# Reliable Data Transfer II

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

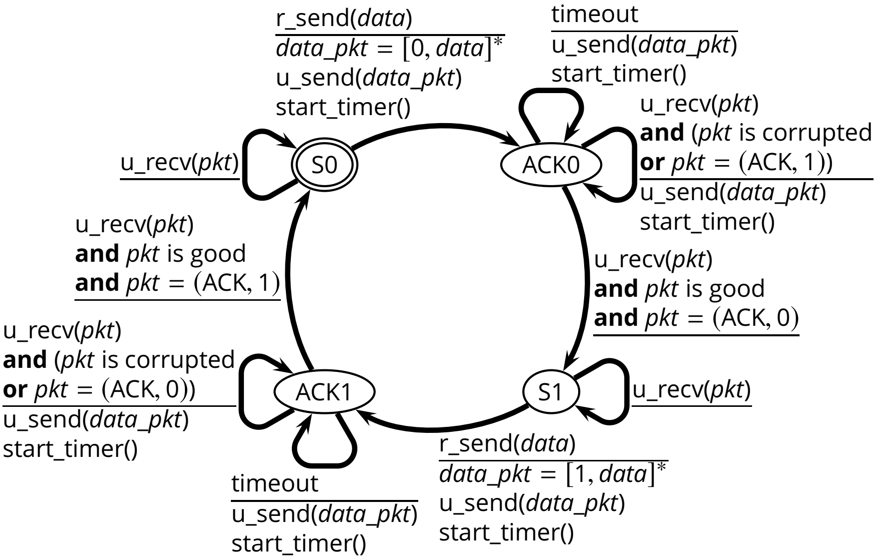
March 25, 2020

# Outline

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

# Back to Reliable Data Tranfer

# Back to Reliable Data Tranfer



r_send(*data*)
*data_pkt* $= [0, \mathit{data}]^*$
u_send(*data_pkt*)
start_timer()

timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* $= (\mathrm{ACK}, 1)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* $= (\mathrm{ACK}, 1)$

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* $= (\mathrm{ACK}, 0)$

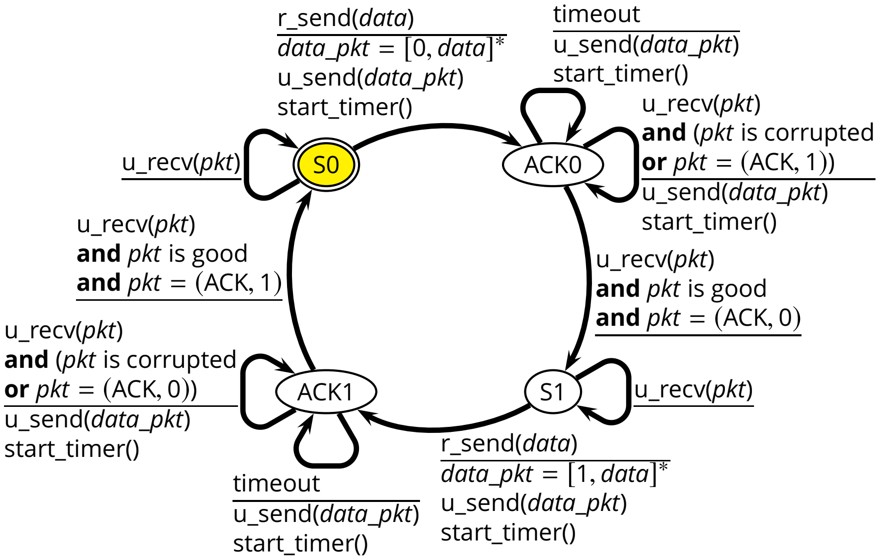
u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* $= (\mathrm{ACK}, 0)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

timeout
u_send(*data_pkt*)
start_timer()

r_send(*data*)
*data_pkt* $= [1, \mathit{data}]^*$
u_send(*data_pkt*)
start_timer()

# Back to Reliable Data Tranfer



r_send(*data*)

---

$data\_pkt = [0, data]^*$
u_send(*data_pkt*)
start_timer()

timeout

---

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

---

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* = (ACK, 1))

---

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* = (ACK, 1)

---

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* = (ACK, 0)

---

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* = (ACK, 0))

---

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

---

timeout

---

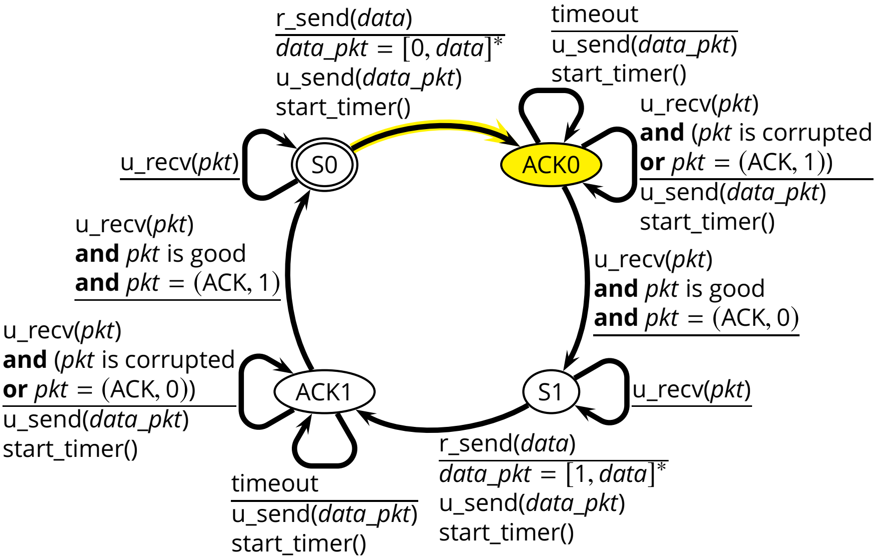
u_send(*data_pkt*)
start_timer()

r_send(*data*)

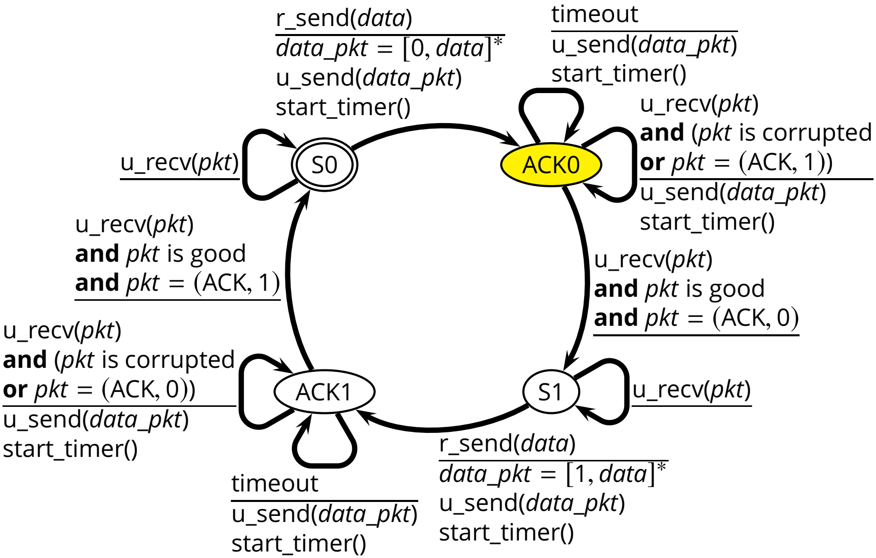
---

$data\_pkt = [1, data]^*$
u_send(*data_pkt*)
start_timer()

# Back to Reliable Data Tranfer



# Back to Reliable Data Tranfer



r_send(*data*)
$data\_pkt = [0, data]^*$
u_send(*data_pkt*)
start_timer()

timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\mathrm{ACK}, 1)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\mathrm{ACK}, 0)$

u_recv(*pkt*)

r_send(*data*)
$data\_pkt = [1, data]^*$
u_send(*data_pkt*)
start_timer()

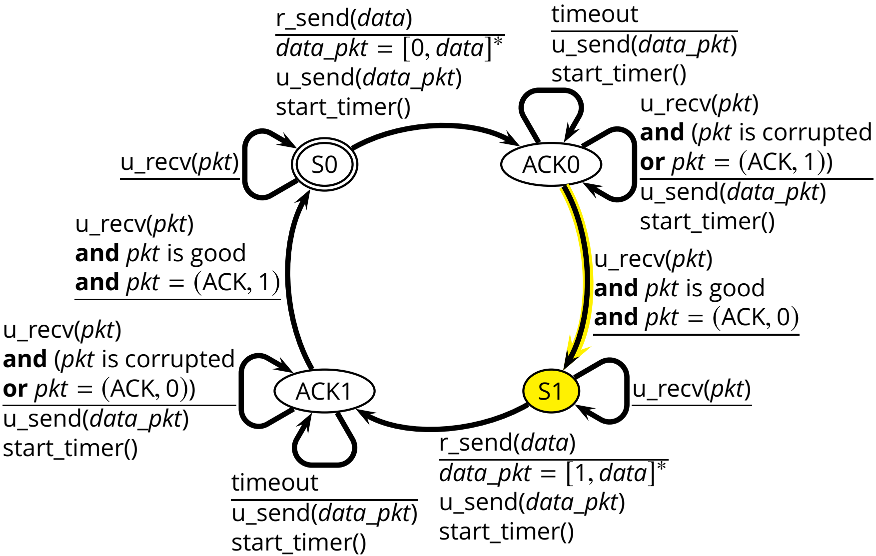
timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\mathrm{ACK}, 0)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\mathrm{ACK}, 1)$

u_recv(*pkt*)

# Back to Reliable Data Tranfer



r_send(*data*)
$data\_pkt = [0, data]^*$
u_send(*data_pkt*)
start_timer()

timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\mathrm{ACK}, 1)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\mathrm{ACK}, 0)$

u_recv(*pkt*)

r_send(*data*)
$data\_pkt = [1, data]^*$
u_send(*data_pkt*)
start_timer()

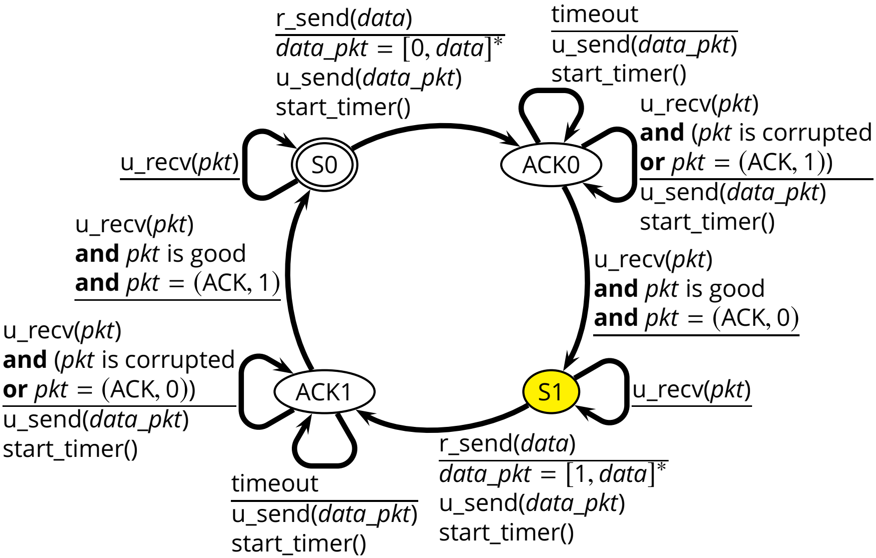
timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\mathrm{ACK}, 0)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\mathrm{ACK}, 1)$

u_recv(*pkt*)

# Back to Reliable Data Tranfer



r_send(*data*)
$\mathit{data\_pkt} = [0, \mathit{data}]^*$
u_send(*data_pkt*)
start_timer()

timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $\mathit{pkt} = (\mathrm{ACK}, 1)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

u_recv(*pkt*)
**and** *pkt* is good
**and** $\mathit{pkt} = (\mathrm{ACK}, 1)$

u_recv(*pkt*)
**and** *pkt* is good
**and** $\mathit{pkt} = (\mathrm{ACK}, 0)$

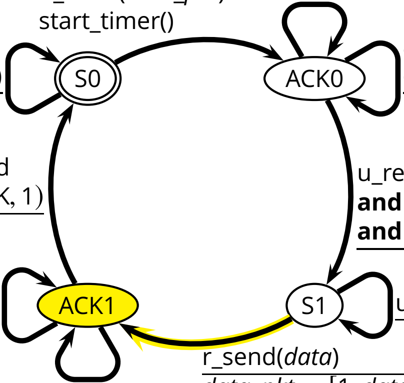
u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $\mathit{pkt} = (\mathrm{ACK}, 0)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

timeout
u_send(*data_pkt*)
start_timer()

r_send(*data*)
$\mathit{data\_pkt} = [1, \mathit{data}]^*$
u_send(*data_pkt*)
start_timer()

# Back to Reliable Data Tranfer



r_send(*data*)

---

*data_pkt* $= [0, \textit{data}]^*$
u_send(*data_pkt*)
start_timer()

timeout

---

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* $= (\text{ACK}, 1)$)

---

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

---

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* $= (\text{ACK}, 1)$

---

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* $= (\text{ACK}, 0)$

---

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* $= (\text{ACK}, 0)$)

---

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

---

timeout

---

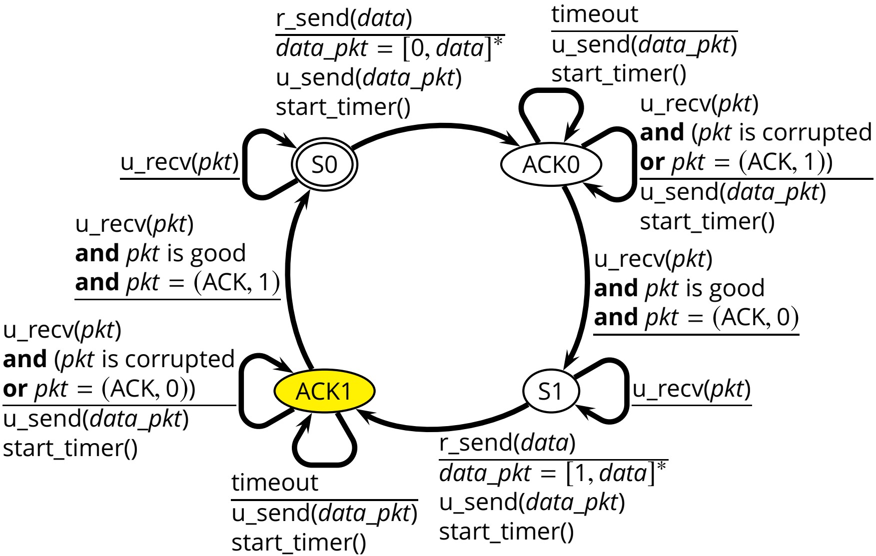
u_send(*data_pkt*)
start_timer()

r_send(*data*)

---

*data_pkt* $= [1, \textit{data}]^*$
u_send(*data_pkt*)
start_timer()

# Back to Reliable Data Tranfer



r_send(*data*)
$data\_pkt = [0, data]^*$
u_send(*data_pkt*)
start_timer()

timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\text{ACK}, 1)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\text{ACK}, 0)$

u_recv(*pkt*)

r_send(*data*)
$data\_pkt = [1, data]^*$
u_send(*data_pkt*)
start_timer()

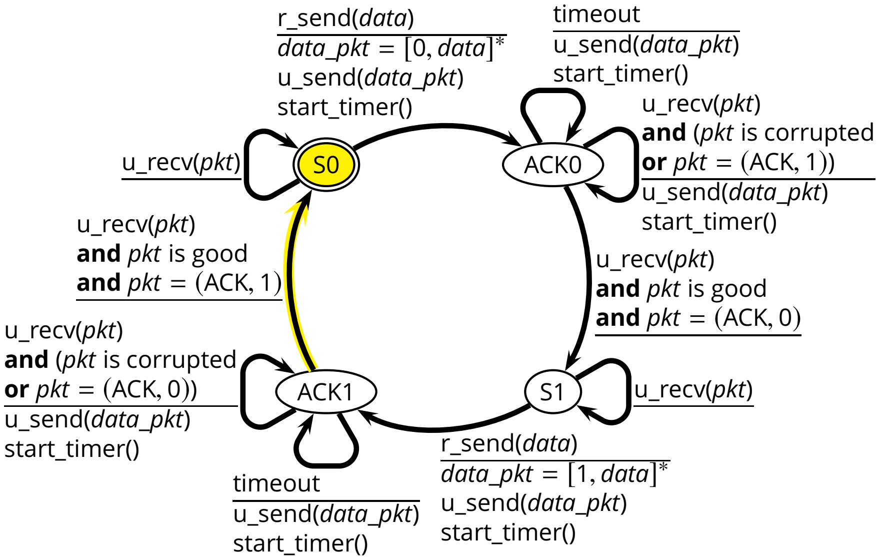
timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\text{ACK}, 0)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\text{ACK}, 1)$

u_recv(*pkt*)

# Back to Reliable Data Tranfer



r_send(*data*)
$data\_pkt = [0, data]^*$
u_send(*data_pkt*)
start_timer()

timeout
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\text{ACK}, 1)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\text{ACK}, 1)$

u_recv(*pkt*)
**and** *pkt* is good
**and** $pkt = (\text{ACK}, 0)$

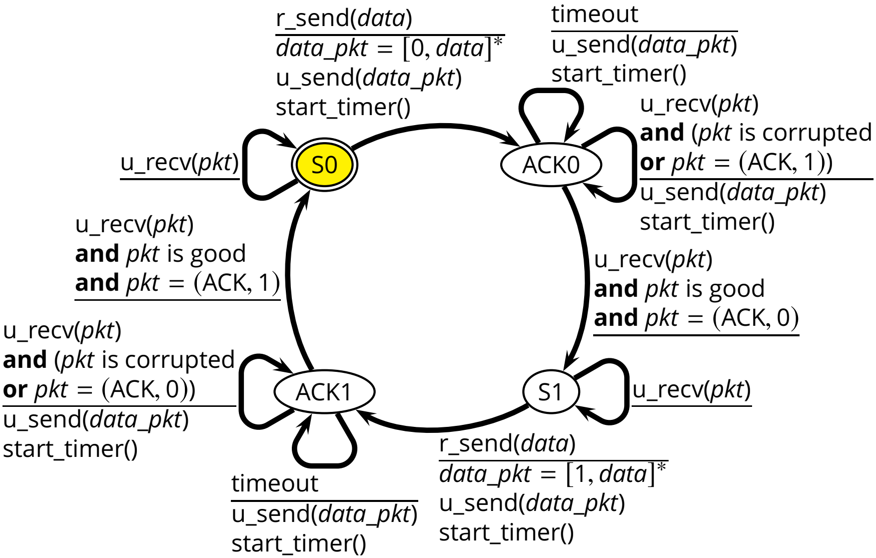
u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** $pkt = (\text{ACK}, 0)$)
u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

timeout
u_send(*data_pkt*)
start_timer()

r_send(*data*)
$data\_pkt = [1, data]^*$
u_send(*data_pkt*)
start_timer()

# Back to Reliable Data Tranfer



r_send(*data*)

$data\_pkt = [0, data]^*$
u_send(*data_pkt*)
start_timer()

timeout

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* = (ACK, 1))

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* = (ACK, 1)

u_recv(*pkt*)
**and** *pkt* is good
**and** *pkt* = (ACK, 0)

u_recv(*pkt*)
**and** (*pkt* is corrupted
**or** *pkt* = (ACK, 0))

u_send(*data_pkt*)
start_timer()

u_recv(*pkt*)

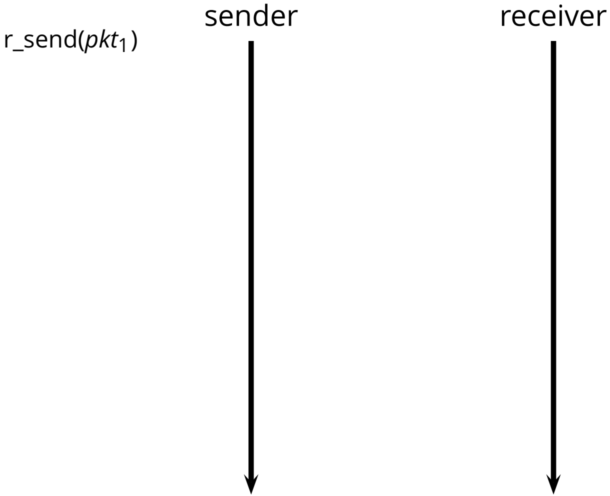
r_send(*data*)

$data\_pkt = [1, data]^*$
u_send(*data_pkt*)
start_timer()

timeout

u_send(*data_pkt*)
start_timer()

# Network Usage

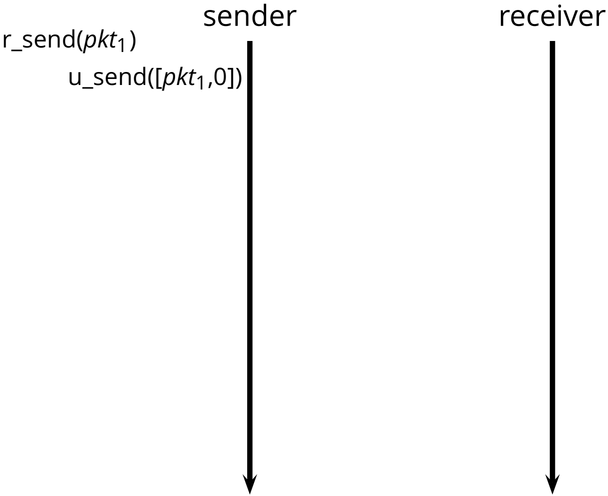


sender

receiver

r_send($pkt_1$)

# Network Usage



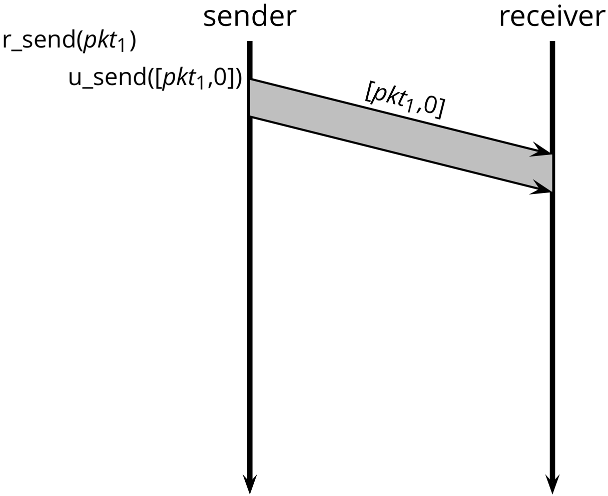
sender

receiver

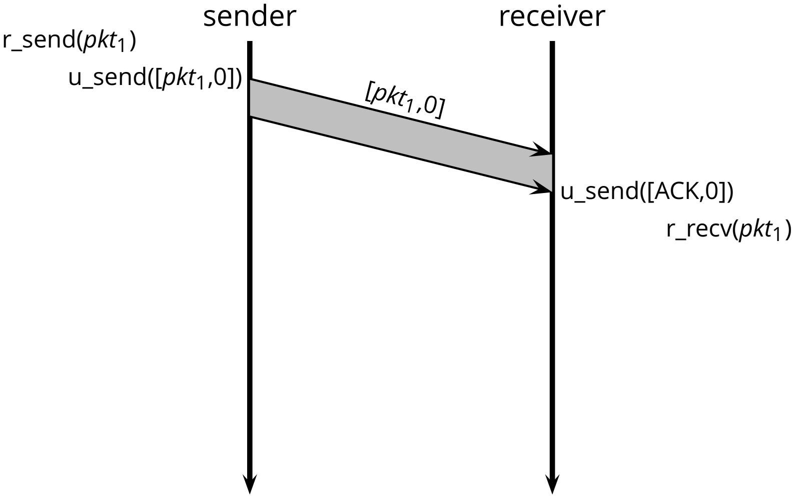
r_send($pkt_1$)

u_send([$pkt_1$,0])

# Network Usage



# Network Usage



sender

receiver

r_send($pkt_1$)

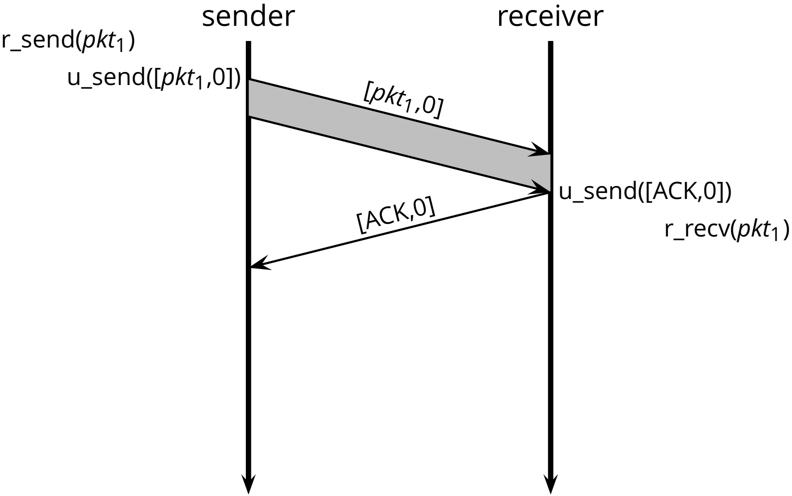
u_send([$pkt_1$,0])

[$pkt_1$,0]

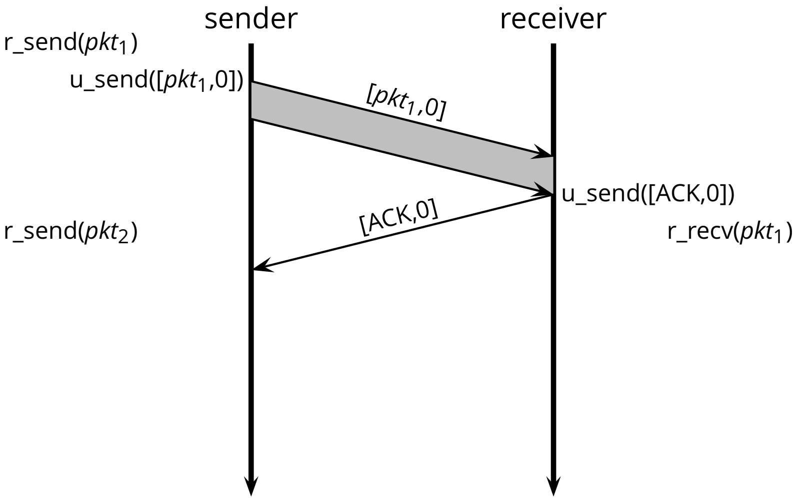
u_send([ACK,0])

r_recv($pkt_1$)

# Network Usage



# Network Usage



sender

receiver

r_send($pkt_1$)

u_send([$pkt_1$,0])

[$pkt_1$,0]

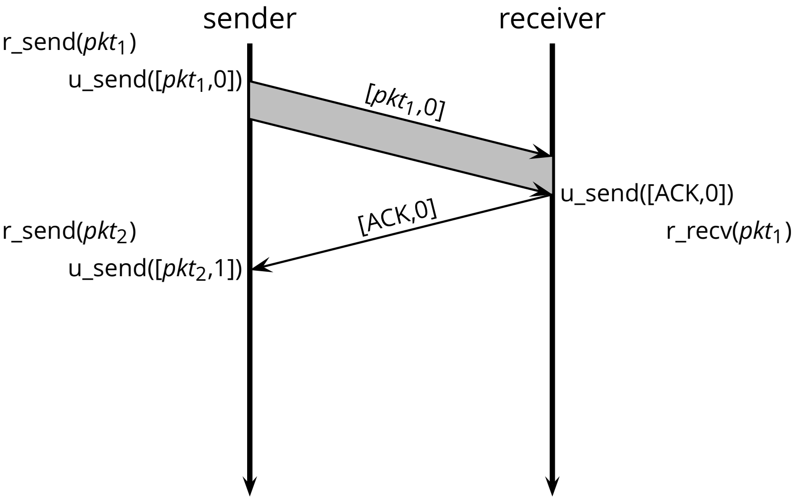
u_send([ACK,0])

[ACK,0]

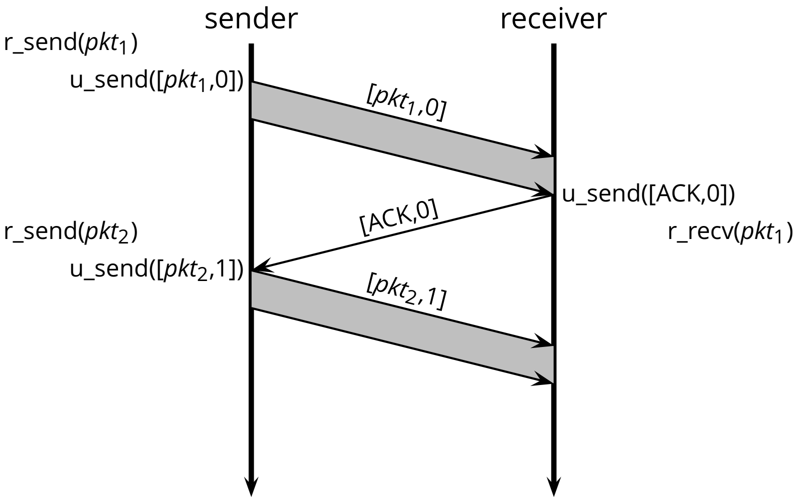
r_send($pkt_2$)

r_recv($pkt_1$)

# Network Usage



# Network Usage



sender

receiver

r_send($pkt_1$)

u_send([$pkt_1$,0])

[$pkt_1$,0]

u_send([ACK,0])

[ACK,0]

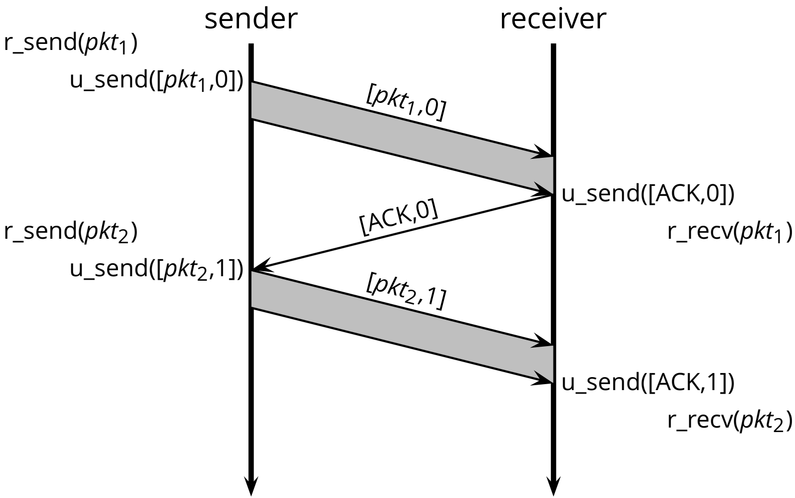
r_send($pkt_2$)

r_recv($pkt_1$)

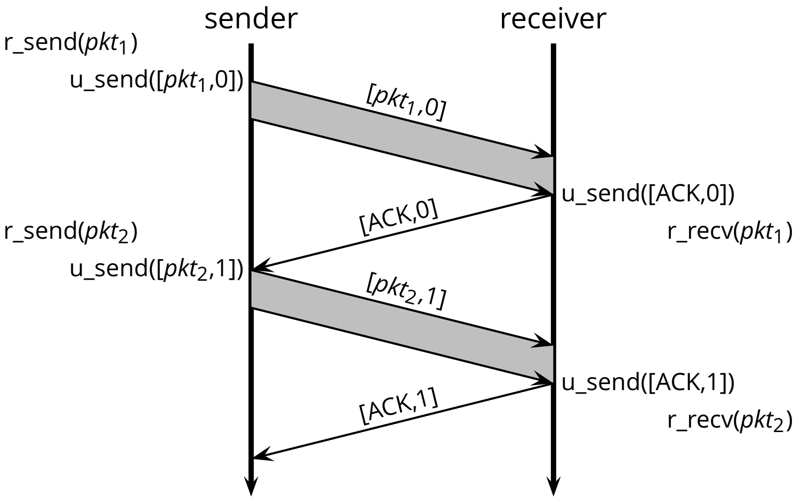
u_send([$pkt_2$,1])

[$pkt_2$,1]

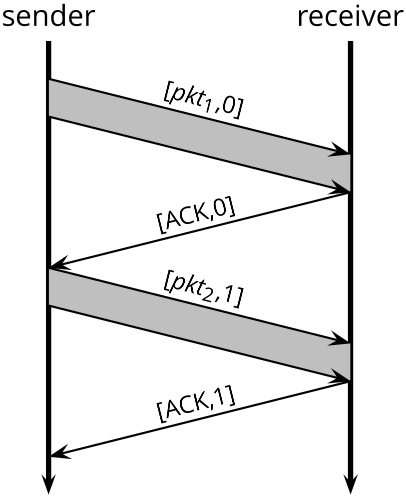
# Network Usage



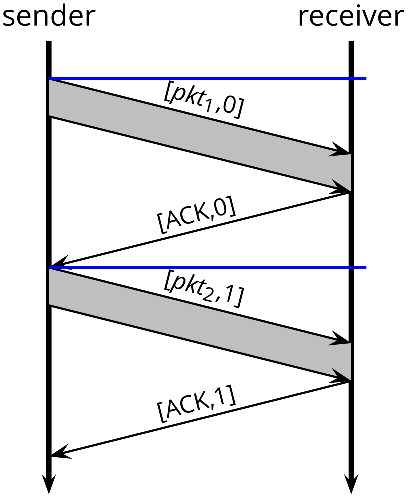
# Network Usage



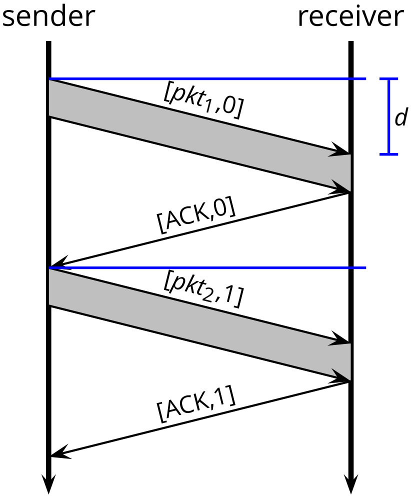
# Network Usage



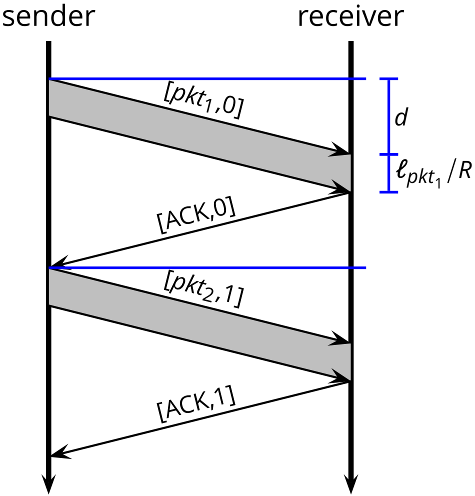
# Network Usage



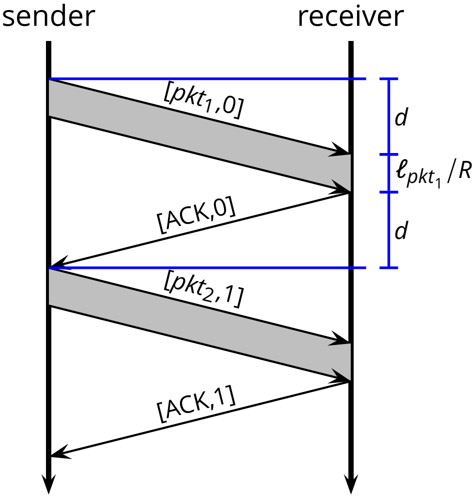
# Network Usage



# Network Usage



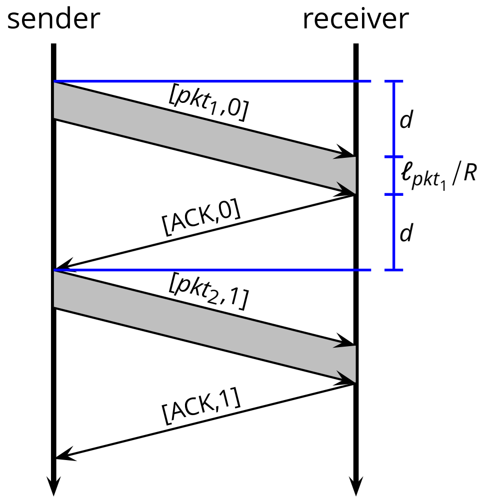
# Network Usage



# Network Usage

*utilization factor*

$$U = \frac{\ell_{pkt}/R}{2d + \ell_{pkt}/R}$$

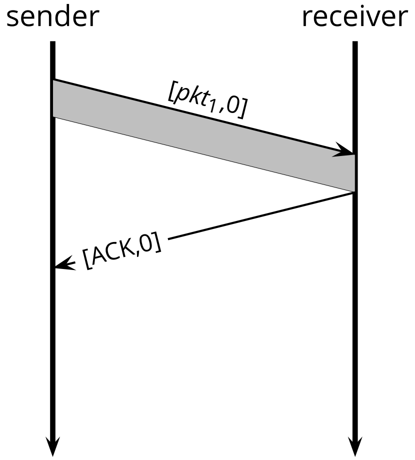


# Improving Network Usage

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

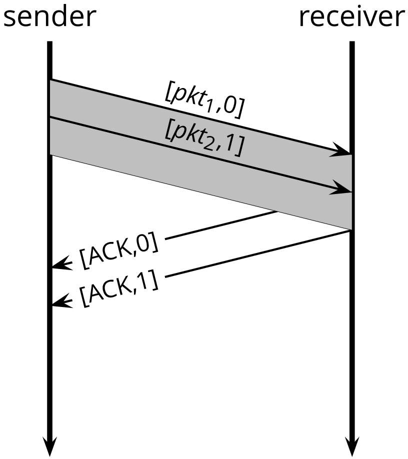
# Improving Network Usage

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



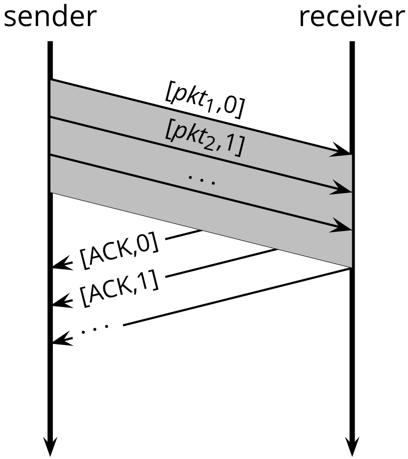
# Improving Network Usage

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



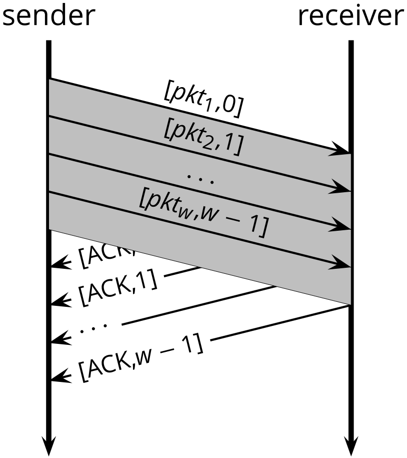
# Improving Network Usage

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



# Improving Network Usage

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



# Go-Back-N

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

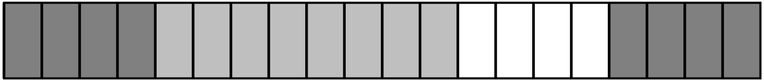
# Go-Back-N

- **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

# Go-Back-N

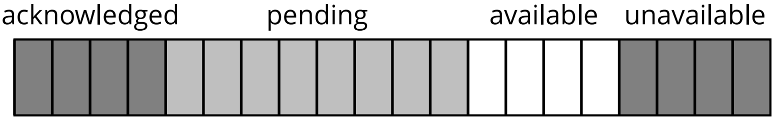
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement

- 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



# Go-Back-N

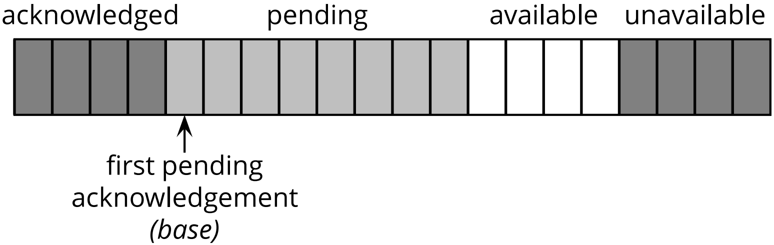
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to *W* unacknowledged packets in the pipeline
  - the sender’s state machine gets very complex
  - we represent the sender’s state with its queue of acknowledgements



# Go-Back-N

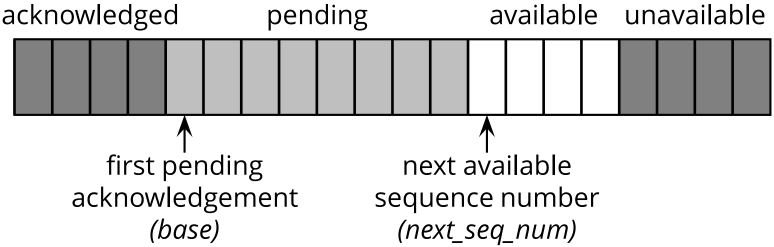
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement

- 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



# Go-Back-N

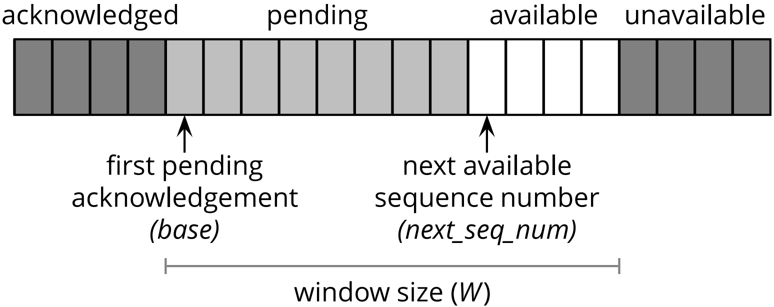
- **Idea:** the sender transmits multiple packets without waiting for an acknowledgement
- Sender has up to *W* unacknowledged packets in the pipeline
  - the sender's state machine gets very complex
  - we represent the sender's state with its queue of acknowledgements



# Go-Back-N

- **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



acknowledged

pending

available

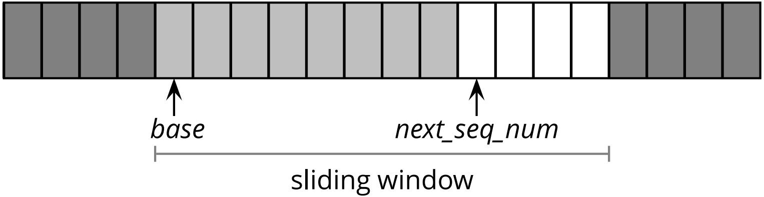
unavailable

first pending acknowledgement *(base)*

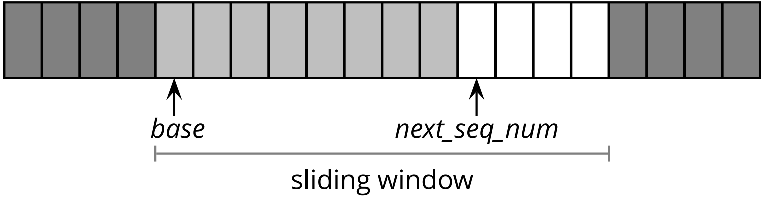
next available sequence number *(next_seq_num)*

window size (*W*)

# Sliding Window Protocol: Sender

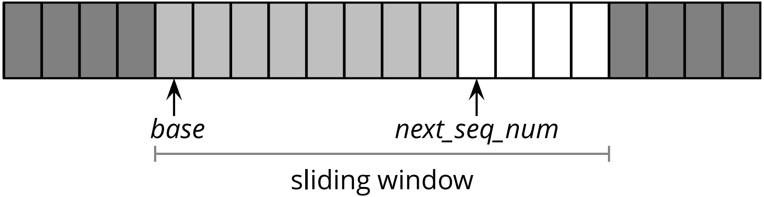


# Sliding Window Protocol: Sender



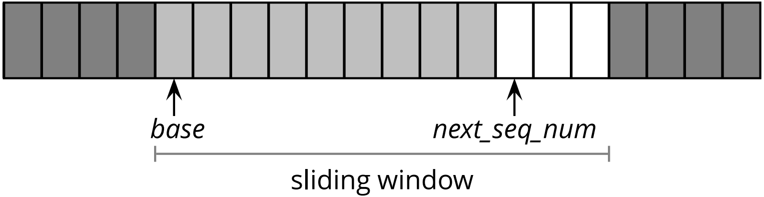
- r_send($pkt_1$)

# Sliding Window Protocol: Sender



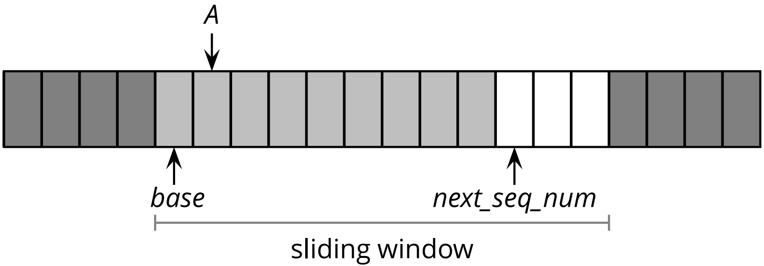
- r_send($pkt_1$)
  - u_send([$pkt_1$,*next_seq_num*])

# Sliding Window Protocol: Sender



- r_send($pkt_1$)
  - u_send([$pkt_1$,$next\_seq\_num$])
  - $next\_seq\_num = next\_seq\_num + 1$

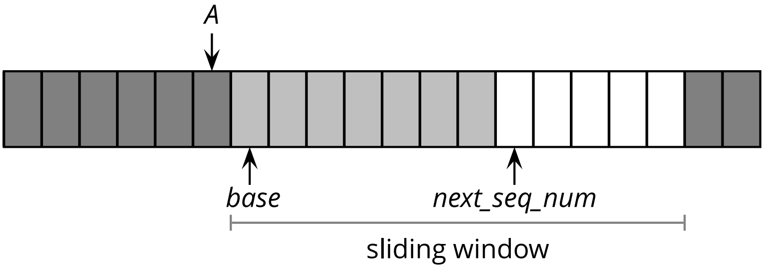
# Sliding Window Protocol: Sender



- r_send($pkt_1$)
  - u_send([$pkt_1$,$next\_seq\_num$])
  - $next\_seq\_num = next\_seq\_num + 1$

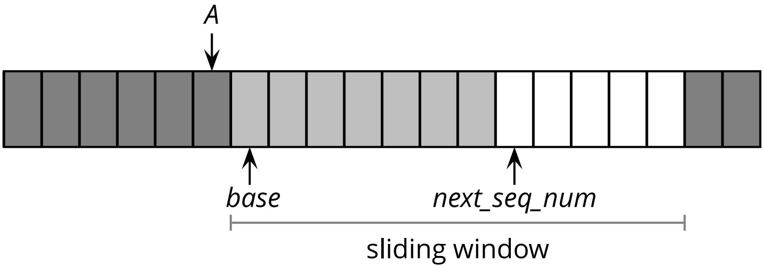
- u_recv([ACK,$A$])

# Sliding Window Protocol: Sender



- r_send($pkt_1$)
  - u_send([$pkt_1$,$next\_seq\_num$])
  - $next\_seq\_num = next\_seq\_num + 1$
- u_recv([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_recv([ACK,$A$])
  - $base = A + 1$
  - notice that acknewledgements 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

- $$\frac{\mathit{init}}{\begin{array}{l} \mathit{base} = 1 \\ \mathit{next\_seq\_num} = 1 \end{array}}$$

# 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_recv(*pkt*) **and** *pkt* is corrupted

# Sliding Window Protocol: Sender

- u_recv(*pkt*) **and** *pkt* is corrupted
- u_recv(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_recv(*pkt*) **and** *pkt* is corrupted

- u_recv(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

- $$\frac{init}{\begin{array}{l} expected\_seq\_num = 1 \\ ackpkt = [ACK, 0]^{*} \end{array}}$$

# Sliding Window Protocol: Receiver

- $$\frac{\mathit{init}}{\begin{array}{l} \mathit{expected\_seq\_num} \; = \; 1 \\ \mathit{ackpkt} \; = \; [\mathit{ACK}, 0]^{*} \end{array}}$$

- $$\frac{\begin{array}{l} \mathrm{u\_recv}([\mathit{data}, \mathit{seq\_num}]) \ \textbf{and} \ \mathrm{good} \\ \textbf{and} \ \mathit{seq\_num} = \mathit{expected\_seq\_num} \end{array}}{\begin{array}{l} \mathrm{r\_recv}(\mathit{data}) \\ \mathit{ackpkt} \; = \; [\mathit{ACK}, \mathit{expected\_seq\_num}]^{*} \\ \mathit{expected\_seq\_num} \; = \; \mathit{expected\_seq\_num} + 1 \\ \mathrm{u\_send}(\mathit{ackpkt}) \end{array}}$$

# Sliding Window Protocol: Receiver

- $$\frac{\textit{init}}{\begin{array}{l} \textit{expected\_seq\_num} = 1 \\ \textit{ackpkt} = [\textit{ACK}, 0]^{*} \end{array}}$$

- $$\frac{\begin{array}{l} \text{u\_recv}([\textit{data}, \textit{seq\_num}]) \ \textbf{and} \ \text{good} \\ \textbf{and} \ \textit{seq\_num} = \textit{expected\_seq\_num} \end{array}}{\begin{array}{l} \text{r\_recv}(\textit{data}) \\ \textit{ackpkt} = [\textit{ACK}, \textit{expected\_seq\_num}]^{*} \\ \textit{expected\_seq\_num} = \textit{expected\_seq\_num} + 1 \\ \text{u\_send}(\textit{ackpkt}) \end{array}}$$

- $$\frac{\begin{array}{l} \text{u\_recv}([\textit{data}, \textit{seq\_num}]) \\ \textbf{and} \ (\text{corrupted} \ \textbf{or} \ \textit{seq\_num} \neq \textit{expected\_seq\_num}) \end{array}}{\text{u\_send}(\textit{ackpkt})}$$

# Comments

- Concepts

# Comments

- Concepts
  - ***sequence numbers***

# Comments

- Concepts
  - ***sequence numbers***
  - ***sliding window***

# Comments

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

# Comments

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

# Comments

- Concepts
  - ***sequence numbers***
  - ***sliding window***
  - ***cumulative acknowledgements***
  - ***checksums***, ***timeouts***, and ***sender-initiated retransmission***
- Advantages: *simple, minimal state*

# Comments

- 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***

# Comments

- 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*

# Comments

- 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

# Comments

- 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. . .

# Performance Analysis

- What is a good value for $W$?

# Performance Analysis

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

# Performance Analysis

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

$$\begin{aligned} \ell &= stream \\ d &= 500ms \\ R &= 1Mb/s \\ W &= ? \end{aligned}$$

# Performance Analysis

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

$$\begin{aligned} \ell &= stream \\ d &= 500ms \\ R &= 1Mb/s \\ W &= ? \end{aligned}$$

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

$$\begin{aligned} \ell_{pkt} &= 1Kb \\ d &= 500ms \\ R &= 1Mb/s \\ W &= \frac{2d\times R}{\ell_{pkt}} = 1000 \end{aligned}$$

# Performance Analysis

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

# Performance Analysis

- 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?

# Performance Analysis

- 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

# Problems with Go-Back-N

- Let’s consider a fully utilized connection

# Problems with Go-Back-N

- Let’s consider a fully utilized connection

$$\begin{aligned}
\ell_{pkt} &= 1Kb \\
d &= 500ms \\
R &= 1Mb/s \\
W &= \frac{R \times d}{\ell_{pkt}} = 1000
\end{aligned}$$

# Problems with Go-Back-N

- Let's consider a fully utilized connection

$$\begin{aligned} \ell_{pkt} &= 1Kb \\ d &= 500ms \\ R &= 1Mb/s \\ W &= \frac{R \times d}{\ell_{pkt}} = 1000 \end{aligned}$$

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

# Problems with Go-Back-N

- Let’s consider a fully utilized connection

$$\begin{aligned}
\ell_{pkt} &= 1Kb \\
d &= 500ms \\
R &= 1Mb/s \\
W &= \frac{R\times d}{\ell_{pkt}} = 1000
\end{aligned}$$

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

# Problems with Go-Back-N

- Let’s consider a fully utilized connection

$$\begin{aligned} \ell_{pkt} &= 1Kb \\ d &= 500ms \\ R &= 1Mb/s \\ W &= \frac{R \times d}{\ell_{pkt}} = 1000 \end{aligned}$$

- 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 conjestions...

# Problems with Go-Back-N

- Let’s consider a fully utilized connection

$$\begin{aligned} \ell_{pkt} &= 1Kb \\ d &= 500ms \\ R &= 1Mb/s \\ W &= \frac{R \times d}{\ell_{pkt}} = 1000 \end{aligned}$$

- 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 conjestions. . .
- Is there a better way to deal with retransmissions?

# Selective Repeat

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

# Selective Repeat

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

# Selective Repeat

- **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 falgs

# Selective Repeat

- **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 falgs
    - in fact, receiver maintains a buffer of out-of-order packets

# Selective Repeat

- **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 falgs
  - in fact, receiver maintains a buffer of out-of-order packets
  - sender maintains a timer for each pending packet

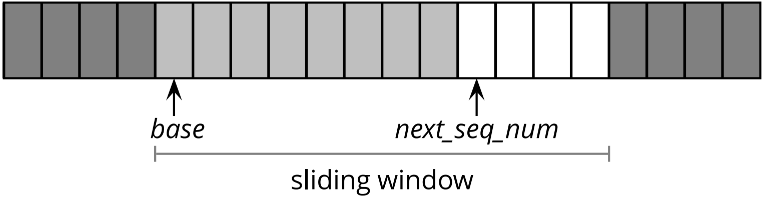
# Selective Repeat

- **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 falgs
  - 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

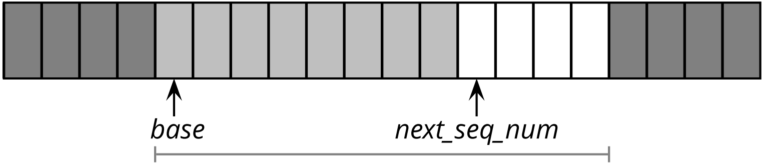
# Selective Repeat

- **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 falgs
  - 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



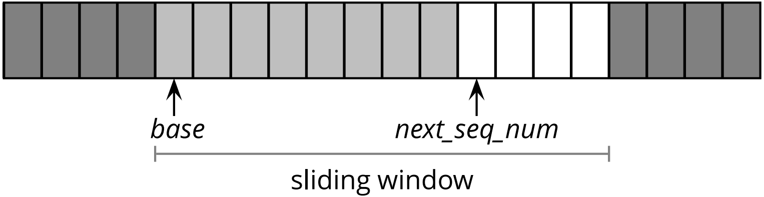
# Selective Repeat: Sender



sliding window

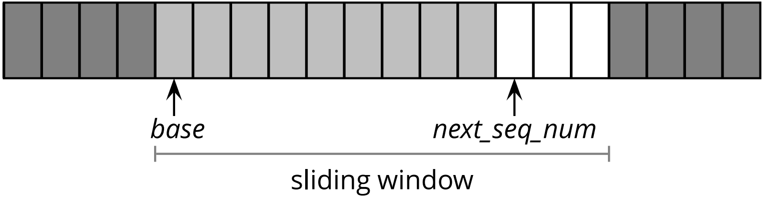
- r_send($pkt_1$)

# Selective Repeat: Sender



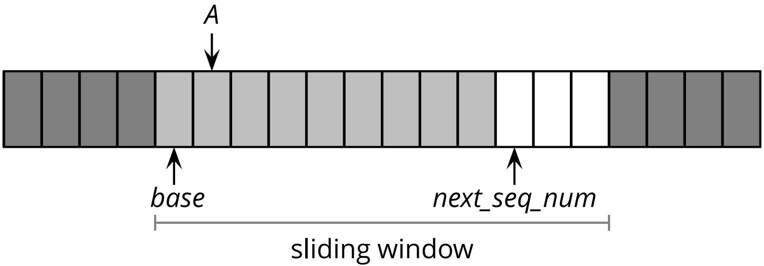
- r_send($pkt_1$)
  - u_send([$pkt_1$,*next_seq_num*])
  - start_timer(*next_seq_num*)

# Selective Repeat: Sender



- r_send($pkt_1$)
  - u_send([$pkt_1$,*next_seq_num*])
  - start_timer(*next_seq_num*)
  - $next\_seq\_num = next\_seq\_num + 1$

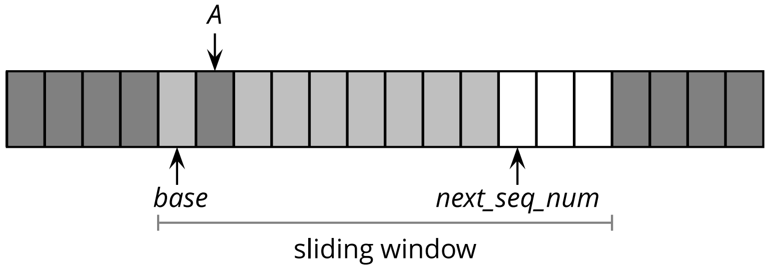
# Selective Repeat: Sender



- r_send($pkt_1$)
  - u_send([$pkt_1$,*next_seq_num*])
  - start_timer(*next_seq_num*)
  - $next\_seq\_num = next\_seq\_num + 1$

- u_recv([ACK,*A*])

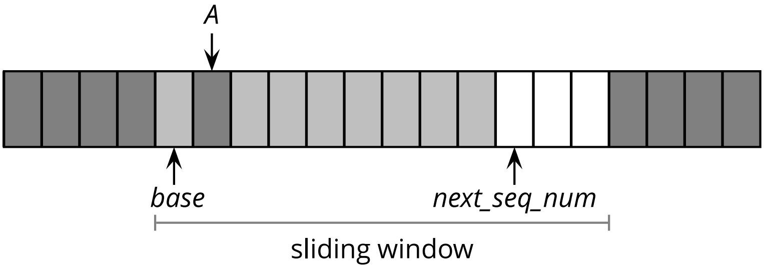
# Selective Repeat: Sender



- r_send($pkt_1$)
  - u_send([$pkt_1$,*next_seq_num*])
  - start_timer(*next_seq_num*)
  - $next\_seq\_num = next\_seq\_num + 1$

- u_recv([ACK,*A*])
  - $acks[A] = 1$ *// remember that A was ACK'd*

# Selective Repeat: Sender

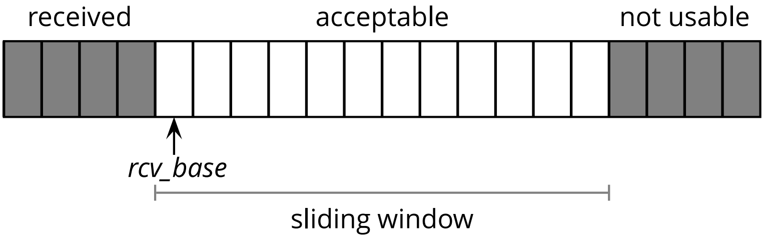


- r_send($pkt_1$)
  - u_send([$pkt_1$,*next_seq_num*])
  - start_timer(*next_seq_num*)
  - $next\_seq\_num = next\_seq\_num + 1$

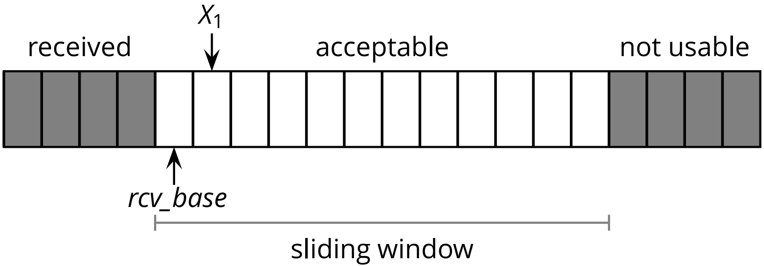
- u_recv([ACK,*A*])
  - $acks[A] = 1$ *// remember that A was ACK'd*
  - acknewledgements are no longer "cumulative"

# Selective Repeat: Receiver

# Selective Repeat: Receiver

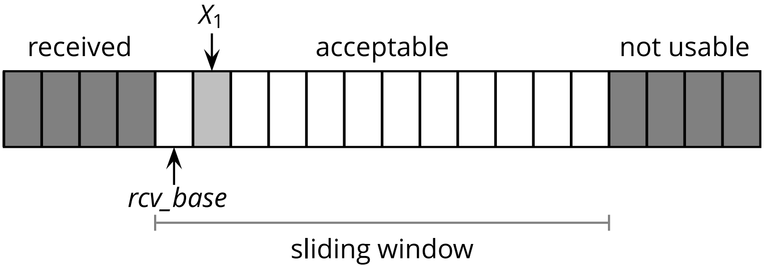


# Selective Repeat: Receiver



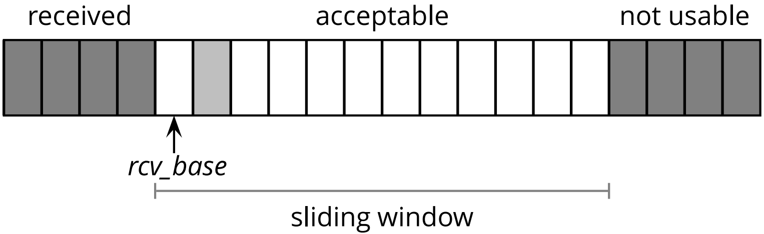
- u_recv([$pkt_1$,$X_1$]) **and** $rcv\_base \leq X_1 < rcv\_base + W$

# Selective Repeat: Receiver

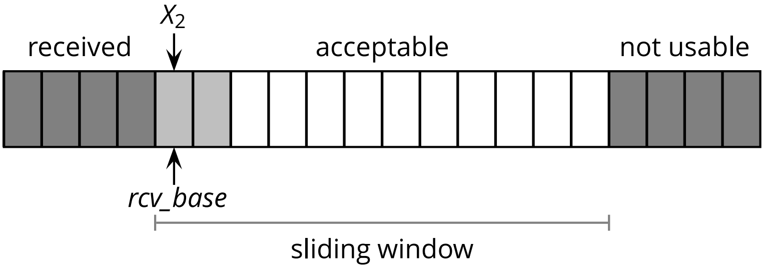


- u_recv([$pkt_1$,$X_1$]) **and** $rcv\_base \le X_1 < rcv\_base + W$
  - $buffer[X_1] = pkt_1$
  - u_send($[ACK, X_1]^*$) *// no longer a "cumulative" ACK*

# Selective Repeat: Receiver

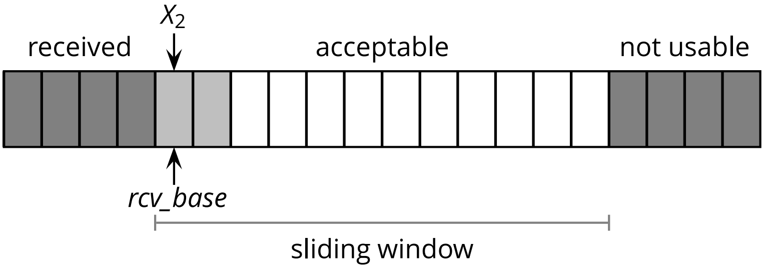


# Selective Repeat: Receiver



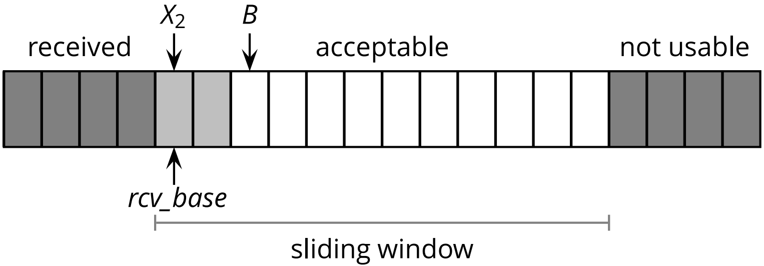
- u_recv([$pkt_2$,$X_2$]) **and** $rcv\_base \leq X_2 < rcv\_base + W$
  - $buffer[X_2] = pkt_2$
  - u_send($[ACK, X_2]^*$)

# Selective Repeat: Receiver



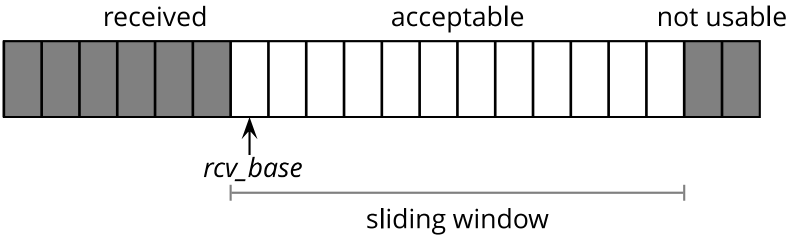
- u_recv([$pkt_2$,$X_2$]) **and** $rcv\_base \leq X_2 < rcv\_base + W$
  - $buffer[X_2] = pkt_2$
  - u_send($[ACK, X_2]^*$)
  - **if** $X_2$ == $rcv\_base$:

# Selective Repeat: Receiver



```
u_recv([$pkt_2$,$X_2$]) and $rcv\_base \le X_2 < rcv\_base + W$
    $buffer[X_2] = pkt_2$
    u_send($[ACK, X_2]^*$)
    if $X_2$ == $rcv\_base$:
        $B = first\_missing\_seq\_num()$
        foreach $i$ in $rcv\_base \ldots B - 1$:
            r_recv($buffer[i]$)
```

# Selective Repeat: Receiver

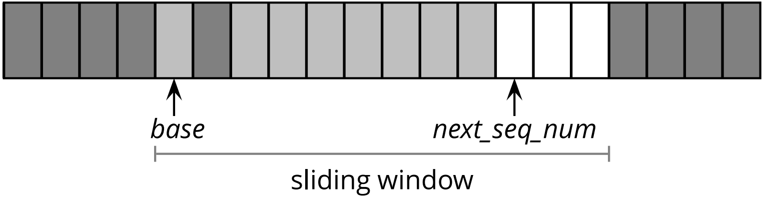


- u_recv([$pkt_2$,$X_2$]) **and** $rcv\_base \le X_2 < rcv\_base + W$
  - $buffer[X_2] = pkt_2$
  - u_send([$ACK, X_2$]$^*$)
  - **if** $X_2$ == $rcv\_base$:

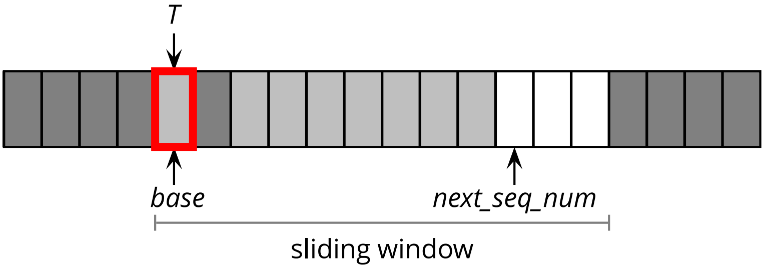
```
        $B = first\_missing\_seq\_num()$
        foreach $i$ in $rcv\_base \ldots B - 1$:
            r_recv($buffer[i]$)
        $rcv\_base = B$
```

# Selective Repeat: Sender

# Selective Repeat: Sender

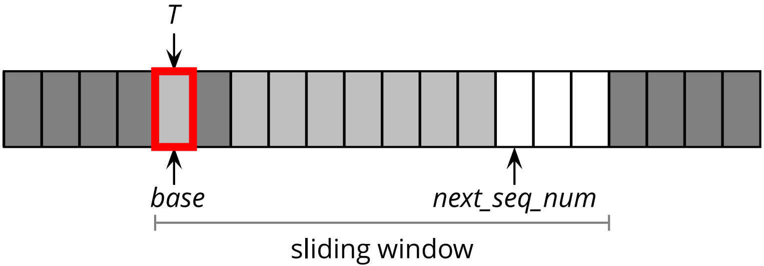


# Selective Repeat: Sender



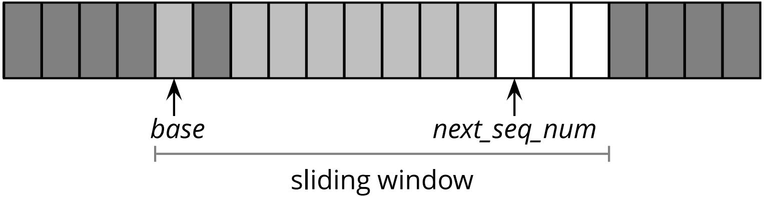
- Timeout for sequence number *T*

# Selective Repeat: Sender

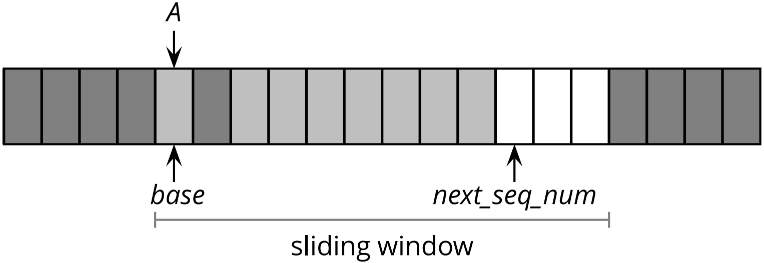


- Timeout for sequence number *T*
  - u_send([*pkt*[*T*], *T*]$^*$)

# Selective Repeat: Sender

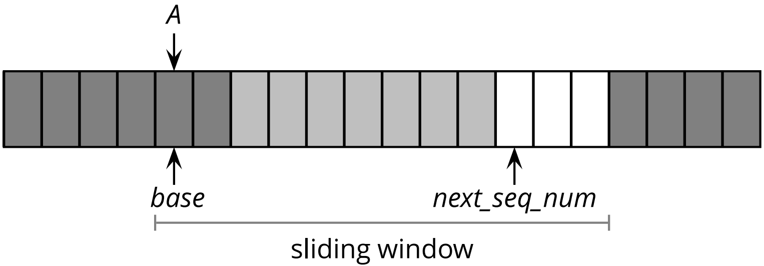


# Selective Repeat: Sender



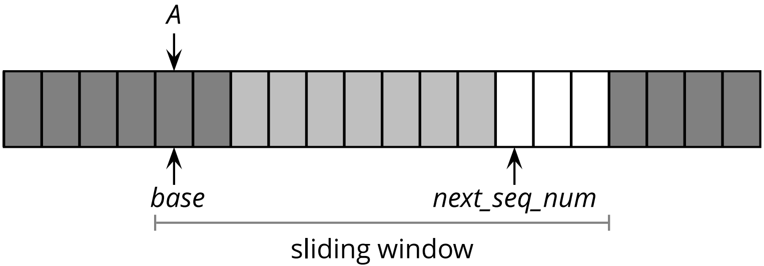
- u_recv([ACK,*A*])

# Selective Repeat: Sender



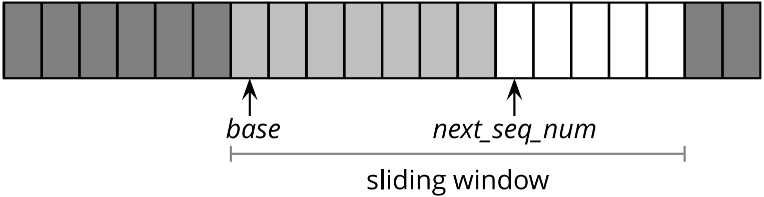
- u_recv([ACK,*A*])
  - $acks[A] = 1$

# Selective Repeat: Sender



- u_recv([ACK,*A*])
  - $acks[A] = 1$
  - **if** *A* == *base*:

# Selective Repeat: Sender



- u_recv([ACK,*A*])

```
acks[A] = 1
if A == base:
        base = first_missing_ack_num()
```