

Reliable Data Transfer II

Antonio Carzaniga

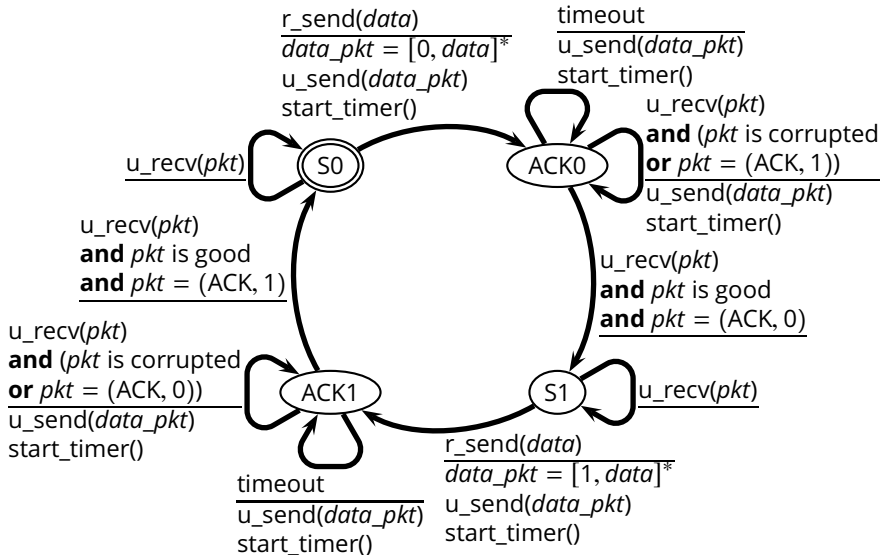
Faculty of Informatics
Università della Svizzera italiana

November 8, 2017

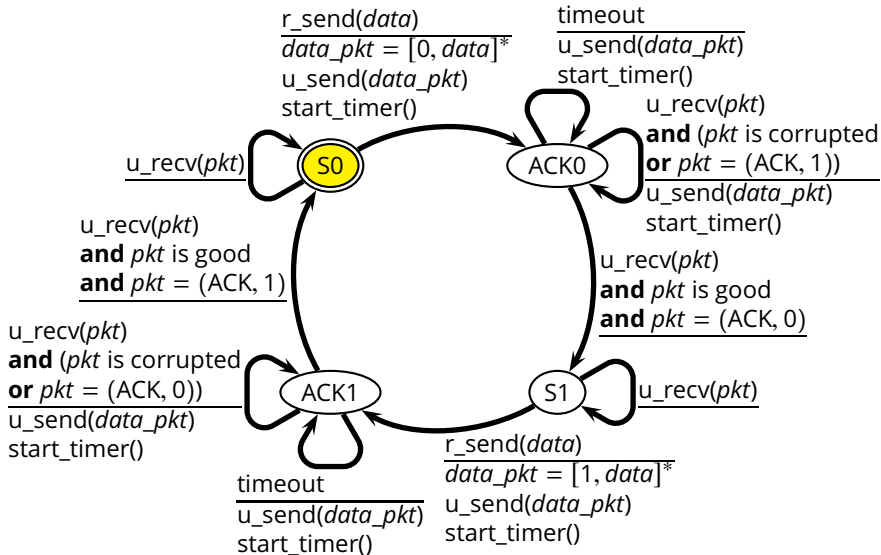
- Performance of the stop-and-wait protocol
- Go-Back-N
- Selective repeat

Back to Reliable Data Transfer

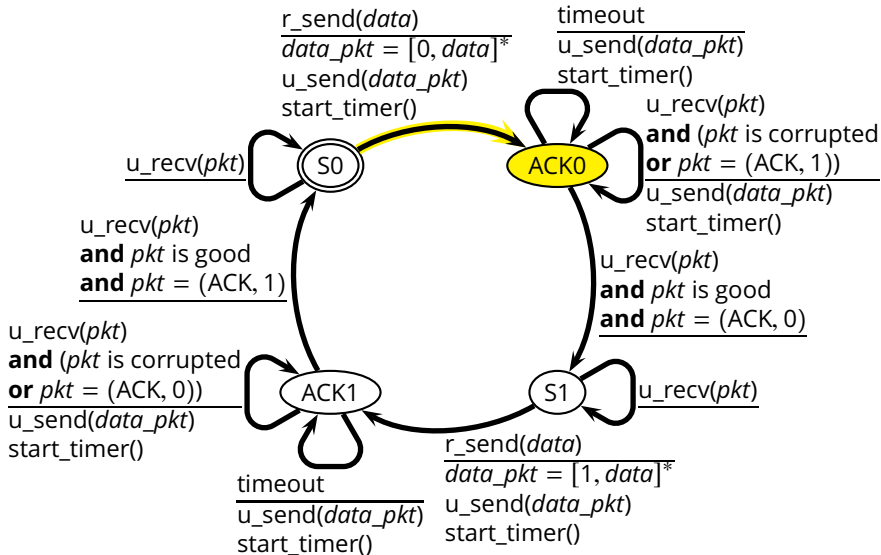
Back to Reliable Data Transfer



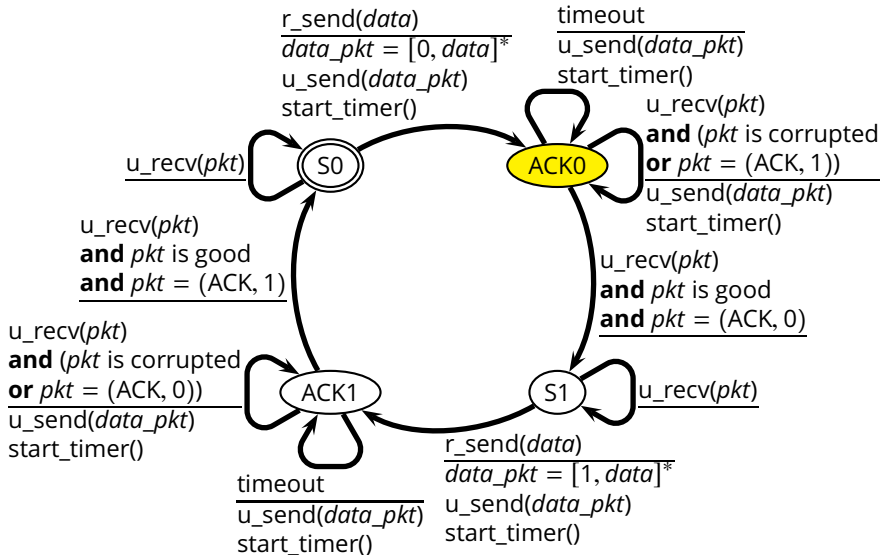
Back to Reliable Data Transfer



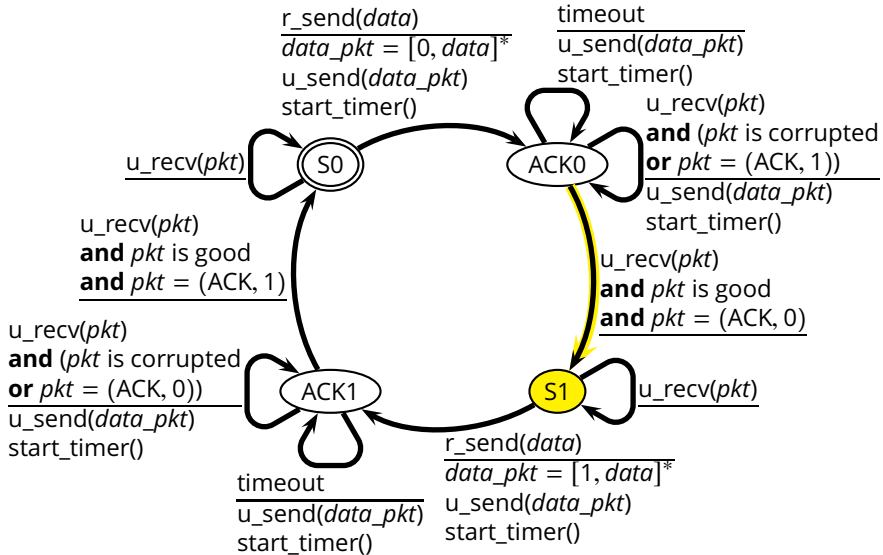
Back to Reliable Data Transfer



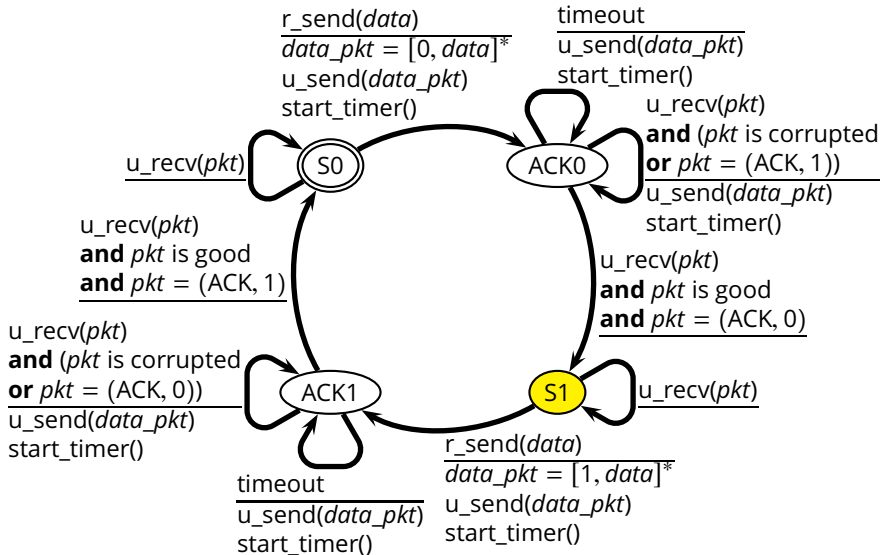
Back to Reliable Data Transfer



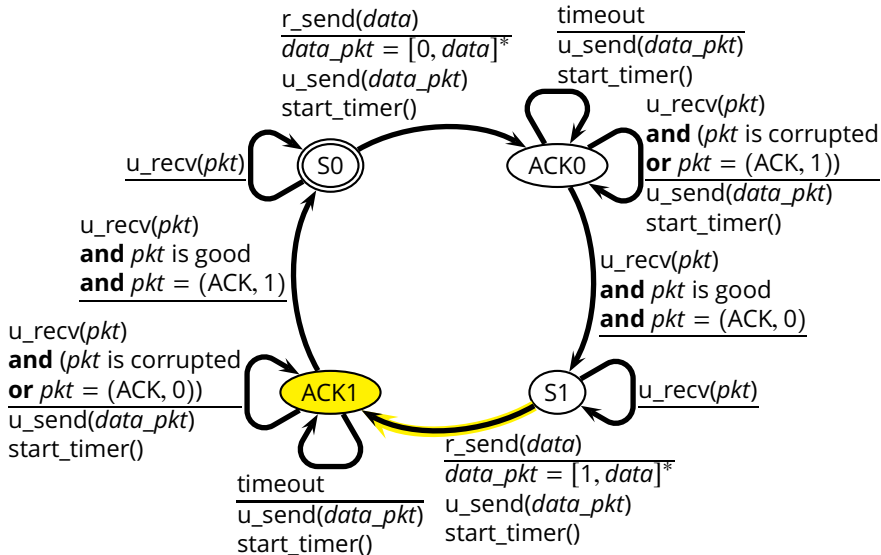
Back to Reliable Data Transfer



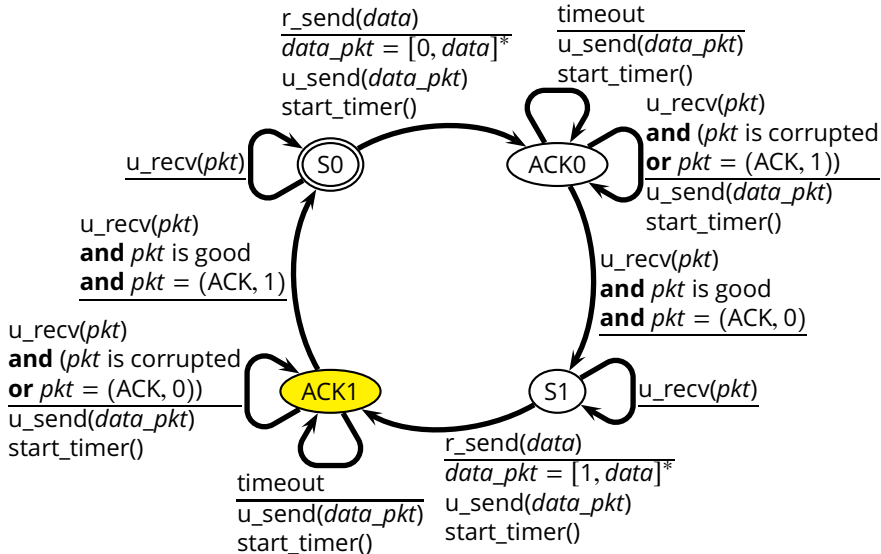
Back to Reliable Data Transfer



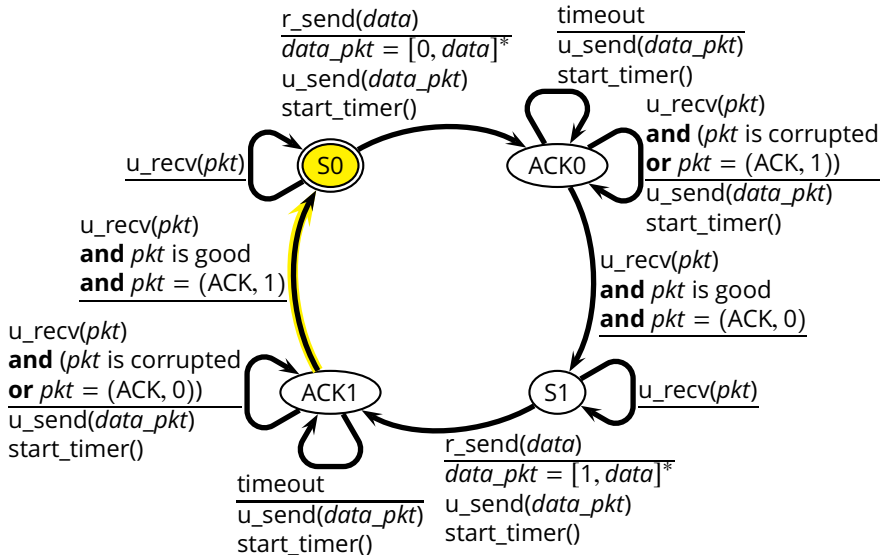
Back to Reliable Data Transfer



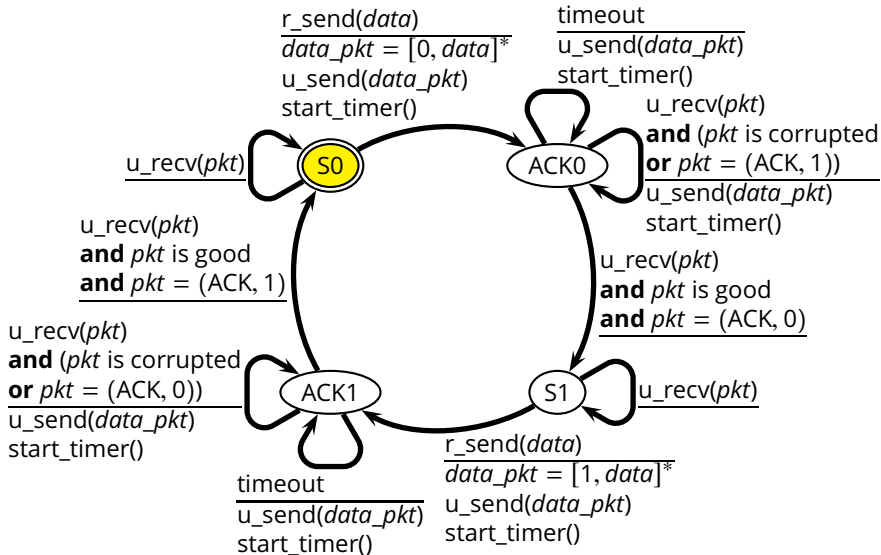
Back to Reliable Data Transfer

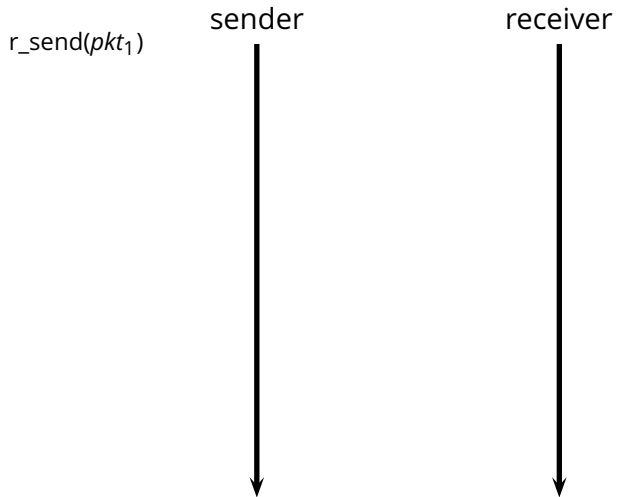


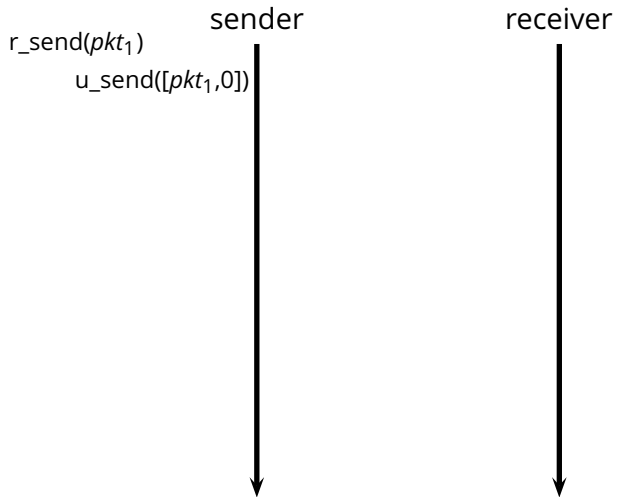
Back to Reliable Data Transfer

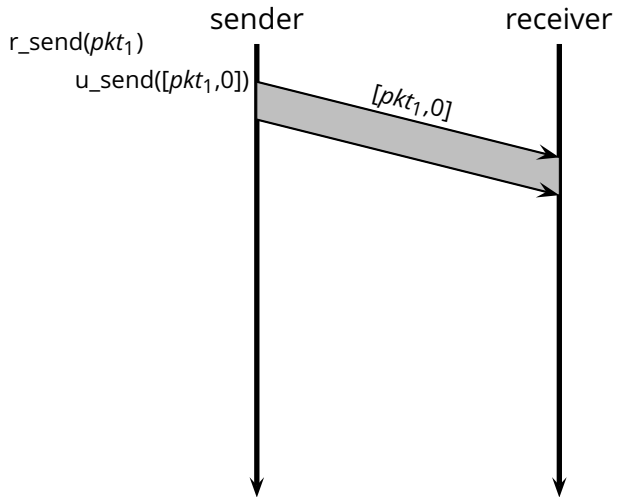


Back to Reliable Data Transfer

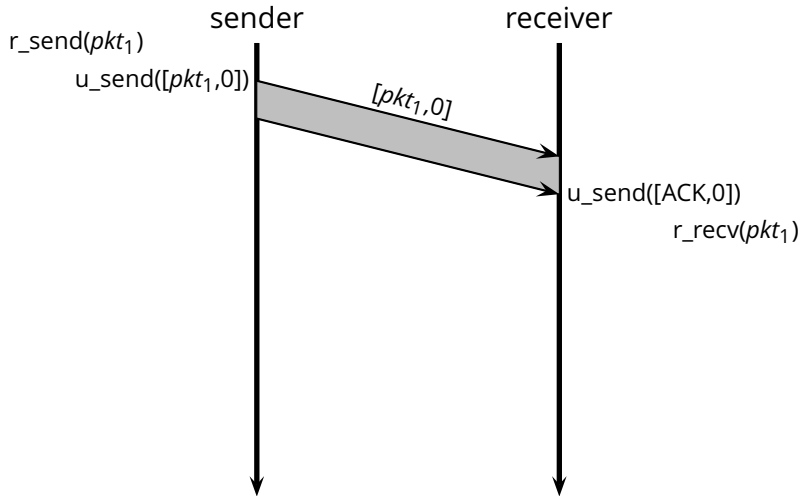




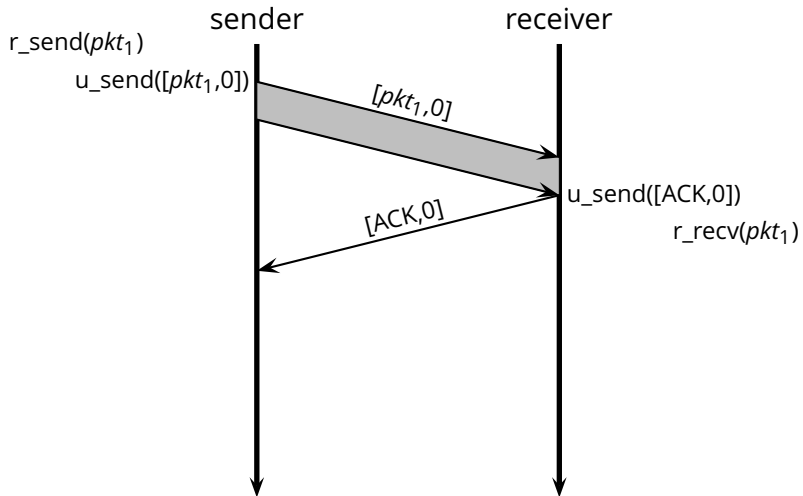




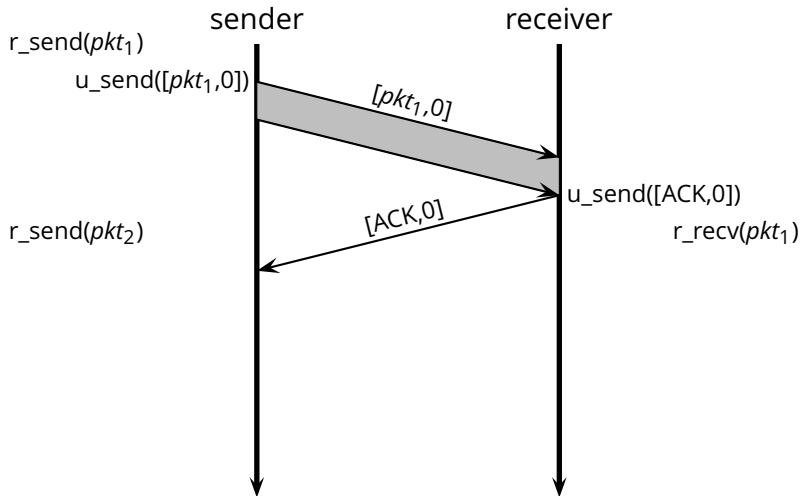
Network Usage



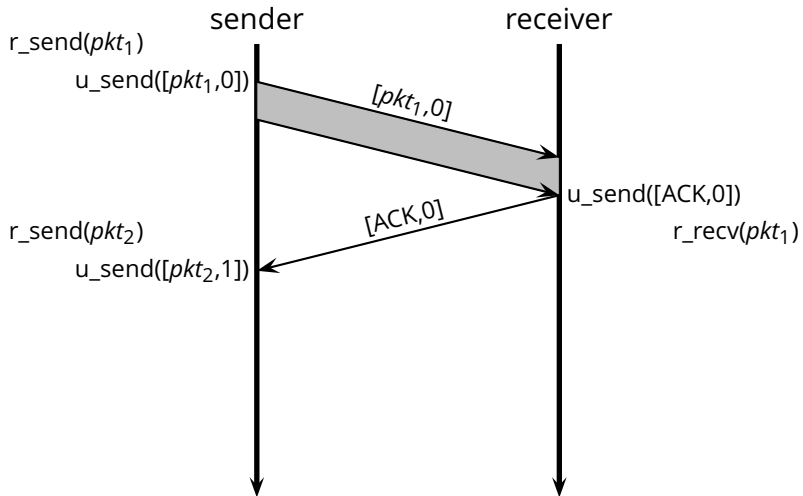
Network Usage



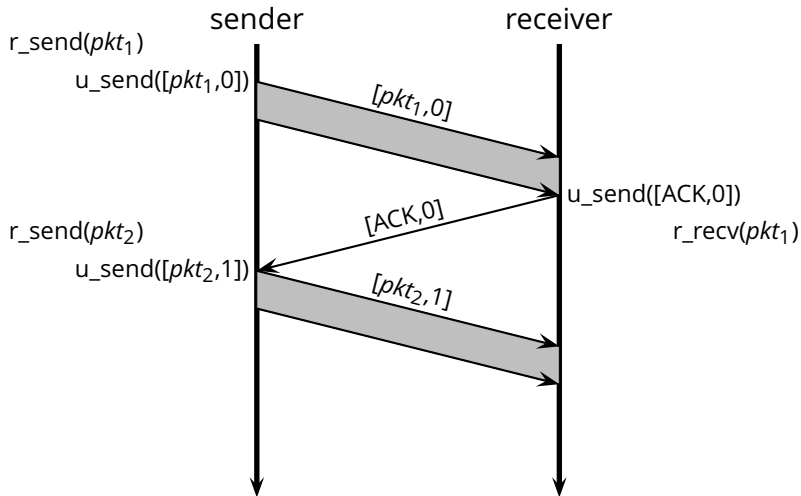
Network Usage



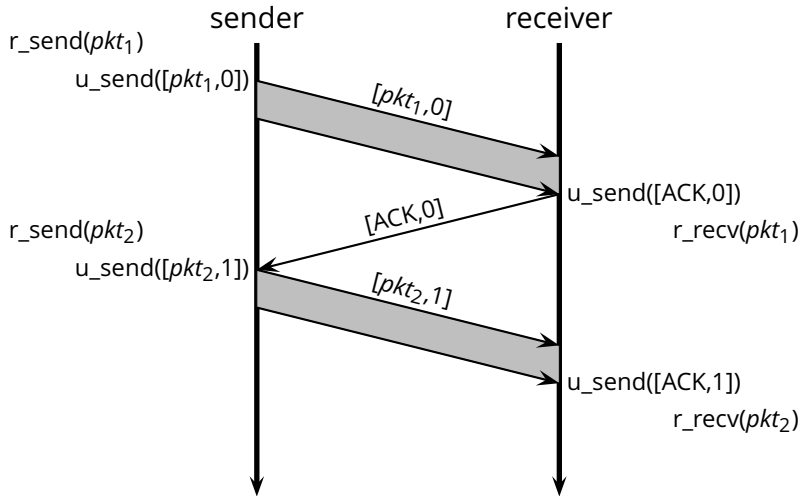
Network Usage



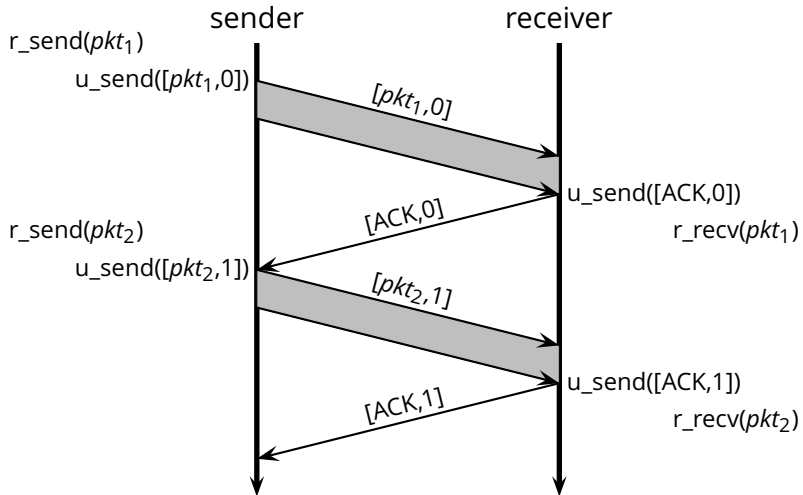
Network Usage

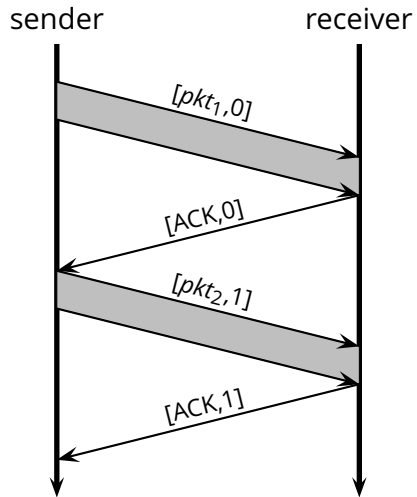


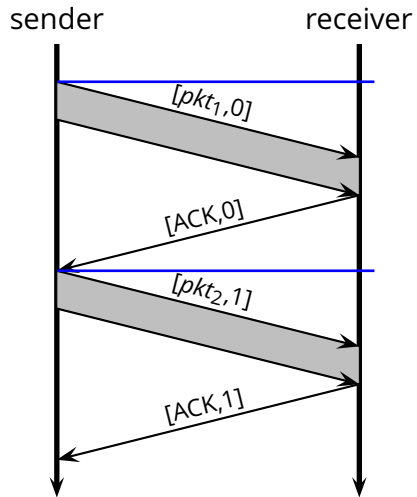
Network Usage

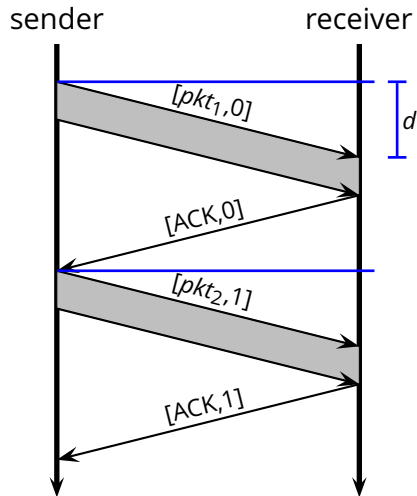


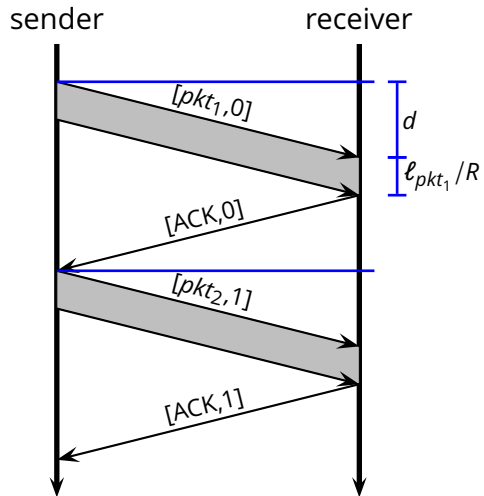
Network Usage

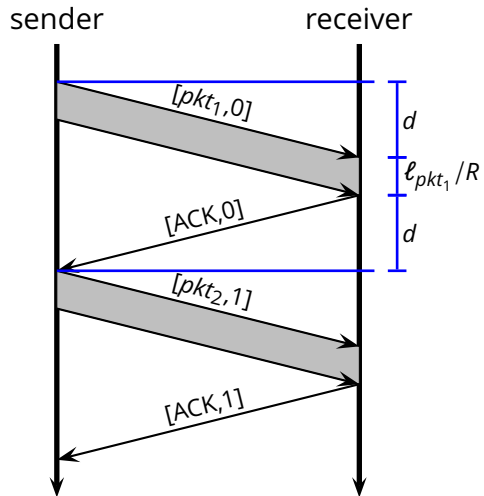


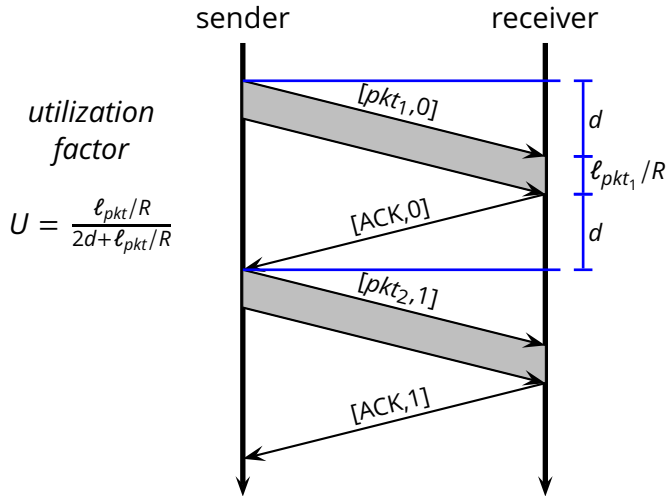








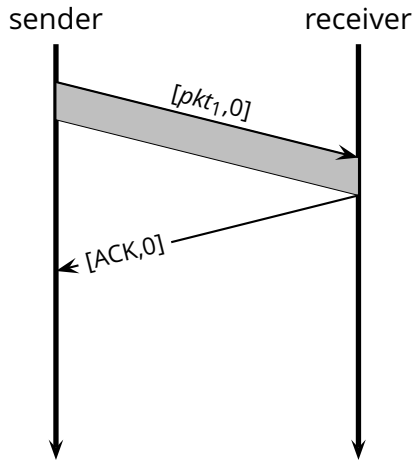




- How do we achieve a better *utilization factor*?

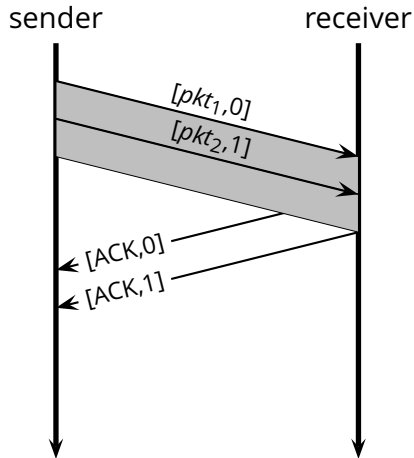
Improving Network Usage

- How do we achieve a better *utilization factor*?



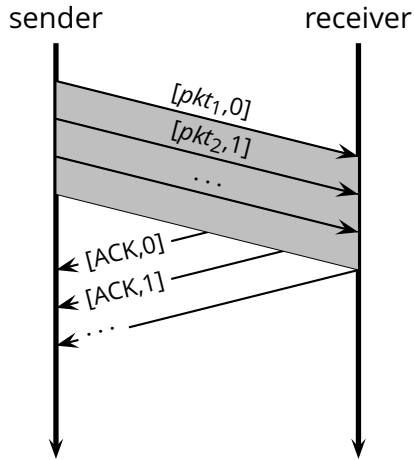
Improving Network Usage

- How do we achieve a better *utilization factor*?

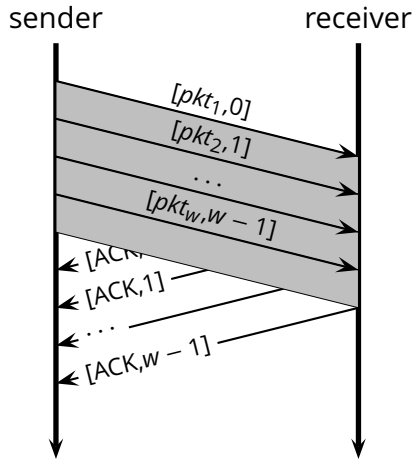


Improving Network Usage

- How do we achieve a better *utilization factor*?



- How do we achieve a better *utilization factor*?



- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement

- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to W unacknowledged packets in the pipeline
 - ▶ the sender's state machine gets very complex
 - ▶ we represent the sender's state with its queue of acknowledgements

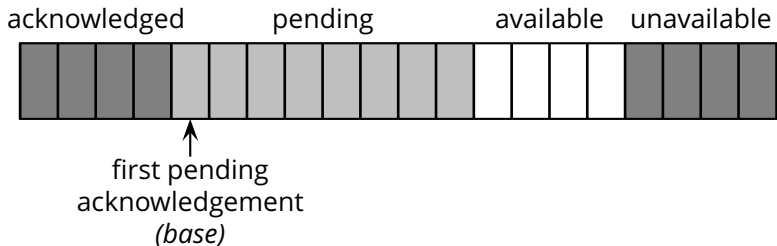
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to W unacknowledged packets in the pipeline
 - ▶ the sender's state machine gets very complex
 - ▶ we represent the sender's state with its queue of acknowledgements



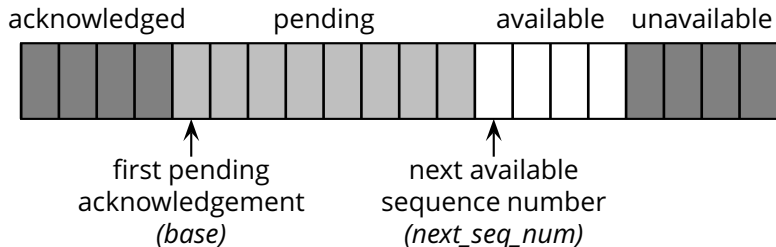
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to W unacknowledged packets in the pipeline
 - ▶ the sender's state machine gets very complex
 - ▶ we represent the sender's state with its queue of acknowledgements



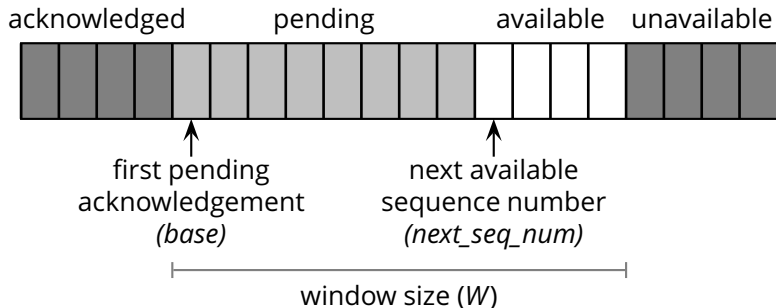
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to W unacknowledged packets in the pipeline
 - ▶ the sender's state machine gets very complex
 - ▶ we represent the sender's state with its queue of acknowledgements



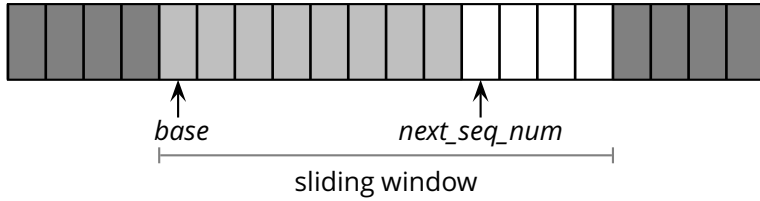
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to W unacknowledged packets in the pipeline
 - ▶ the sender's state machine gets very complex
 - ▶ we represent the sender's state with its queue of acknowledgements



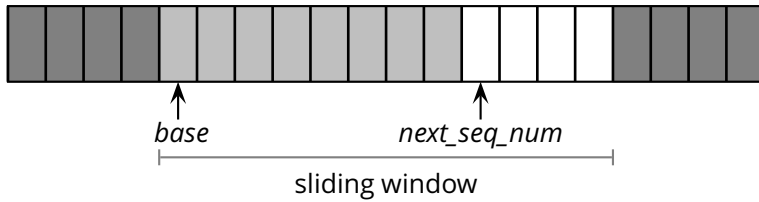
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to W unacknowledged packets in the pipeline
 - ▶ the sender's state machine gets very complex
 - ▶ we represent the sender's state with its queue of acknowledgements



Sliding Window Protocol: Sender

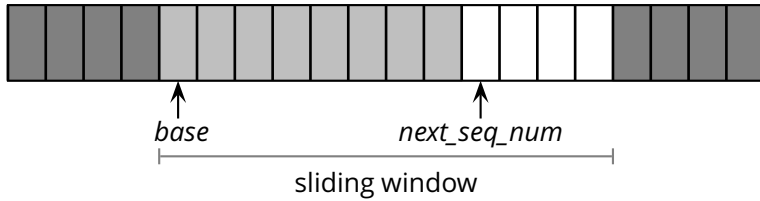


Sliding Window Protocol: Sender



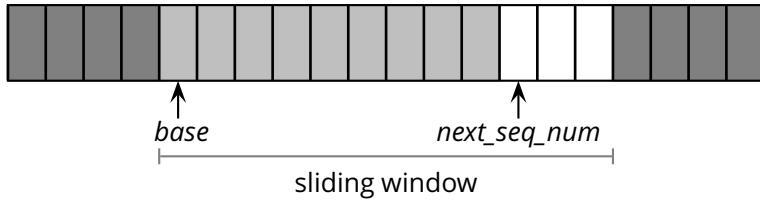
■ $r_send(pkt_1)$

Sliding Window Protocol: Sender



- $r_send(pkt_1)$
 - ▶ $u_send([pkt_1, next_seq_num])$

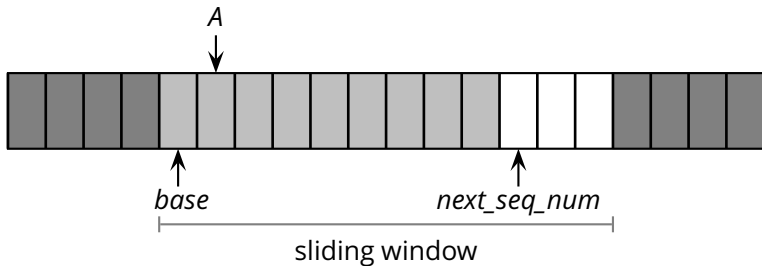
Sliding Window Protocol: Sender



■ $r_send(pkt_1)$

- ▶ $u_send([pkt_1, next_seq_num])$
- ▶ $next_seq_num = next_seq_num + 1$

Sliding Window Protocol: Sender

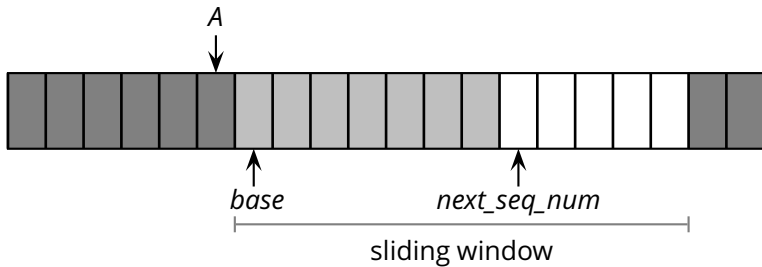


■ $r_send(pkt_1)$

- ▶ $u_send([pkt_1, next_seq_num])$
- ▶ $next_seq_num = next_seq_num + 1$

■ $u_rcv([ACK, A])$

Sliding Window Protocol: Sender



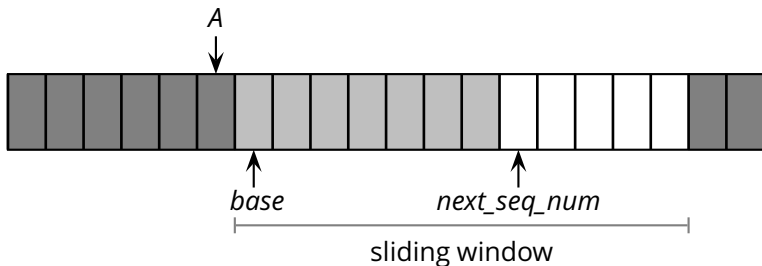
■ $r_send(pkt_1)$

- ▶ $u_send([pkt_1, next_seq_num])$
- ▶ $next_seq_num = next_seq_num + 1$

■ $u_rcv([ACK, A])$

- ▶ $base = A + 1$

Sliding Window Protocol: Sender



■ $r_send(pkt_1)$

- ▶ $u_send([pkt_1, next_seq_num])$
- ▶ $next_seq_num = next_seq_num + 1$

■ $u_rcv([ACK, A])$

- ▶ $base = A + 1$
- ▶ notice that acknowledgements are “cumulative”

Sliding Window Protocol: Sender

Sliding Window Protocol: Sender

- The sender remembers the first sequence number that has not yet been acknowledged
 - ▶ or the highest acknowledged sequence number
- The sender remembers the first available sequence number
 - ▶ or the highest used sequence number (i.e., sent to the receiver)
- The sender responds to three types of events

Sliding Window Protocol: Sender

- The sender remembers the first sequence number that has not yet been acknowledged
 - ▶ or the highest acknowledged sequence number
- The sender remembers the first available sequence number
 - ▶ or the highest used sequence number (i.e., sent to the receiver)
- The sender responds to three types of events
 - ▶ *r_send()*: invocation from the application layer: send more data if a sequence number is available

Sliding Window Protocol: Sender

- The sender remembers the first sequence number that has not yet been acknowledged
 - ▶ or the highest acknowledged sequence number
- The sender remembers the first available sequence number
 - ▶ or the highest used sequence number (i.e., sent to the receiver)
- The sender responds to three types of events
 - ▶ *r_send()*: invocation from the application layer: send more data if a sequence number is available
 - ▶ *ACK*: receipt of an acknowledgement: shift the window (it's a "cumulative" ACK)

Sliding Window Protocol: Sender

- The sender remembers the first sequence number that has not yet been acknowledged
 - ▶ or the highest acknowledged sequence number
- The sender remembers the first available sequence number
 - ▶ or the highest used sequence number (i.e., sent to the receiver)
- The sender responds to three types of events
 - ▶ *r_send()*: invocation from the application layer: send more data if a sequence number is available
 - ▶ *ACK*: receipt of an acknowledgement: shift the window (it's a "cumulative" ACK)
 - ▶ *timeout*: "Go-Back-N." I.e., resend all the packets that have been sent but not acknowledged

Sliding Window Protocol: Sender

- *init*

base = 1

next_seq_num = 1

Sliding Window Protocol: Sender

- *init*

base = 1

next_seq_num = 1

- *r_send(data)*

if *next_seq_num* < *base* + *W*:

pkt[next_seq_num] = [*next_seq_num*, *data*]*

u_send(pkt[next_seq_num])

if *next_seq_num* == *base*:

start_timer()

next_seq_num = *next_seq_num* + 1

else:

refuse_data(data) // block the sender

Sliding Window Protocol: Sender

- u_rcv(pkt) and pkt is corrupted

Sliding Window Protocol: Sender

- u_rcv(pkt) and pkt is corrupted

- u_rcv(ACK,ack_num)

base = ack_num + 1 // resume the sender

if *next_seq_num == base:*

 stop_timer()

else:

 start_timer()

Sliding Window Protocol: Sender

- u_rcv(pkt) and pkt is corrupted

- u_rcv(ACK,ack_num)

base = ack_num + 1 // resume the sender

if *next_seq_num == base*:

 stop_timer()

else:

 start_timer()

- timeout

start_timer()

foreach *i in base ... next_seq_num - 1*:

 u_send(pkt[i])

Sliding Window Protocol: Receiver

Sliding Window Protocol: Receiver

- Simple: as in the stop-and-wait case, the receiver maintains a counter representing the *expected sequence number*

Sliding Window Protocol: Receiver

- Simple: as in the stop-and-wait case, the receiver maintains a counter representing the *expected sequence number*
- The receiver waits for a (good) data packet with the expected sequence number

Sliding Window Protocol: Receiver

- Simple: as in the stop-and-wait case, the receiver maintains a counter representing the *expected sequence number*
- The receiver waits for a (good) data packet with the expected sequence number
 - ▶ acknowledges the expected sequence number

Sliding Window Protocol: Receiver

- Simple: as in the stop-and-wait case, the receiver maintains a counter representing the *expected sequence number*
- The receiver waits for a (good) data packet with the expected sequence number
 - ▶ acknowledges the expected sequence number
 - ▶ delivers the data to the application

Sliding Window Protocol: Receiver

- *init*

$expected_seq_num = 1$

$ackpkt = [ACK, 0]^*$

Sliding Window Protocol: Receiver

- *init*

$expected_seq_num = 1$

$ackpkt = [ACK, 0]^*$

- $u_recv([data, seq_num])$ **and** good
and $seq_num = expected_seq_num$

$r_recv(data)$

$ackpkt = [ACK, expected_seq_num]^*$

$expected_seq_num = expected_seq_num + 1$

$u_send(ackpkt)$

Sliding Window Protocol: Receiver

- *init*

$expected_seq_num = 1$
 $ackpkt = [ACK, 0]^*$

- $u_recv([data, seq_num])$ **and** good
and $seq_num = expected_seq_num$

$r_recv(data)$
 $ackpkt = [ACK, expected_seq_num]^*$
 $expected_seq_num = expected_seq_num + 1$
 $u_send(ackpkt)$

- $u_recv([data, seq_num])$
and (corrupted **or** $seq_num \neq expected_seq_num$)

$u_send(ackpkt)$

- Concepts

- Concepts

- ▶ *sequence numbers*

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*
- ▶ *cumulative acknowledgements*

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*
- ▶ *cumulative acknowledgements*
- ▶ *checksums, timeouts, and sender-initiated retransmission*

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*
- ▶ *cumulative acknowledgements*
- ▶ *checksums, timeouts, and sender-initiated retransmission*

■ Advantages: *simple, minimal state*

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*
- ▶ *cumulative acknowledgements*
- ▶ *checksums, timeouts, and sender-initiated retransmission*

■ Advantages: *simple, minimal state*

- ▶ the sender maintains *two counters* and *one timer*, plus *packet buffer*
- ▶ the receiver maintains *one counter, no packet buffer*

■ Concepts

- ▶ ***sequence numbers***
- ▶ ***sliding window***
- ▶ ***cumulative acknowledgements***
- ▶ ***checksums, timeouts,*** and ***sender-initiated retransmission***

■ Advantages: *simple, minimal state*

- ▶ the sender maintains ***two counters*** and ***one timer***, plus ***packet buffer***
- ▶ the receiver maintains ***one counter, no packet buffer***

■ Disadvantages: *not optimal, not adaptive*

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*
- ▶ *cumulative acknowledgements*
- ▶ *checksums, timeouts, and sender-initiated retransmission*

■ Advantages: *simple, minimal state*

- ▶ the sender maintains *two counters* and *one timer*, plus *packet buffer*
- ▶ the receiver maintains *one counter, no packet buffer*

■ Disadvantages: *not optimal, not adaptive*

- ▶ the sender can fill the window without filling the pipeline

■ Concepts

- ▶ *sequence numbers*
- ▶ *sliding window*
- ▶ *cumulative acknowledgements*
- ▶ *checksums, timeouts, and sender-initiated retransmission*

■ Advantages: *simple, minimal state*

- ▶ the sender maintains *two counters* and *one timer*, plus *packet buffer*
- ▶ the receiver maintains *one counter, no packet buffer*

■ Disadvantages: *not optimal, not adaptive*

- ▶ the sender can fill the window without filling the pipeline
- ▶ the receiver may buffer out-of-order packets...

- What is a good value for W ?

- What is a good value for W ?
 - ▶ W that achieves the *maximum utilization* of the connection

- What is a good value for W ?

- ▶ W that achieves the *maximum utilization* of the connection

ℓ = *stream*

d = 500ms

R = 1Mb/s

W = ?

- What is a good value for W ?

- ▶ W that achieves the *maximum utilization* of the connection

$$\ell = \text{stream}$$

$$d = 500\text{ms}$$

$$R = 1\text{Mb/s}$$

$$W = ?$$

- The problem may seem a bit underspecified. What is the (average) packet size?

$$\ell_{pkt} = 1\text{Kb}$$

$$d = 500\text{ms}$$

$$R = 1\text{Mb/s}$$

$$W = \frac{2d \times R}{\ell_{pkt}} = 1000$$

- The RTT-throughput product ($2d \times R$) is the crucial factor

- The RTT-throughput product ($2d \times R$) is the crucial factor
 - ▶ $W \times \ell_{pkt} \leq 2d \times R$
 - ▶ why $W \times \ell_{pkt} > 2d \times R$ doesn't make much sense?

- The RTT-throughput product ($2d \times R$) is the crucial factor
 - ▶ $W \times \ell_{pkt} \leq 2d \times R$
 - ▶ why $W \times \ell_{pkt} > 2d \times R$ doesn't make much sense?
 - ▶ maximum channel utilization when $W \times \ell_{pkt} = 2d \times R$
 - ▶ $2d \times R$ can be thought of as the *capacity* of a connection

- Let's consider a fully utilized connection

- Let's consider a fully utilized connection

$$\ell_{pkt} = 1Kb$$

$$d = 500ms$$

$$R = 1Mb/s$$

$$W = \frac{R \times d}{\ell_{pkt}} = 1000$$

- Let's consider a fully utilized connection

$$\ell_{pkt} = 1Kb$$

$$d = 500ms$$

$$R = 1Mb/s$$

$$W = \frac{R \times d}{\ell_{pkt}} = 1000$$

- What happens if the first packet (or acknowledgement) is lost?

- Let's consider a fully utilized connection

$$\ell_{pkt} = 1Kb$$

$$d = 500ms$$

$$R = 1Mb/s$$

$$W = \frac{R \times d}{\ell_{pkt}} = 1000$$

- What happens if the first packet (or acknowledgement) is lost?
- Sender retransmits the entire content of its buffers

- Let's consider a fully utilized connection

$$\ell_{pkt} = 1Kb$$

$$d = 500ms$$

$$R = 1Mb/s$$

$$W = \frac{R \times d}{\ell_{pkt}} = 1000$$

- What happens if the first packet (or acknowledgement) is lost?
- Sender retransmits the entire content of its buffers
 - ▶ $W \times \ell_{pkt} = 2d \times R = 1Mb$
 - ▶ retransmitting 1Mb to recover 1Kb worth of data isn't exactly the best solution. Not to mention congestions...

- Let's consider a fully utilized connection

$$\ell_{pkt} = 1Kb$$

$$d = 500ms$$

$$R = 1Mb/s$$

$$W = \frac{R \times d}{\ell_{pkt}} = 1000$$

- What happens if the first packet (or acknowledgement) is lost?
- Sender retransmits the entire content of its buffers
 - ▶ $W \times \ell_{pkt} = 2d \times R = 1Mb$
 - ▶ retransmitting 1Mb to recover 1Kb worth of data isn't exactly the best solution. Not to mention congestions...
- Is there a better way to deal with retransmissions?

- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted

- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted
 - ▶ sender maintains a vector of acknowledgement flags

- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted
 - ▶ sender maintains a vector of acknowledgement flags
 - ▶ receiver maintains a vector of acknowledged flags

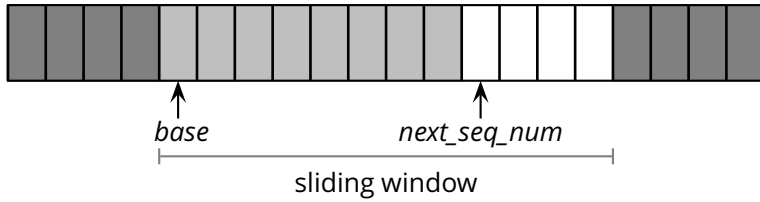
- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted
 - ▶ sender maintains a vector of acknowledgement flags
 - ▶ receiver maintains a vector of acknowledged flags
 - ▶ in fact, receiver maintains a buffer of out-of-order packets

- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted
 - ▶ sender maintains a vector of acknowledgement flags
 - ▶ receiver maintains a vector of acknowledged flags
 - ▶ in fact, receiver maintains a buffer of out-of-order packets
 - ▶ sender maintains a timer for each pending packet

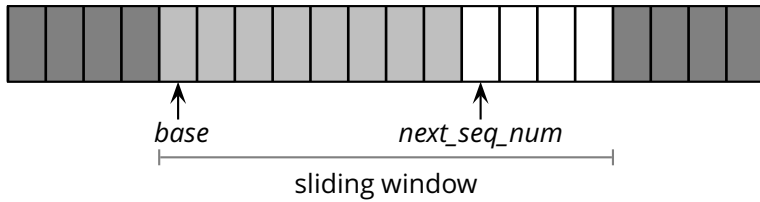
- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted
 - ▶ sender maintains a vector of acknowledgement flags
 - ▶ receiver maintains a vector of acknowledged flags
 - ▶ in fact, receiver maintains a buffer of out-of-order packets
 - ▶ sender maintains a timer for each pending packet
 - ▶ sender resends a packet when its timer expires

- **Idea:** have the sender retransmit only those packets that it suspects were lost or corrupted
 - ▶ sender maintains a vector of acknowledgement flags
 - ▶ receiver maintains a vector of acknowledged flags
 - ▶ in fact, receiver maintains a buffer of out-of-order packets
 - ▶ sender maintains a timer for each pending packet
 - ▶ sender resends a packet when its timer expires
 - ▶ sender slides the window when the lowest pending sequence number is acknowledged

Selective Repeat: Sender

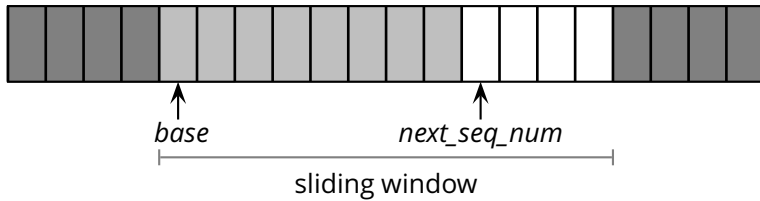


Selective Repeat: Sender



■ $r_send(pkt_1)$

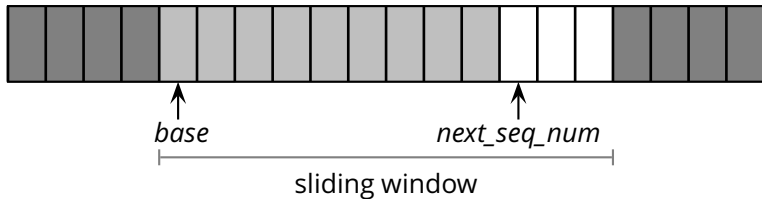
Selective Repeat: Sender



■ `r_send(pkt1)`

- ▶ `u_send([pkt1, next_seq_num])`
- ▶ `start_timer(next_seq_num)`

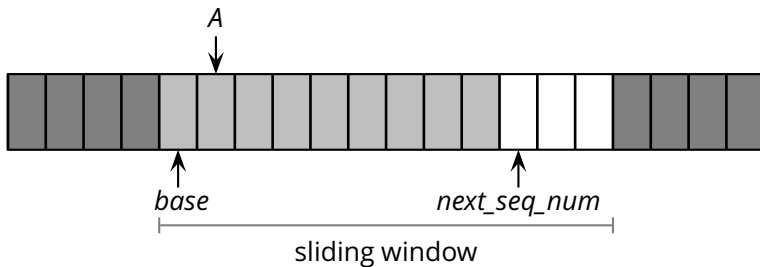
Selective Repeat: Sender



■ `r_send(pkt1)`

- ▶ `u_send([pkt1, next_seq_num])`
- ▶ `start_timer(next_seq_num)`
- ▶ `next_seq_num = next_seq_num + 1`

Selective Repeat: Sender

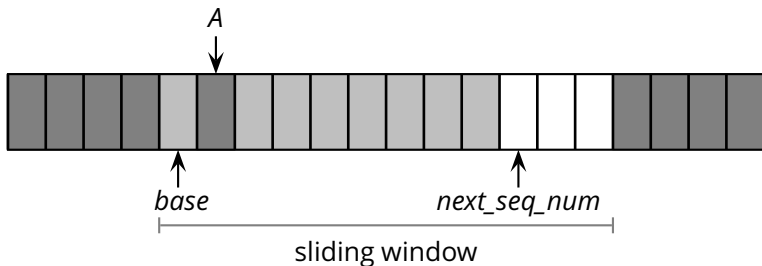


■ `r_send(pkt1)`

- ▶ `u_send([pkt1, next_seq_num])`
- ▶ `start_timer(next_seq_num)`
- ▶ `next_seq_num = next_seq_num + 1`

■ `u_rcv([ACK, A])`

Selective Repeat: Sender



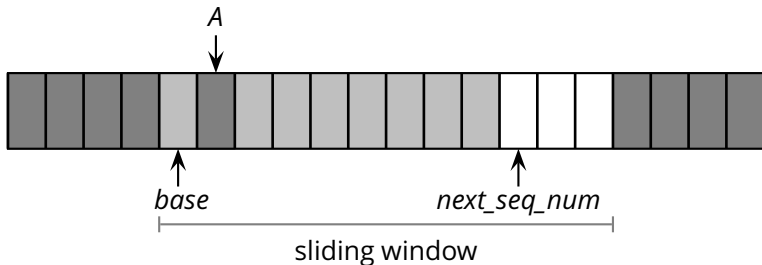
■ `r_send(pkt1)`

- ▶ `u_send([pkt1, next_seq_num])`
- ▶ `start_timer(next_seq_num)`
- ▶ `next_seq_num = next_seq_num + 1`

■ `u_rcv([ACK, A])`

- ▶ `acks[A] = 1` *// remember that A was ACK'd*

Selective Repeat: Sender



■ `r_send(pkt1)`

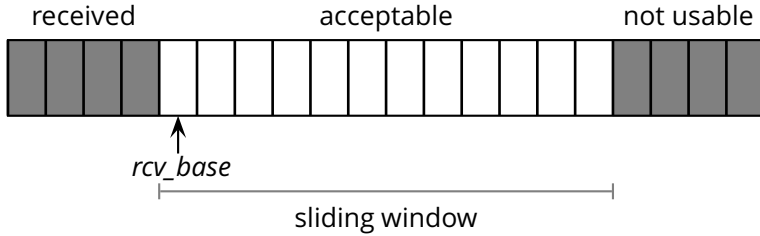
- ▶ `u_send([pkt1, next_seq_num])`
- ▶ `start_timer(next_seq_num)`
- ▶ `next_seq_num = next_seq_num + 1`

■ `u_rcv([ACK, A])`

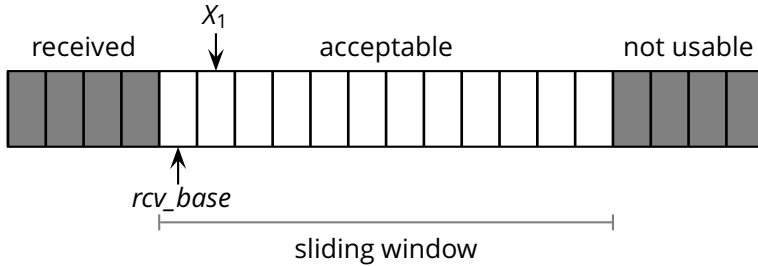
- ▶ `acks[A] = 1` // remember that A was ACK'd
- ▶ acknowledgements are no longer "cumulative"

Selective Repeat: Receiver

Selective Repeat: Receiver

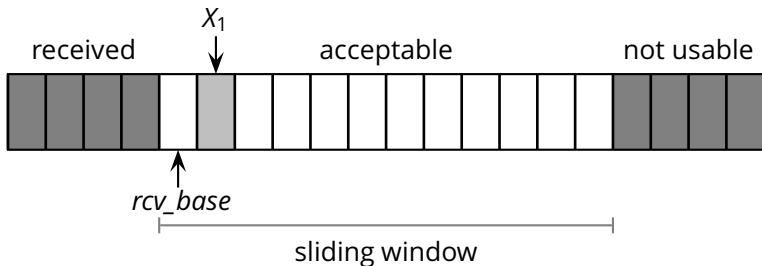


Selective Repeat: Receiver



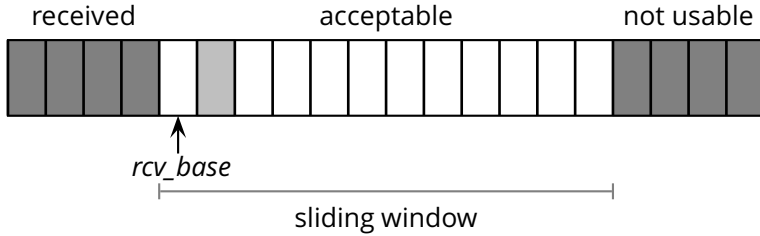
- $u_rcv([pkt_1, X_1])$ and $rcv_base \leq X_1 < rcv_base + W$

Selective Repeat: Receiver

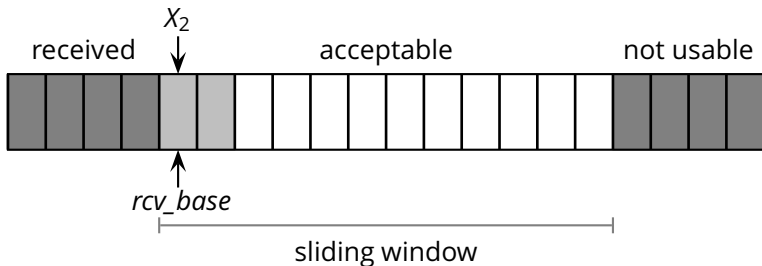


- $u_recv([pkt_1, X_1])$ and $rcv_base \leq X_1 < rcv_base + W$
 - ▶ $buffer[X_1] = pkt_1$
 - ▶ $u_send([ACK, X_1]^*)$ // no longer a "cumulative" ACK

Selective Repeat: Receiver

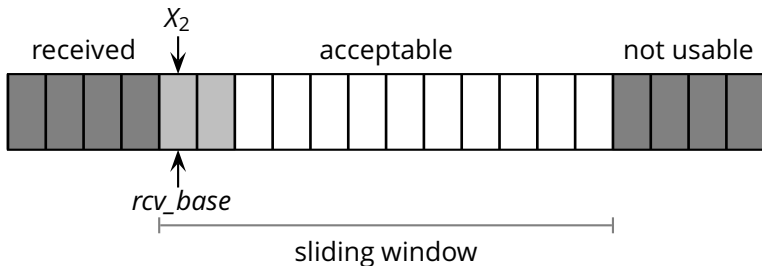


Selective Repeat: Receiver



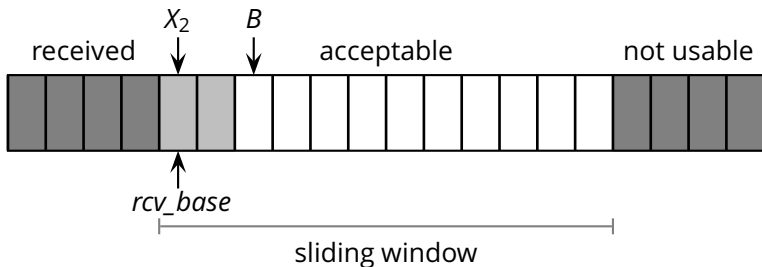
- $u_recv([pkt_2, X_2])$ and $rcv_base \leq X_2 < rcv_base + W$
 - ▶ $buffer[X_2] = pkt_2$
 - ▶ $u_send([ACK, X_2]^*)$

Selective Repeat: Receiver



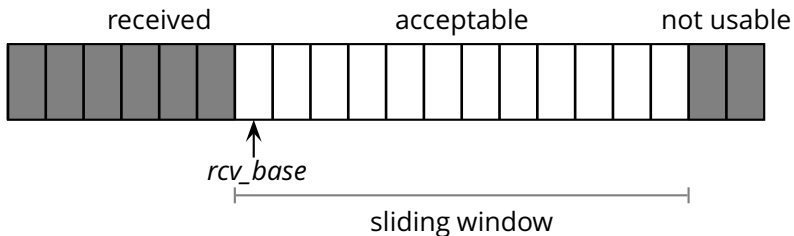
- $u_recv([pkt_2, X_2])$ and $rcv_base \leq X_2 < rcv_base + W$
 - ▶ $buffer[X_2] = pkt_2$
 - ▶ $u_send([ACK, X_2]^*)$
 - ▶ **if** $X_2 == rcv_base$:

Selective Repeat: Receiver



- $u_recv([pkt_2, X_2])$ and $rcv_base \leq X_2 < rcv_base + W$
 - ▶ $buffer[X_2] = pkt_2$
 - ▶ $u_send([ACK, X_2]^*)$
 - ▶ **if** $X_2 == rcv_base$:
 - $B = first_missing_seq_num()$
 - foreach** i in $rcv_base \dots B - 1$:
 - $r_recv(buffer[i])$

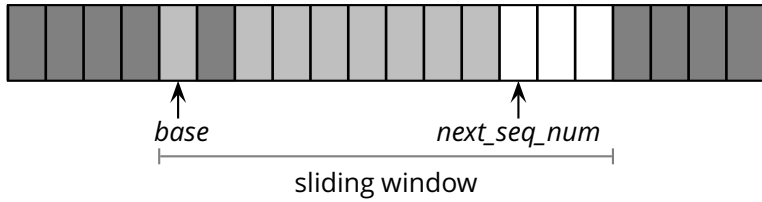
Selective Repeat: Receiver



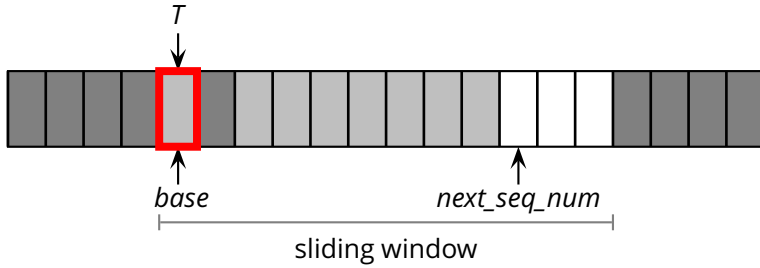
- $u_recv([pkt_2, X_2])$ and $rcv_base \leq X_2 < rcv_base + W$
 - ▶ $buffer[X_2] = pkt_2$
 - ▶ $u_send([ACK, X_2]^*)$
 - ▶ **if** $X_2 == rcv_base$:
 - $B = first_missing_seq_num()$
 - foreach** i in $rcv_base \dots B - 1$:
 - $r_recv(buffer[i])$
 - $rcv_base = B$

Selective Repeat: Sender

Selective Repeat: Sender

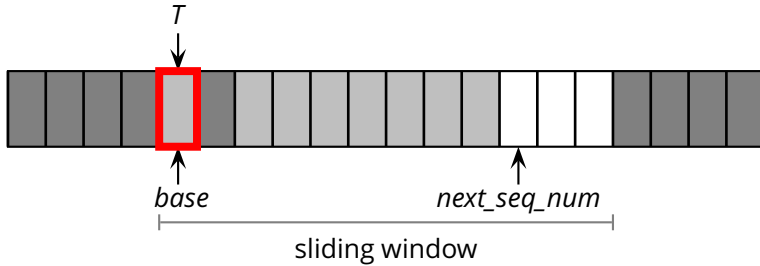


Selective Repeat: Sender



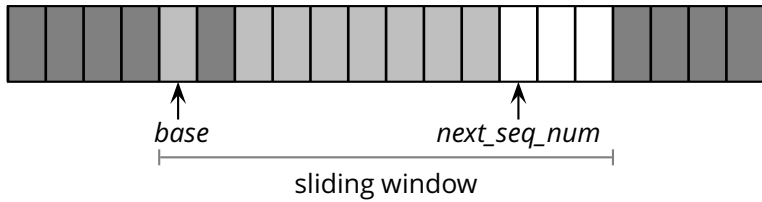
- Timeout for sequence number T

Selective Repeat: Sender

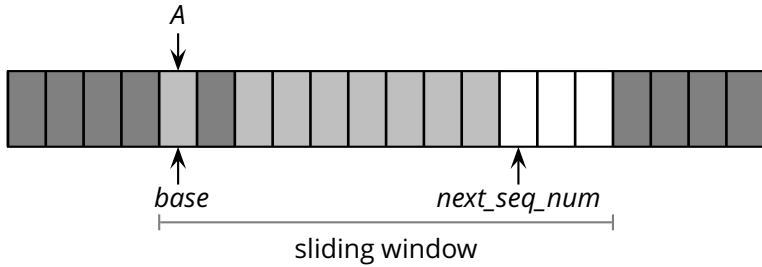


- Timeout for sequence number T
 - ▶ $u_send([pkt[T], T]^*)$

Selective Repeat: Sender

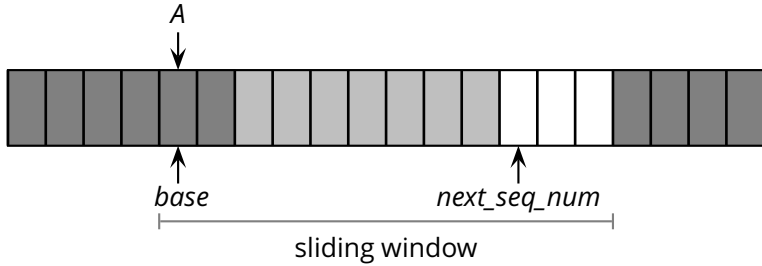


Selective Repeat: Sender



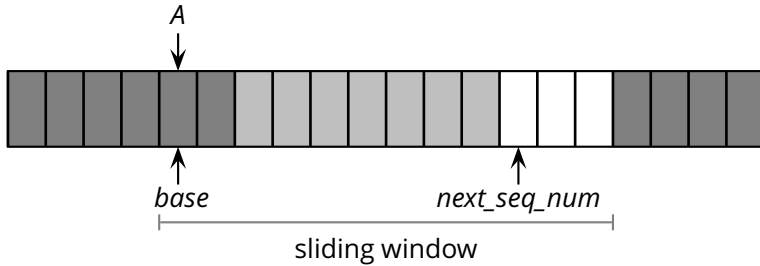
■ `u_rcv([ACK,A])`

Selective Repeat: Sender



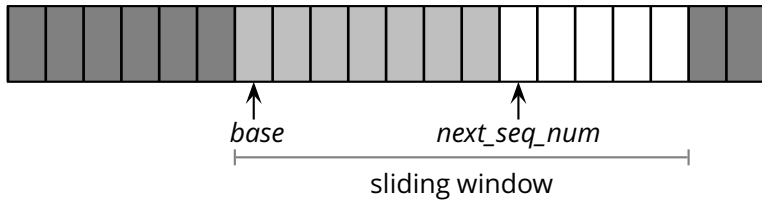
- `u_rcv([ACK,A])`
 - ▶ `acks[A] = 1`

Selective Repeat: Sender



- `u_rcv([ACK,A])`
 - ▶ `acks[A] = 1`
 - ▶ **if** `A == base`:

Selective Repeat: Sender



- `u_rcv([ACK,A])`
 - ▶ `acks[A] = 1`
 - ▶ **if** `A == base`:
 - `base = first_missing_ack_num()`