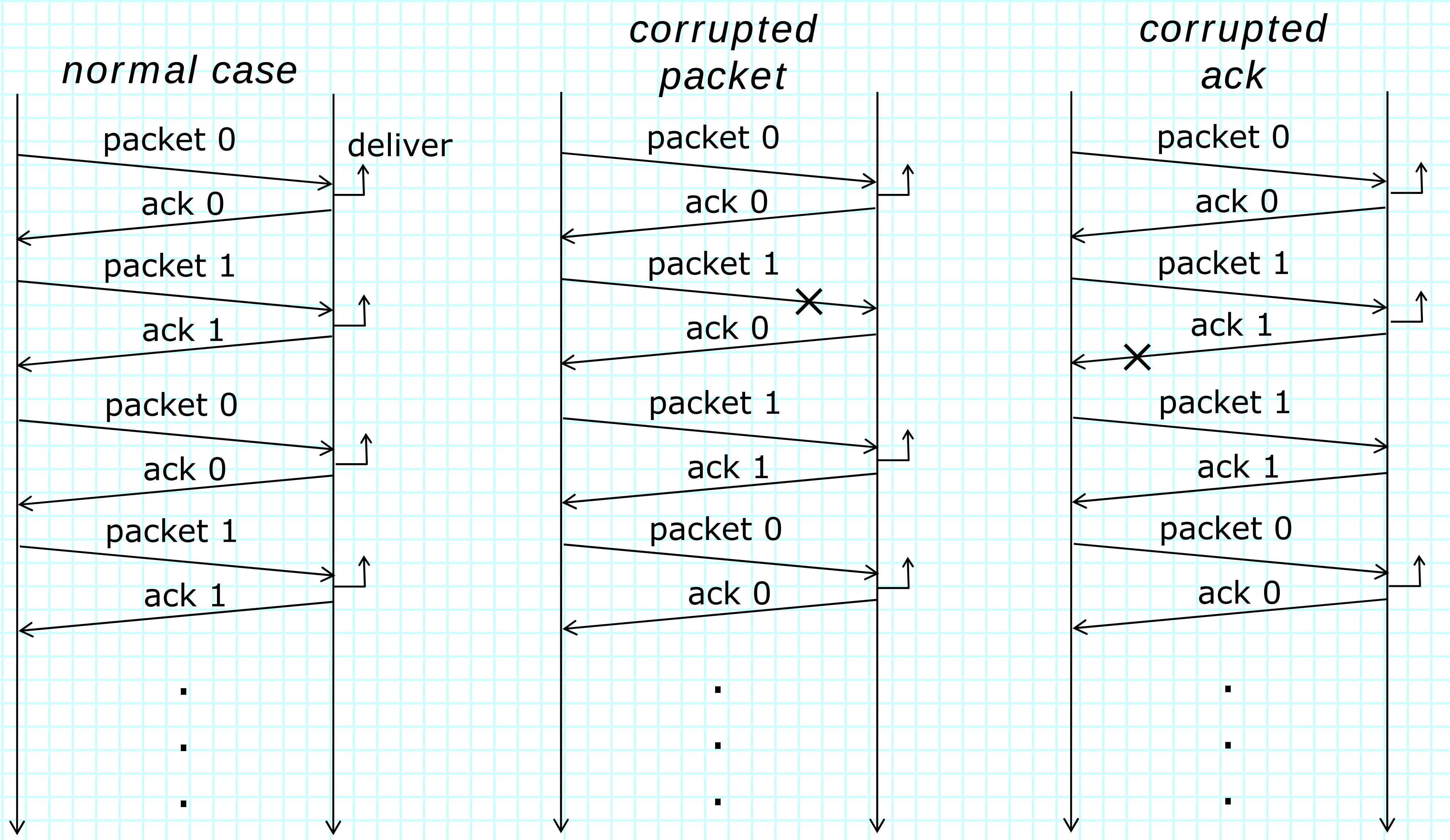
Basic Protocol – RDT 2.x

- many things can go wrong?
  - » Corruption of acks, nacks, or packets means that sender may not be able to tell which packet was received correctly or not
    - if sender retransmits the packet, the receiver could deliver the same packet to the application twice. If it transmits a new one, the receiver can fail to deliver a packet to the application.



- solution is to add a 1 bit sequence number
  - » in this case, it is sufficient to use positive acks only

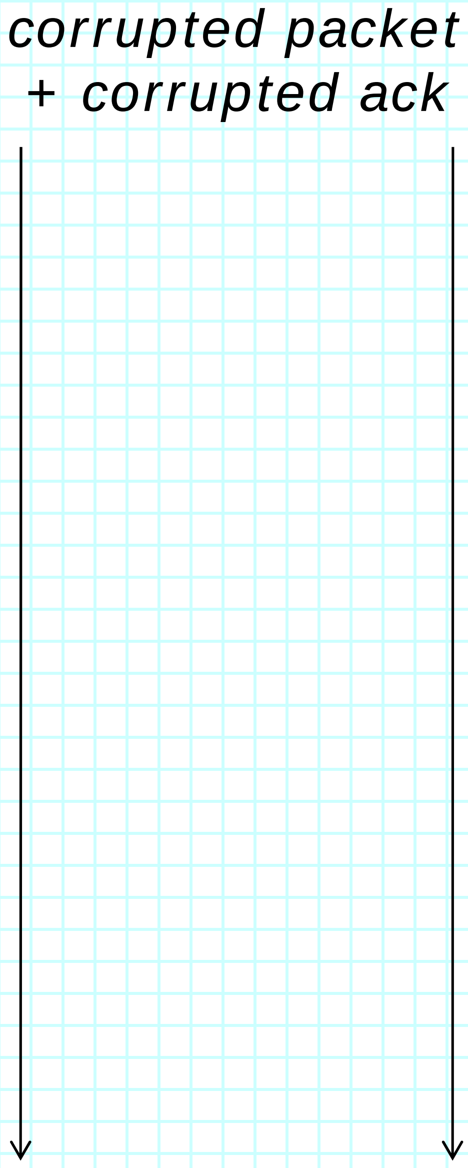
RDT 2.2 Behavior



Exercise

- Consider this scenario for RDT 2.2. Two packets are sent.
  - » the first packet and its ack are not corrupted
  - » the second packet is corrupted, but when it is sent a second time, the packet is not corrupted, but the ack is

  Draw a time-space diagram (like those on the previous slide) describing this scenario, showing how RDT 2.2 recovers and delivers packets to the receiving application.

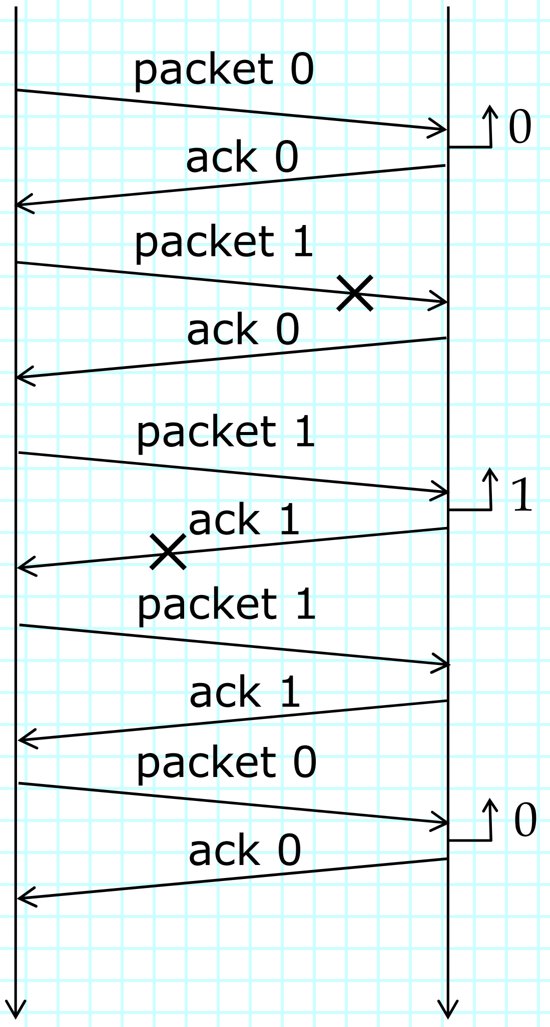


Exercise

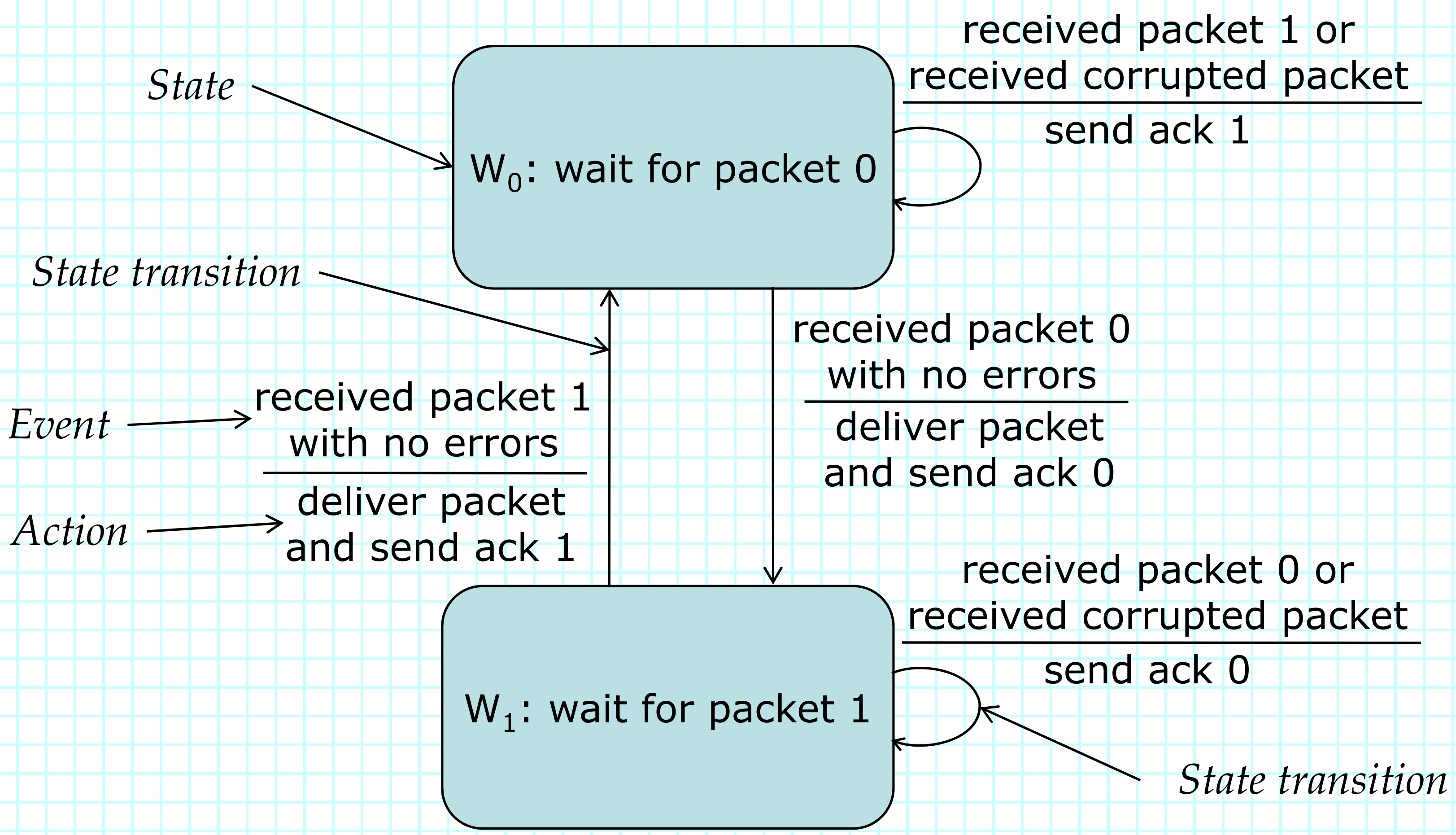
- Consider this scenario for RDT 2.2. Two packets are sent.
  - » the first packet and its ack are not corrupted
  - » the second packet is corrupted, but when it is sent a second time, the packet is not corrupted, but the ack is

Draw a time-space diagram (like those on the previous slide) describing this scenario, showing how RDT 2.2 recovers and delivers packets to the receiving application.

corrupted packet
+ corrupted ack

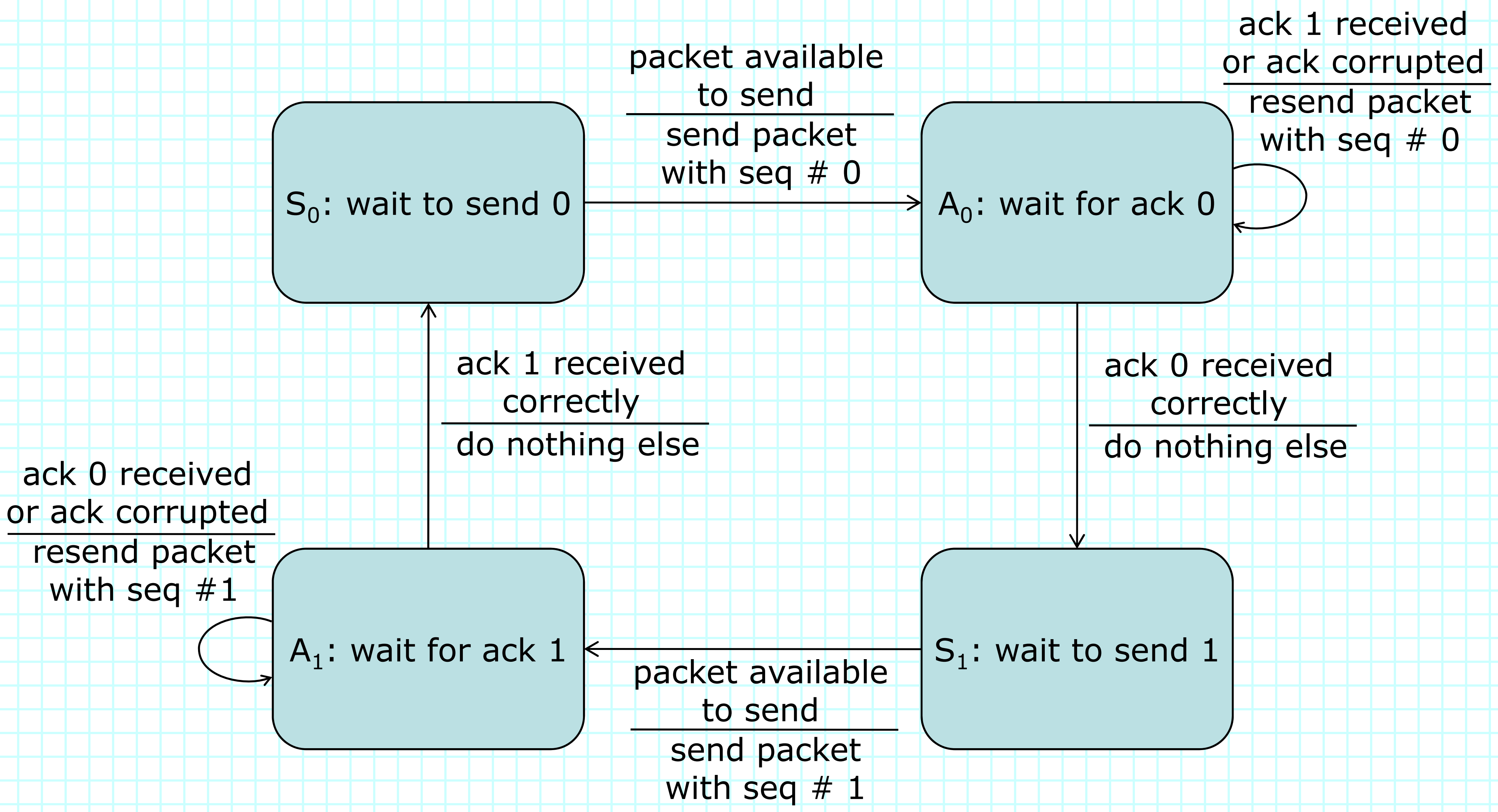


RDT 2.2 Receiver Specification


State
W$_0$: wait for packet 0
received packet 1 or received corrupted packet
send ack 1
State transition
received packet 0 with no errors
deliver packet and send ack 0
Event
received packet 1 with no errors
Action
deliver packet and send ack 1
W$_1$: wait for packet 1
received packet 0 or received corrupted packet
send ack 0
State transition

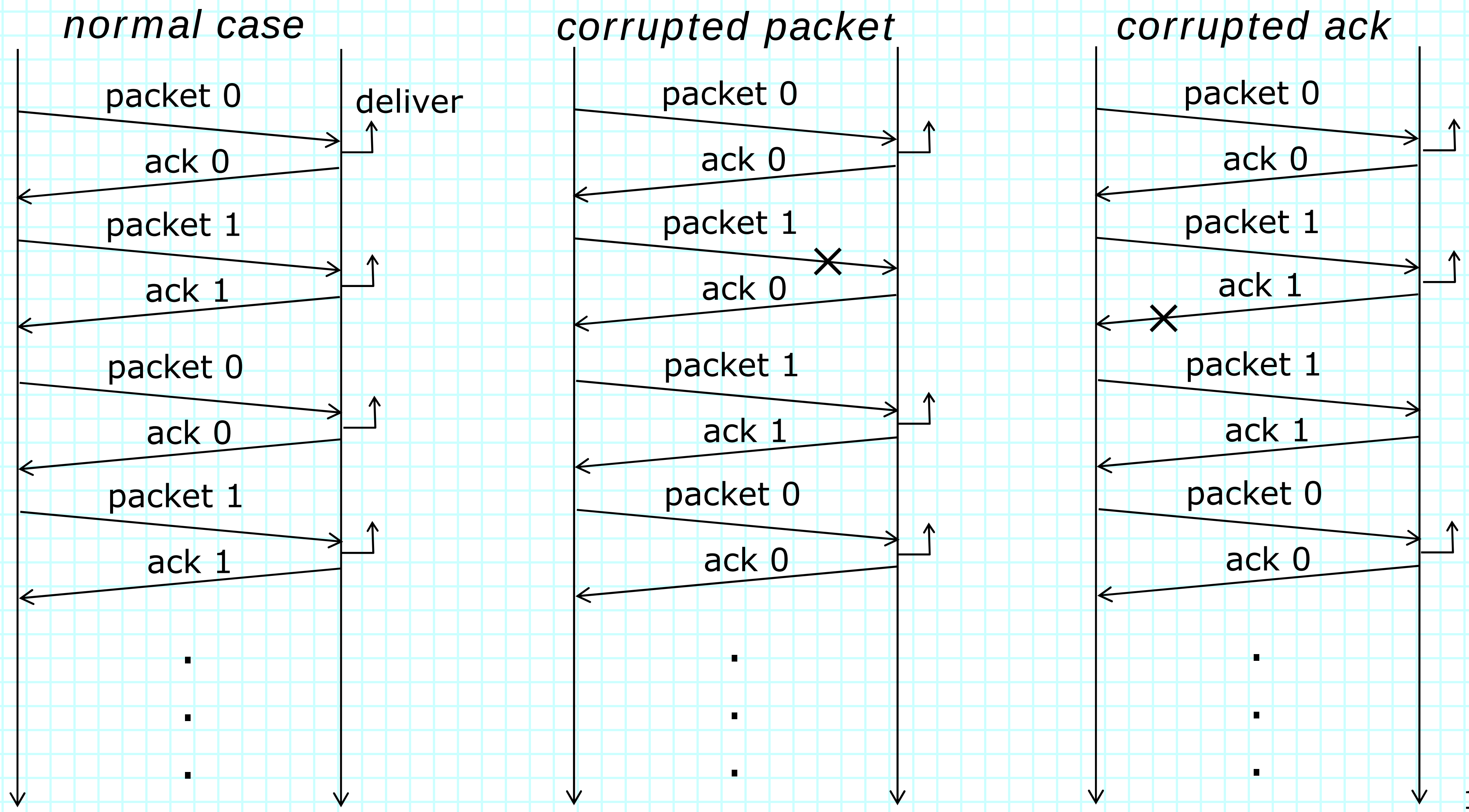
- State machine specifies behavior
  - » States, events, and state transition+action
  - » Used to guide implementation and to analyze behavior

RDT 2.2 Sender Specification


S_0: wait to send 0
A_0: wait for ack 0
S_1: wait to send 1
A_1: wait for ack 1
packet available to send / send packet with seq # 0
ack 1 received or ack corrupted / resend packet with seq # 0
ack 0 received correctly / do nothing else
packet available to send / send packet with seq # 1
ack 0 received or ack corrupted / resend packet with seq #1
ack 1 received correctly / do nothing else

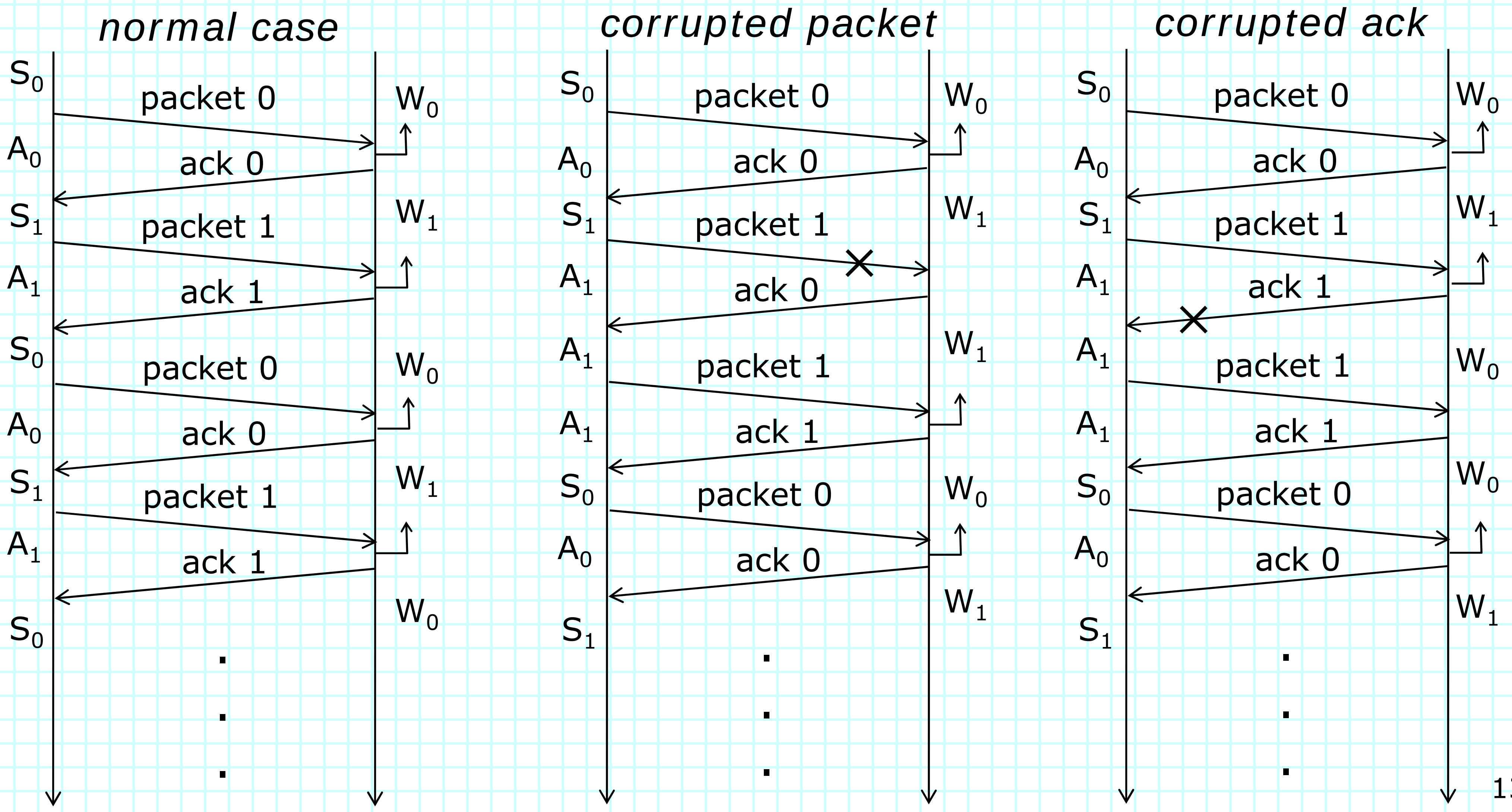
Exercise

1. For each of the time-space diagrams on slide 8, show the sender state and the receiver state at each point in time (just add state labels to the diagram next to the vertical lines).



Exercise

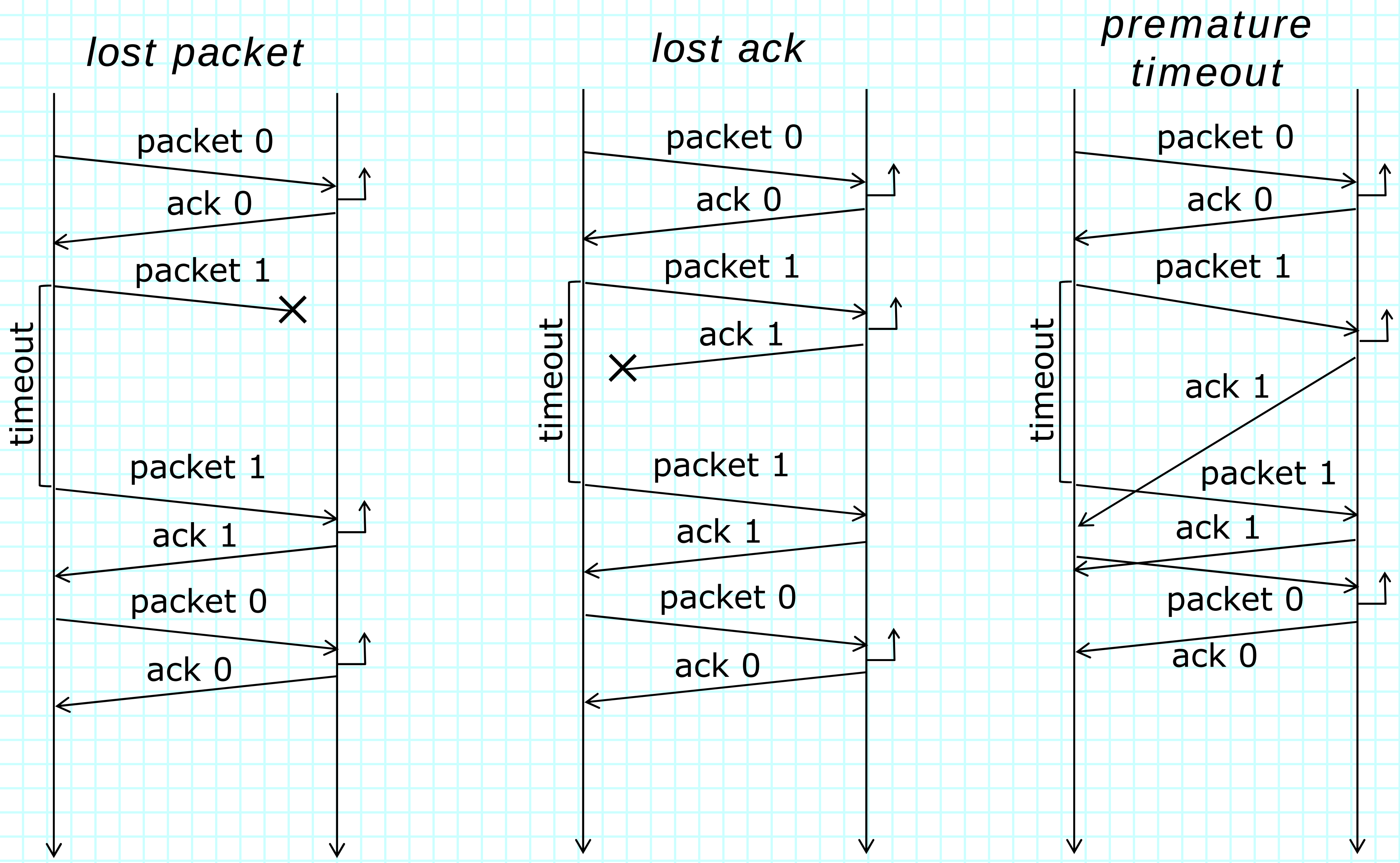
1. For each of the time-space diagrams on slide 8, show the sender state and the receiver state at each point in time (just add state labels to the diagram next to the vertical lines).



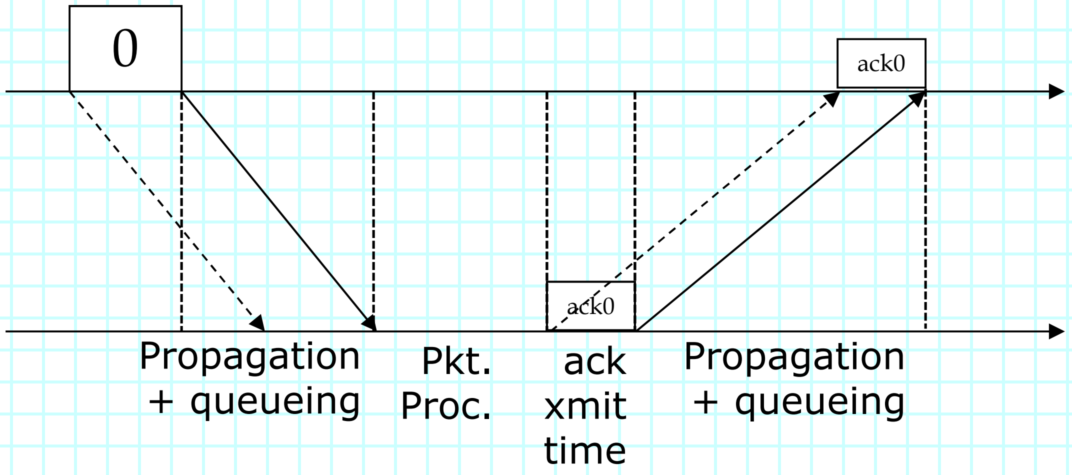
A More Realistic Protocol

- RDT 2.x assumed the channel can corrupt packets but never loses them entirely
  - » means receiver always gets something whenever a packet is sent and sender always receives an “ack” after sending a packet
- In practice, packets can also be lost “without a trace”
  - » receiver never acks lost packet, since does not know it was sent
  - » sender left waiting for ack, but no ack ever comes
- Recovering from lost packets
  - » option 1: keep sending packet repeatedly until ack received
    - • means that in normal case, sender/receiver both do extra work
  - » option 2: sender waits for ack for a limited time period, then re-sends the packet if ack fails to arrive in time
    - • works well if maximum ack delay is known and does not change

RDT 3.0 Behavior

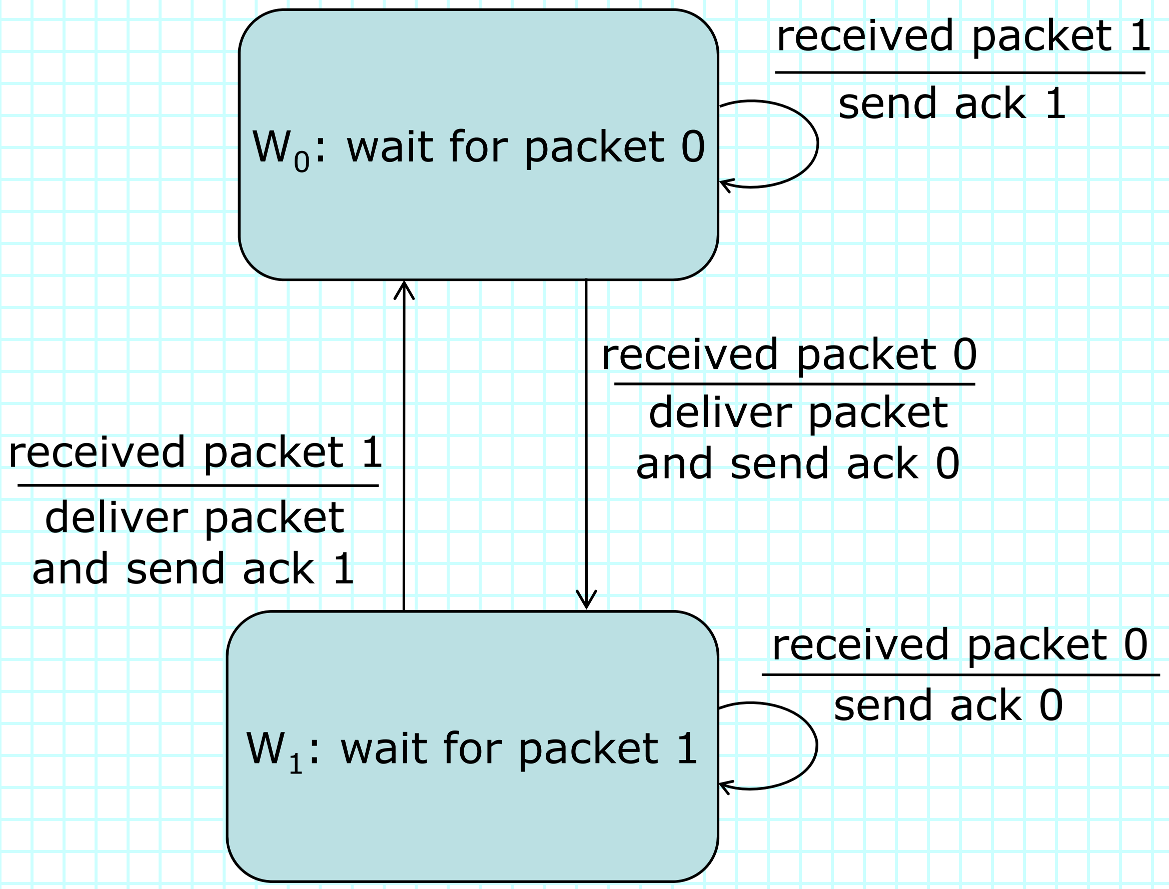


How Long Should *timeout* Be?

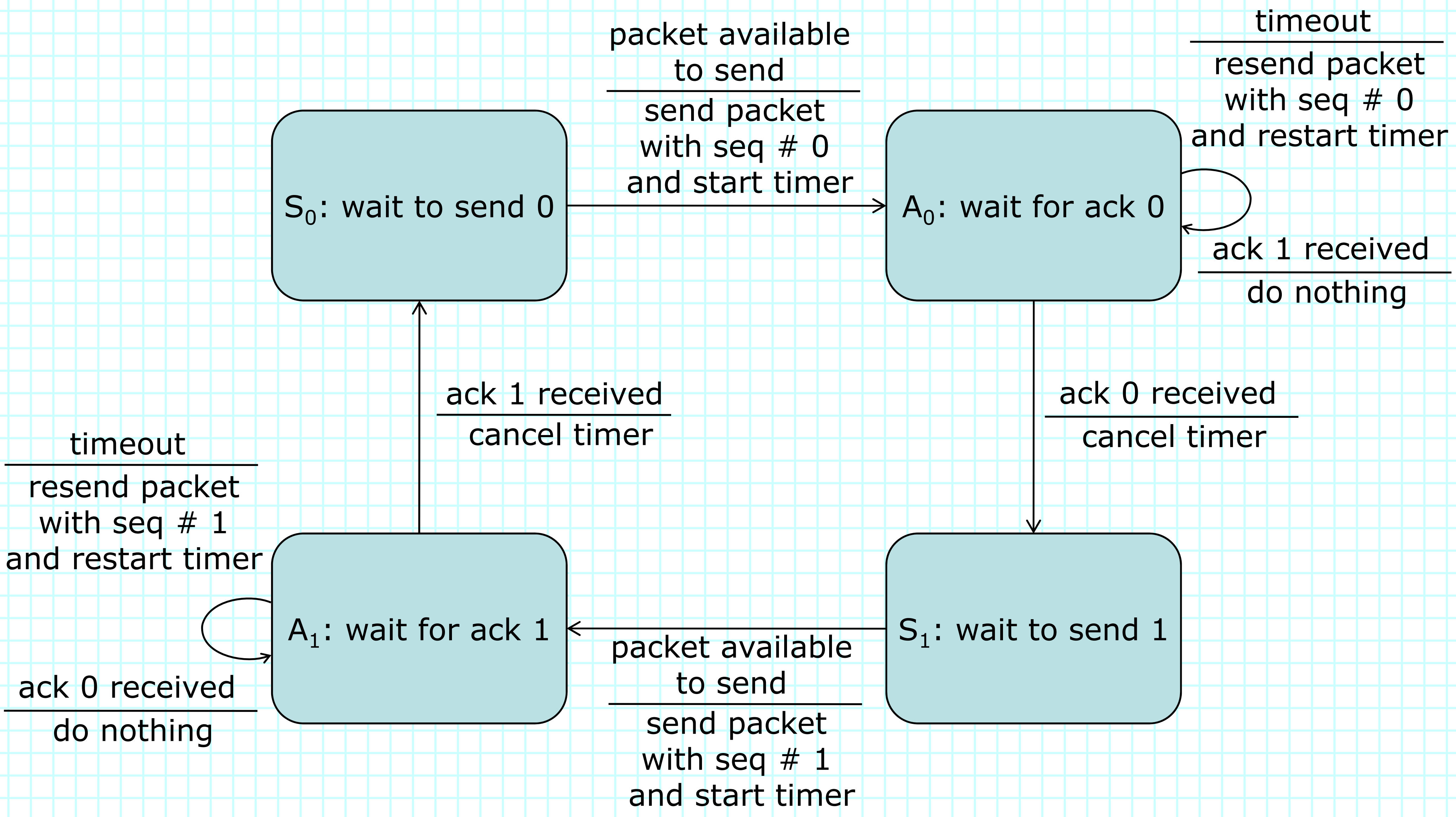


- *timeout* should be at least equal to *roundtrip time* (RTT), *i.e.*, transmission+propagation+queueing of packet & ack + processing time at receiver
  - » *timeout* ≥ RTT
- The challenge lies in (accurately) estimating RTT, when its components are unknown and variable (more on this later, when we discuss TCP)

RDT 3.0 Receiver Specification

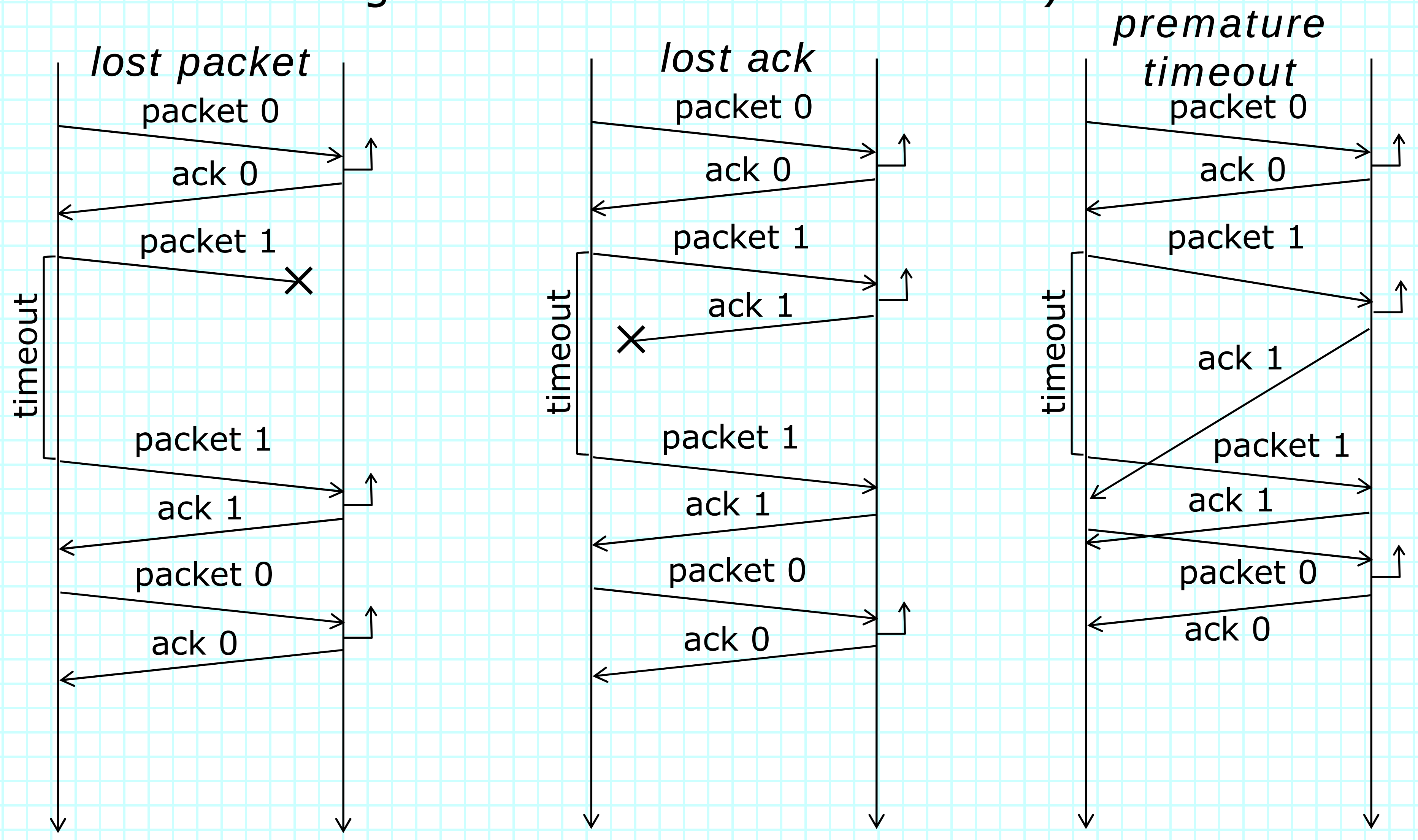

W_0: wait for packet 0
received packet 1 / send ack 1
received packet 0 / deliver packet and send ack 0
received packet 1 / deliver packet and send ack 1
W_1: wait for packet 1
received packet 0 / send ack 0

RDT 3.0 Sender Specification


S$_0$: wait to send 0
A$_0$: wait for ack 0
S$_1$: wait to send 1
A$_1$: wait for ack 1
packet available to send / send packet with seq # 0 and start timer
timeout / resend packet with seq # 0 and restart timer
ack 1 received / do nothing
ack 0 received / cancel timer
packet available to send / send packet with seq # 1 and start timer
timeout / resend packet with seq # 1 and restart timer
ack 0 received / do nothing
ack 1 received / cancel timer

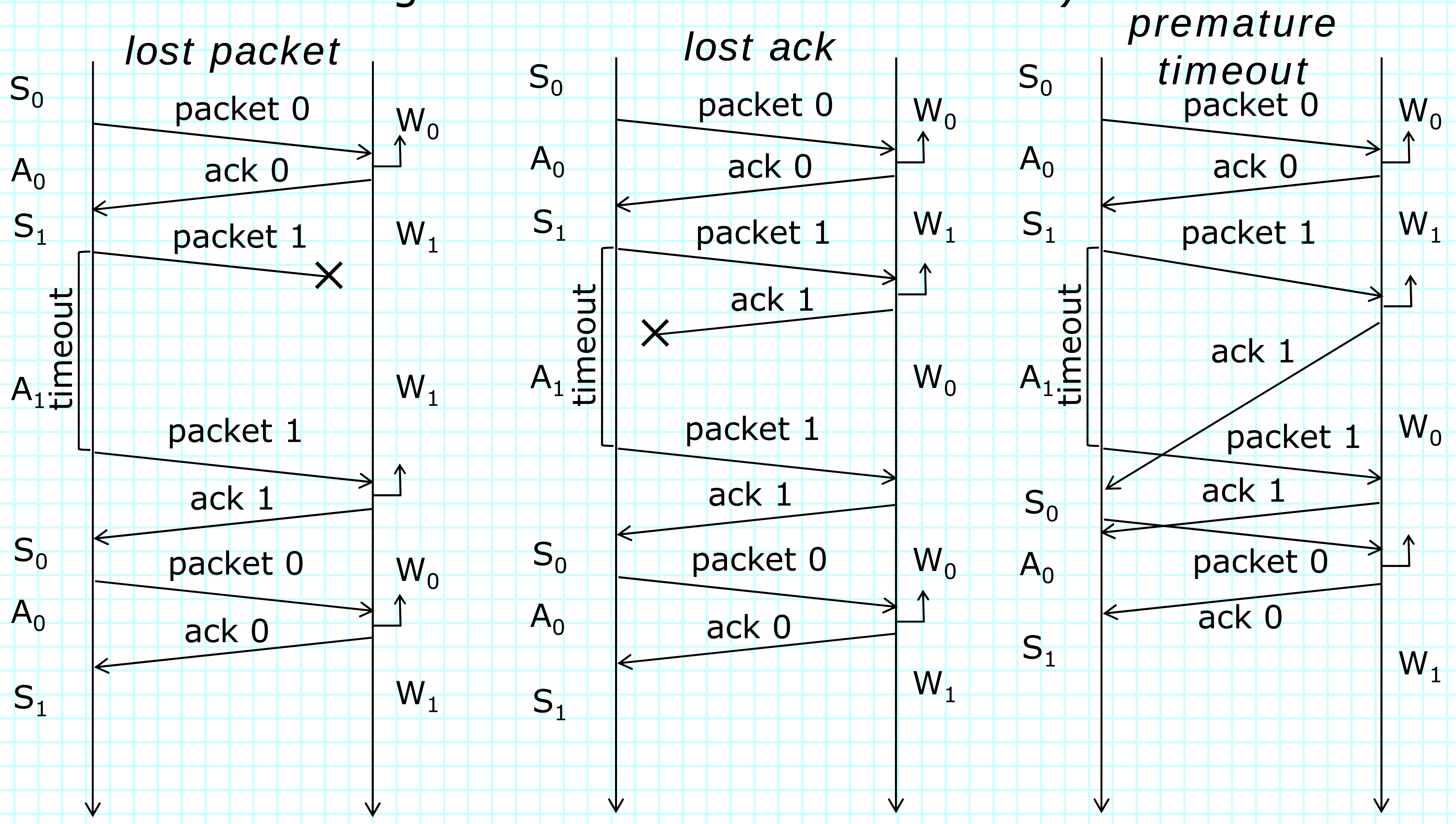
Exercise

1. For each of the time-space diagrams on slide 15, show the sender state and the receiver state at each point in time (just add state labels to the diagram next to the vertical lines).



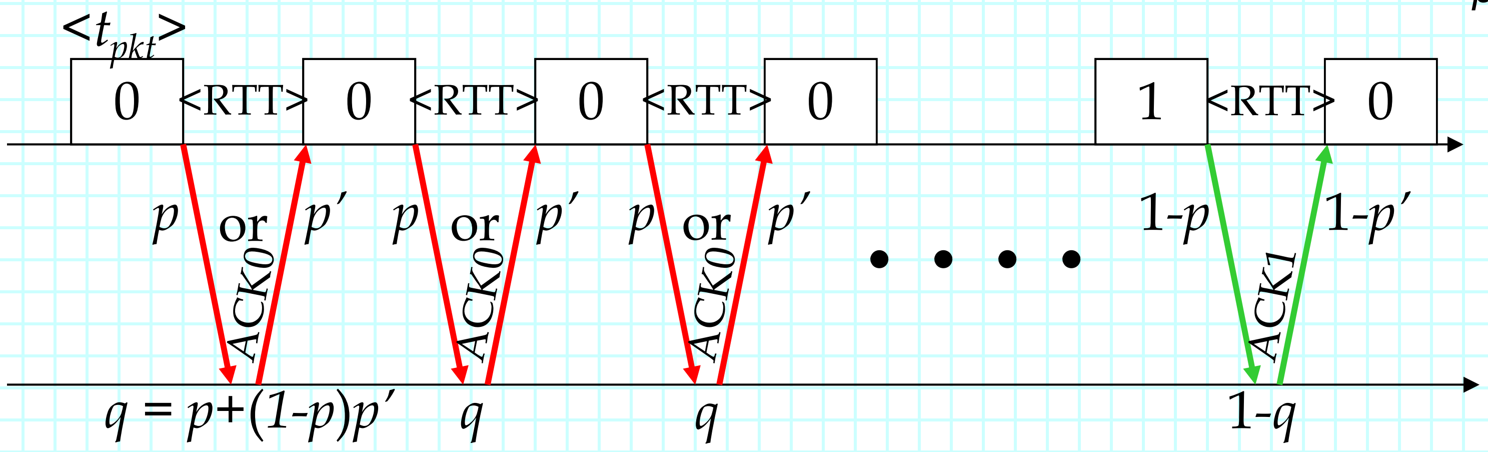
Exercise

1. For each of the time-space diagrams on slide 15, show the sender state and the receiver state at each point in time (just add state labels to the diagram next to the vertical lines).



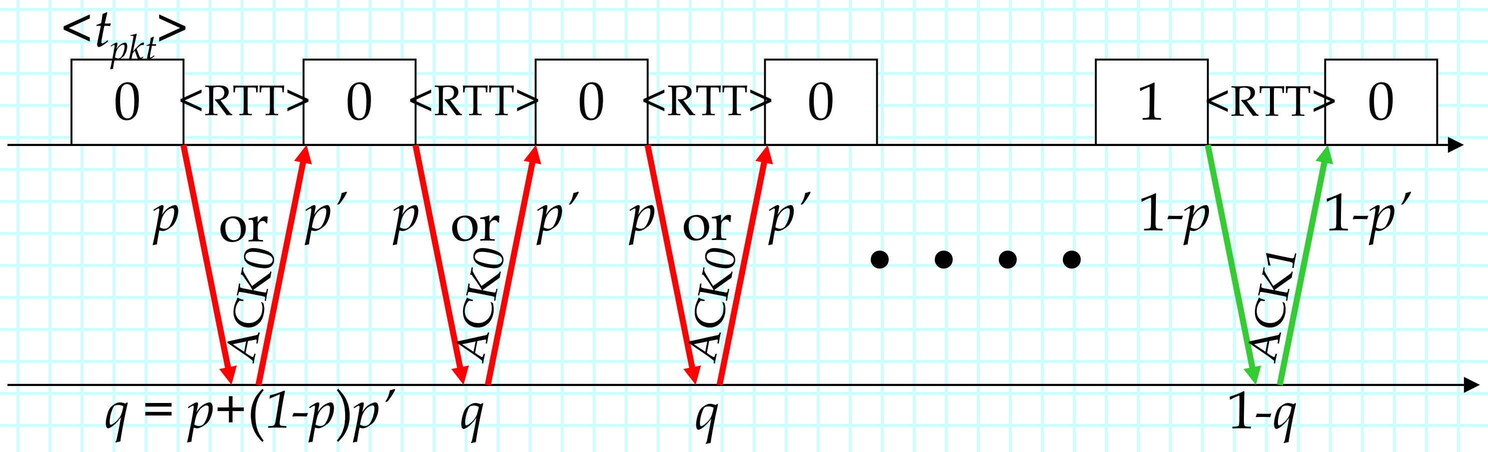
RDT 3.0 Performance

- RDT 3.0 sends only one packet at a time
  - works well if delay between sender and receiver is small
  - inefficient if RTT >> transmission time of a packet ($t_{pkt}=L/C$)



- RDT 3.0 performance as a function of
  - RTT (= t_{out})
  - link speed C
  - average packet size L
  - packet/ack loss/corruption probability q

RDT 3.0 Performance



- RDT 3.0 performance as a function of
  - » RTT = t_{out}, link speed C, average packet size L, and packet/ack loss/corruption probability q
- Expected time between successful transmissions

$$T_{\text{succ}} = \sum_{k=0}^{\infty}(k+1)(\text{RTT}+t_{pkt})q^k(1-q) = (1-q)(\text{RTT}+t_{pkt})\frac{1}{(1-q)^2} = \frac{\text{RTT}+t_{pkt}}{1-q}$$

- Throughput = L/T_{succ} = $C(1\text{-}q)/(1\text{+RTT}/t_{pkt})$
  - » Improves as q gets smaller (no surprise)
  - » Improves as RTT gets smaller compared to t_{pkt}

Exercise

- 1 Gb/s link with 20 ms RTT, L=10,000 bits, q=10^{-6}
  - » What is the link throughput under RDT 3.0
  - » What is the best possible throughput if the maximum packet size is 12,000 bits?

Exercise

- 1 Gb/s link with 20 ms RTT, L=10,000 bits, $q=10^{-6}$
  - » What is the link throughput under RDT 3.0

$$Throughput = C(1-q)/(1+RTT/t_{pkt}) = 10^{9}(1-10^{-6})/(1+2x10^{-2}/10^{-5})$$
$$\sim 10^{9}/2001 \sim 500\ kbps$$

  - » What is the best possible throughput if the maximum packet size is 12,000 bits?

Exercise

- 1 Gb/s link with 20 ms RTT, L=10,000 bits, $q=10^{-6}$
  - » What is the link throughput under RDT 3.0

Throughput $= C(1-q)/(1+RTT/t_{pkt}) = 10^9(1-10^{-6})/(1+2\text{x}10^{-2}/10^{-5})$

$\sim 10^9/2001 \sim 500$ *kbps*

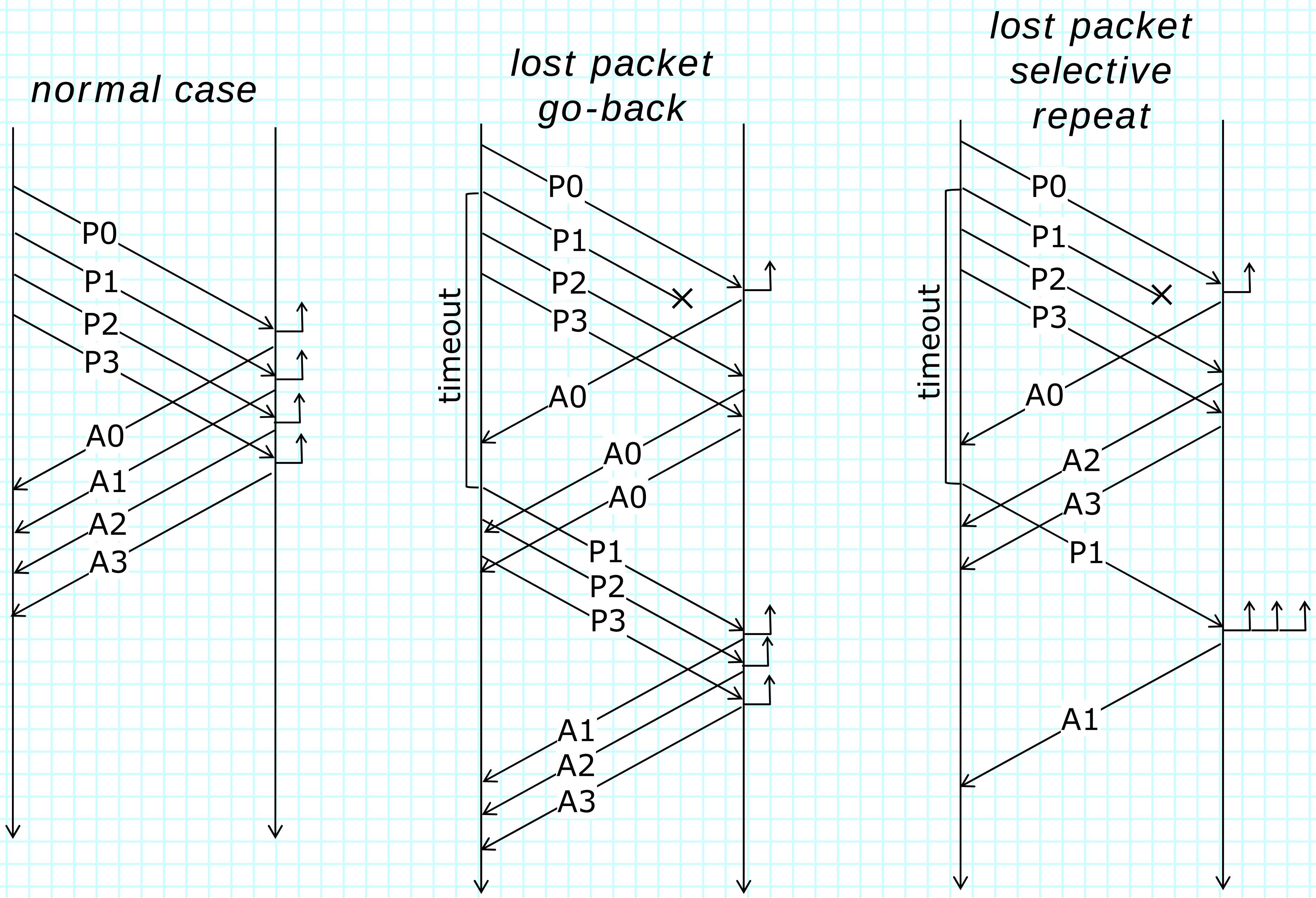
  - » What is the best possible throughput if the maximum packet size is 12,000 bits?

The only difference is that tpkt increases to $1.2\text{x}10^{-5}$*, so that the throughput is now about 600 kbps. Still about 2000 times slower than the link capacity!*

Improving RDT 3.0

- To enable more efficient operation, we want a protocol that can keep transmitting while waiting for acks, *i.e.*, *pipelines* transmissions
  - » sender can transmit up to *W* packets while waiting for acks
    - each packet labeled with sequence # from set of *N*>*W* sequence #s
  - » receiver sends acks with sequence numbers
- Basic goal is to be able to transmit non-stop under normal operation
  - » first ack must arrive before the sender finishes transmission of *W*th packet
- Two major variants
  - » go-back-*N* (cumulative ack) and selective repeat (specific ack)

Go-back-*N* with *W*=4



Go-Back-N Operation

Sender variables

- » Send window size: SWS
- » Last pkt sent: LPS
- » Last ack received: LAR

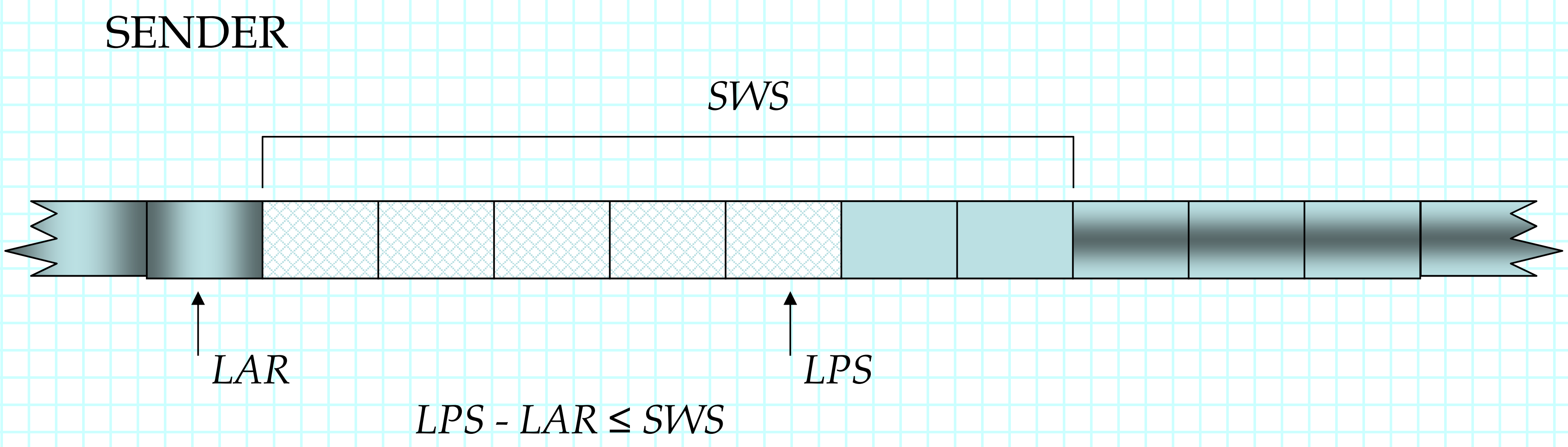
- Allow pkt transmissions as long as $LPS\text{-}LAR \le SWS$
  - » If $LPS\text{-}LAR \le SWS$, send next pkt with $SN = LPS+1$
- ACK for pkt number $X>LAR$ updates LAR to X (slides the window)
  - » Note that $X \neq LAR+1$ is OK (cumulative ACKs acknowledge all pkts up to X)

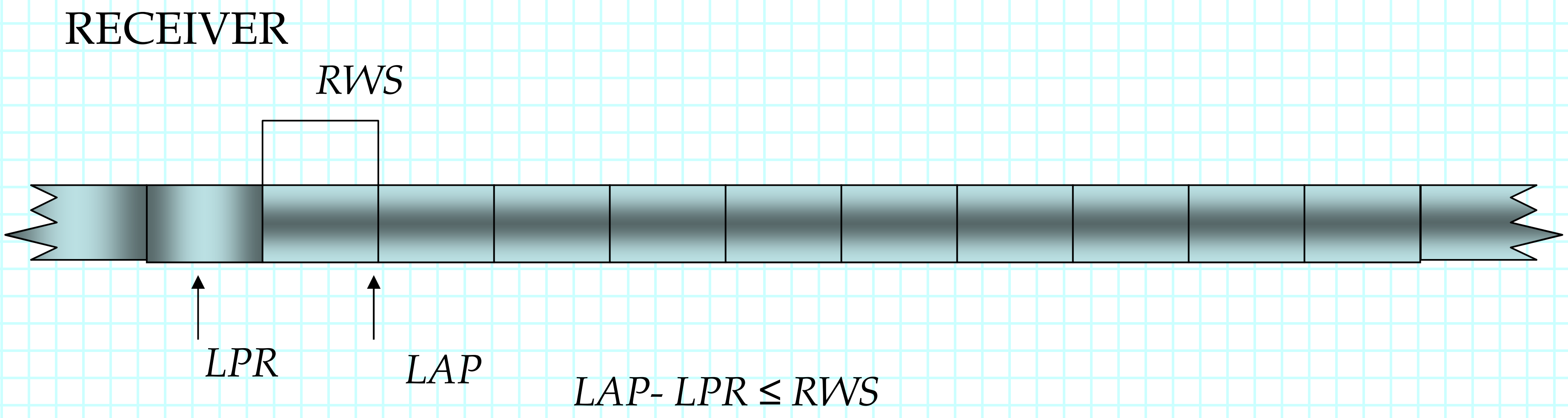
Receiver variables

- » Receive window size: $RWS = 1$
- » Last pkt received: LPR
- » Latest acceptable pkt: LAP

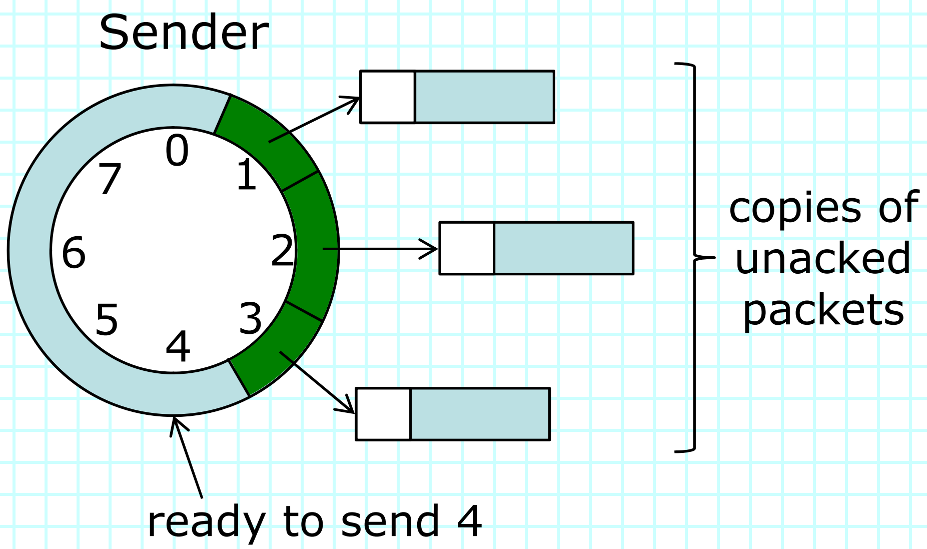
- Need: $LAP\text{-}LPR \le RWS$
  - » i.e., $LAP = RWS + LPR$ ($RWS=1 \Rightarrow$ only one acceptable pkt)
- When pkt SN arrives
  - » If $SN \le LPR$ or $SN > LAP$, discard pkt
  - » Else accept pkt and set LPR = SN
  - » Send (cumulative) ACK for LPR

Go-Back-N – Sender & Receiver





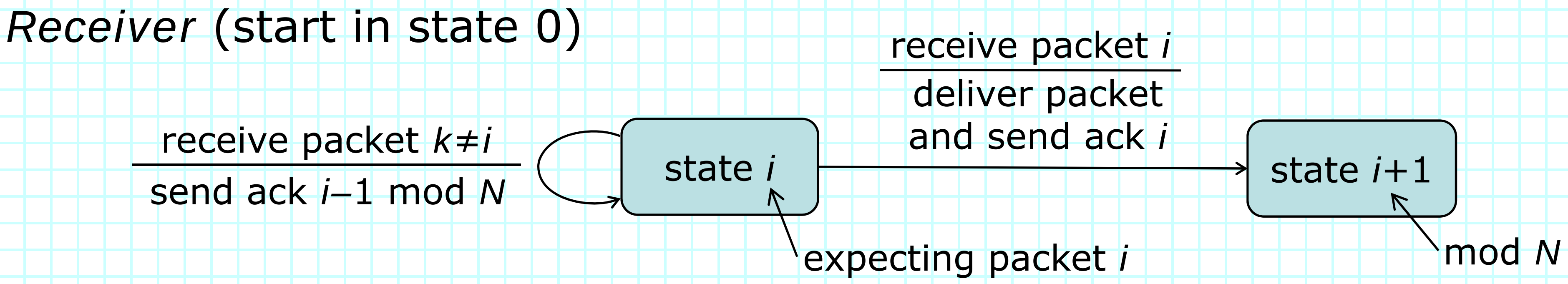
Buffering & Numbering Packets

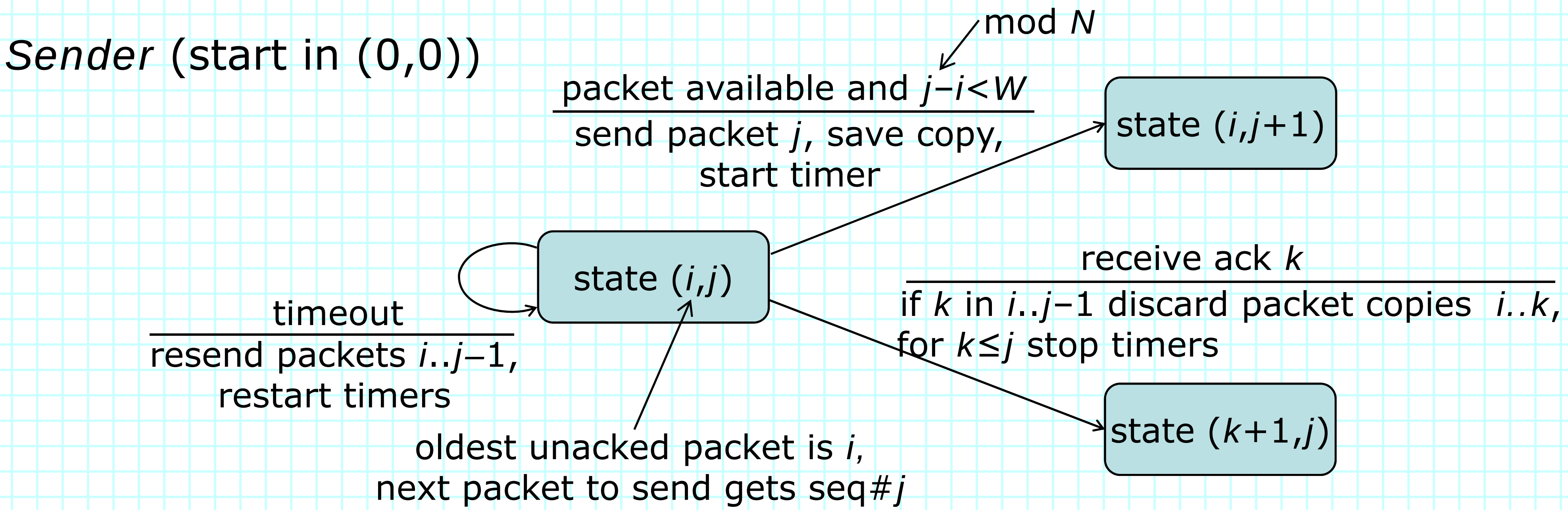


- Protocol uses sequence numbers 0..N–1=2^m-1
- Re-use sequence numbers in circular fashion
  - » all sequence number arithmetic is modulo N
- Window size SWS specifies number of pckts that can be unacknowledged

- How big should N be?
- Obviously) $N \geq SWS$
  - » Suppose N=4 (Seq. Num. field $\{0,\ldots,3\}$) and SWS = 4
    - Sender transmit frames $0,\ldots,3$
    - Arrive successfully, but all ACKs are lost
    - Sender retransmits $0,\ldots,3$ (after timeout)
    - Receiver expects **new** $0,\ldots,3$, gets second incarnation of **old** $0,\ldots,3$, incorrectly accepts as new
- In general, we need
  - » $RWS + SWS \leq N$

Go-Back-*N* State Machines


Receiver (start in state 0)
receive packet *k*≠*i* / send ack *i*–1 mod *N*
state *i*
receive packet *i* / deliver packet and send ack *i*
state *i*+1
expecting packet *i*
mod *N*


Sender (start in (0,0))
mod *N*
packet available and *j*–*i*<*W* / send packet *j*, save copy, start timer
state (*i*,*j*+1)
timeout / resend packets *i*..*j*–1, restart timers
state (*i*,*j*)
receive ack *k* / if *k* in *i*..*j*−1 discard packet copies *i*..*k*, for *k*≤*j* stop timers
state (*k*+1,*j*)
oldest unacked packet is *i*, next packet to send gets seq#*j*

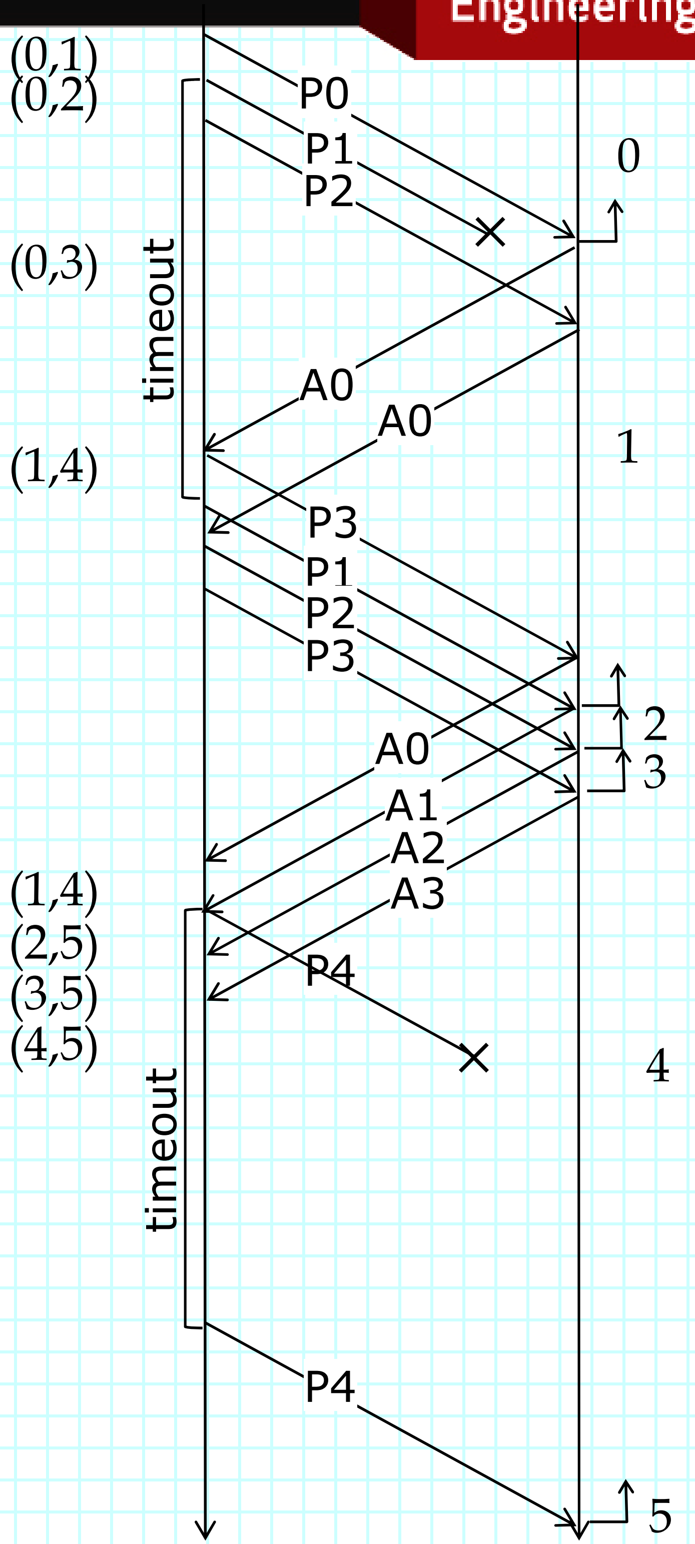
Exercises

1. Draw a time-space diagram for the go-back-N protocol with $W=3$, $N=6$, assuming that five packets are sent, and the second and fifth packets are lost the first time they are sent. Show the states of the sender and receiver.

Go-back-*N* with *W*=3, *N*=6

1. Draw a time-space diagram for the go-back-*N* protocol with *W*=3, *N*=6, assuming that five packets are sent, and the second and fifth packets are lost the first time they are sent. Show the states of the sender and receiver.

State starts at (0,0) and eventually ends at (5,5) when ack for P4 is received



Exercises

1. Consider a link that uses go-back-*N* with *N*=10, *W*=5. Suppose 4 packets have been sent and 2 acks have been received. What is the smallest possible number of packets that could still be in the send buffer? What is the largest number?

Exercises

1. Consider a link that uses go-back-*N* with *N*=10, *W*=5. Suppose 4 packets have been sent and 2 acks have been received. What is the smallest possible number of packets that could still be in the send buffer? What is the largest number?

Consider a scenario, where the first and second packets are lost. This will trigger ACKs when the third and fourth packets arrive, acknowledging receipt of the packet before the first (or the SYN packet). Under such a scenario, the two acks will not be for any of the four packets that have been sent, so that the sender will still have all four in its send buffer.

In the best case, the two acks are for the 3rd and 4th packets and there are zero packets left in the send buffer (acks for packets 1 and 2 were lost).

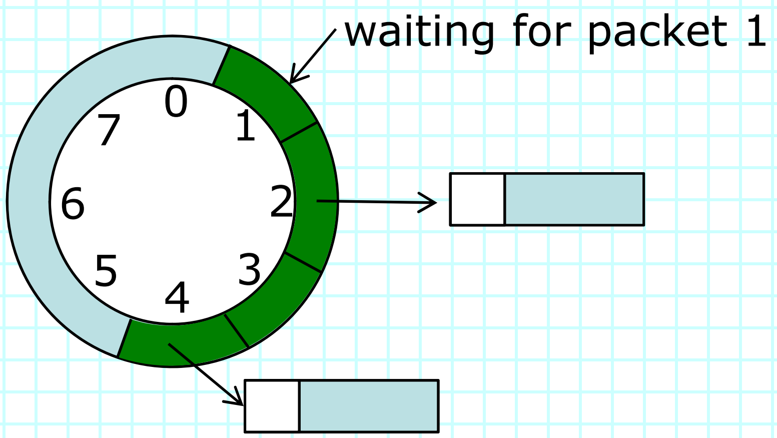
Go-Back-N Performance

- Performance depends on
  - Impact and frequency of retransmissions
  - Time-out value. Assume $t_{out} = (N\text{-}1)t_{pkt} \geq \mathrm{RTT}$
- Let T_{total} be the time spent for one ***retransmission***
  - $T_{total} = t_{pkt} + t_{out} = \alpha t_{pkt}$
- Time spent per packet: $T = RT_{total} + t_{pkt}$
  - R is number of ***retransmissions*** (geometrically distributed)
  - E[*R*]=*q*/(1-*q*), where *q*~*p* the pkt loss probability
- Expected time for successful packet transmission

$$T_{succ} = E(R)T_{total} + t_{pkt} = \frac{1+(\alpha-1)q}{1-q}t_{pkt}$$

- 100% throughput if $t_{pkt} = T_{succ}$
  - Achievable when *q*=0
  - For a given value of *q*, throughput decreases as α, i.e., RTT, increases

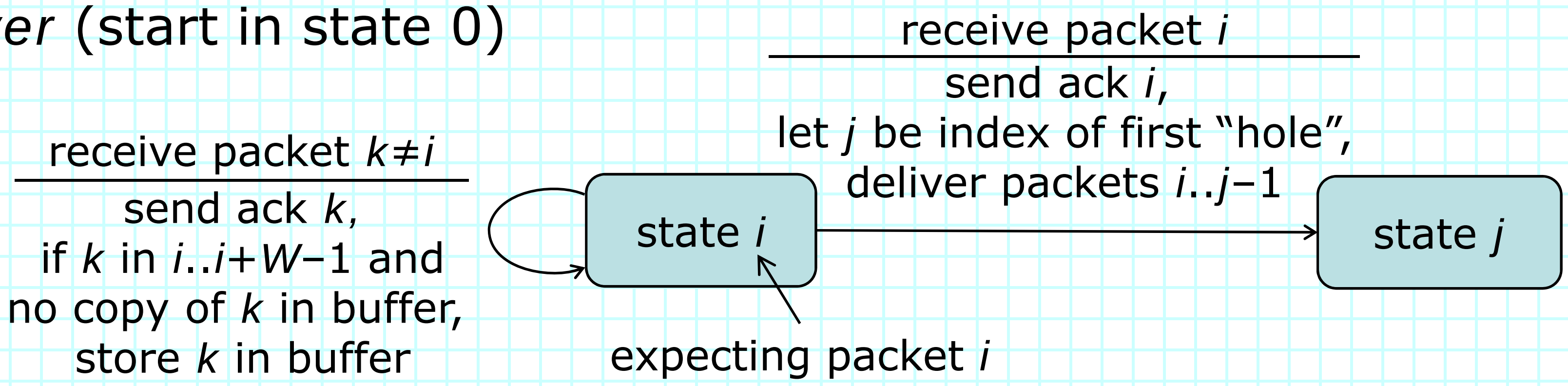
Receive Buffer in Selective Repeat



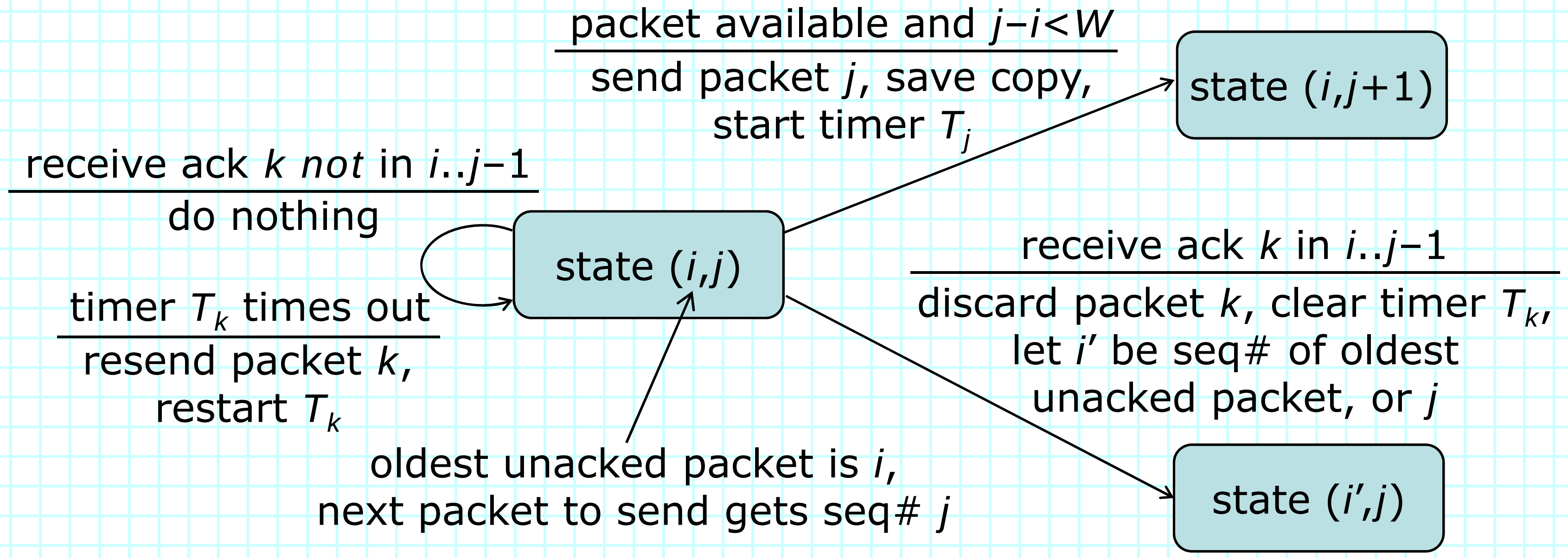
- Receiver maintains buffer of up to *RWS*=*SWS* packets
  - » On receiving packet *i*, send ack *i*
  - » If arriving packet is in window range and no previous copy is in buffer, store new packet
  - » If arriving packet is next one expected
    - deliver it and all other packets up until the first “hole” in buffer
    - advance window to location of first hole
- Correct operation requires in-order packets and $N \geq 2W$

Selective Repeat State Machines

Receiver (start in state 0)



Sender (start in (0,0))


packet available and $j-i<W$
send packet j, save copy,
start timer T_j
state $(i,j+1)$
receive ack k *not* in $i..j-1$
do nothing
state (i,j)
timer T_k times out
resend packet k,
restart T_k
receive ack k in $i..j-1$
discard packet k, clear timer T_k,
let i' be seq# of oldest
unacked packet, or j
state (i',j)
oldest unacked packet is i,
next packet to send gets seq# j

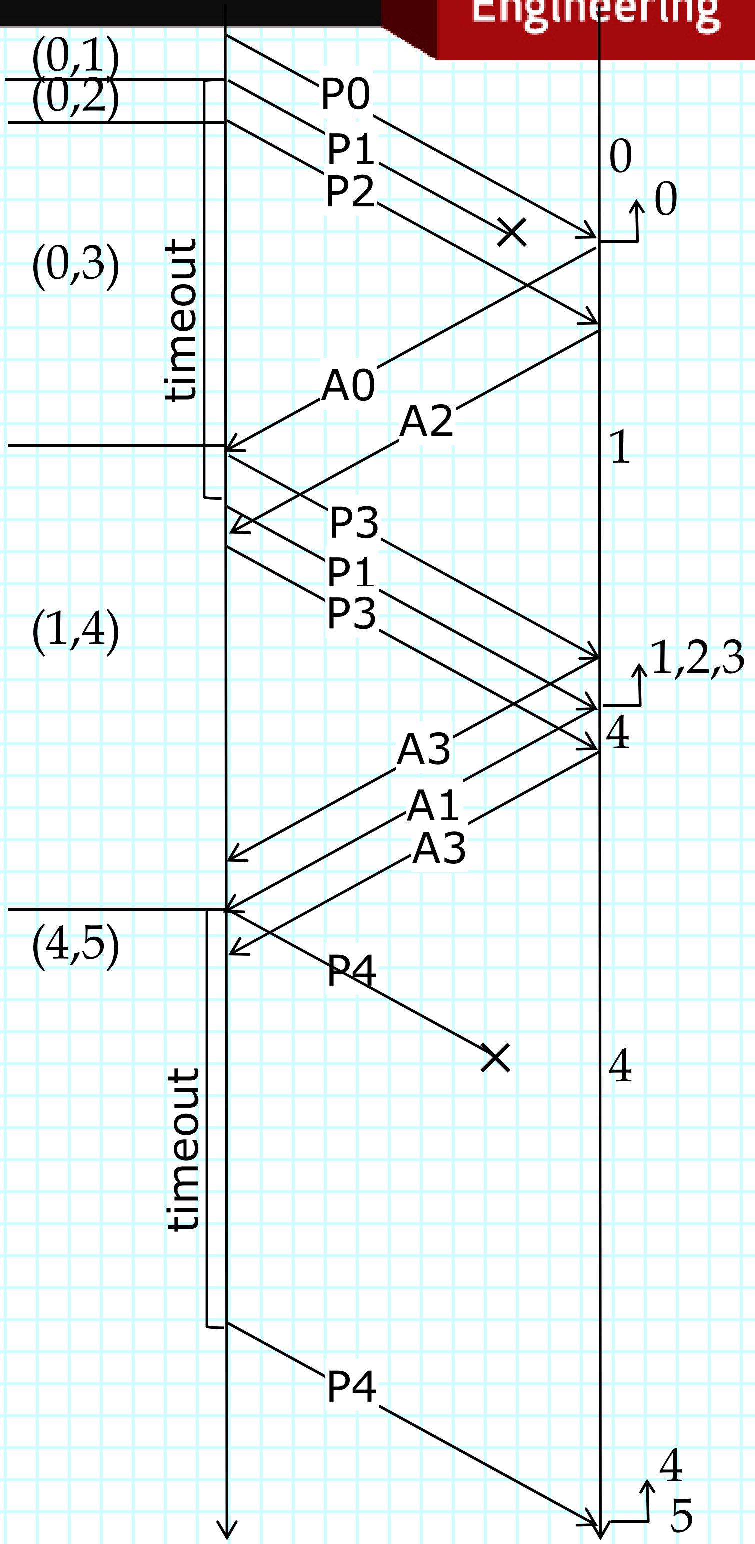
Exercises

1. Draw a time-space diagram for the selective repeat protocol with W=3, N=6, assuming that five packets are sent, and the second and fifth packets are lost the first time they are sent. Show when each packet is delivered and the states of the sender and receiver at each step.

Selective Repeat with *W*=3, *N*=6

1. Draw a time-space diagram for the selective repeat protocol with *W*=3, *N*=6, assuming that five packets are sent, and the second and fifth packets are lost the first time they are sent. Show when each packet is delivered and the states of the sender and receiver at each step.

State starts at (0,0) and eventually ends at (5,5) when ack for P4 is received



Exercises

1. Consider a link that uses selective repeat with N=10, W=5. Suppose 4 packets have been sent and 2 acks have been received. What is the smallest possible number of packets that could still be in the send buffer? What is the largest number?

Exercises

1. Consider a link that uses selective repeat with N=10, W=5. Suppose 4 packets have been sent and 2 acks have been received. What is the smallest possible number of packets that could still be in the send buffer? What is the largest number?

With selective repeat, packets are acked individually. So in all cases, the receipt of two ACKs identifies two packets (out of four) that have been received. Hence, there will then be two packets in the sender's send buffer (even if all four packets were successfully received).

Note that in practice, selective acknowledgments are used in combination with cumulative ACKs, e.g., through the use of a SACK option as in TCP (see RFC 2018). This helps overcome the fragility problem caused by requiring that each packet be individually acked. In particular, TCP's SACK option allows the receiver to inform the sender of multiple (up to 4) non-contiguous blocks of data it received. It is used in combination with regular ACKs.