Computer Networks: The Transport Layer

Antonio Carzaniga

Faculty of Informatics
University of Lugano

St. Petersburg, Russia
April 26, 2008

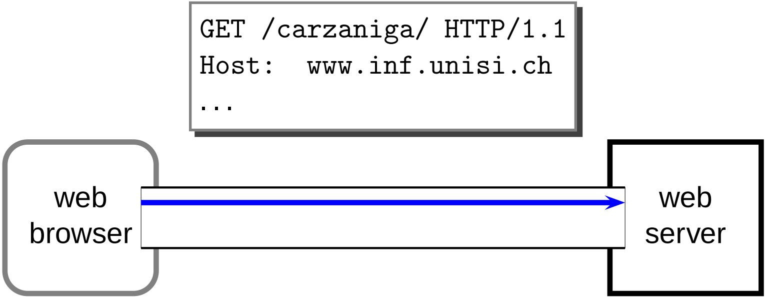
Outline

- Very quick intro to computer networking
- The *transport layer*
  - reliability
  - congestion control
  - brief intro to TCP

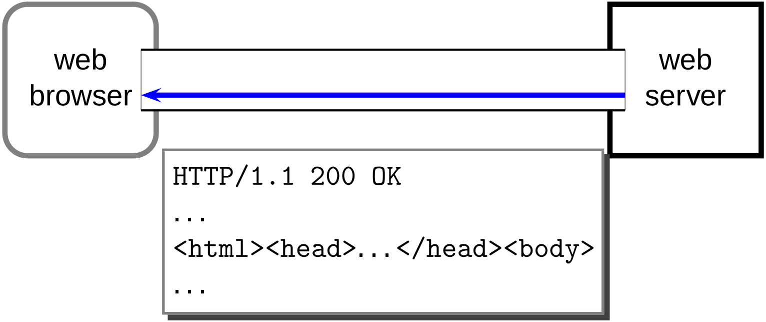
Application

Application

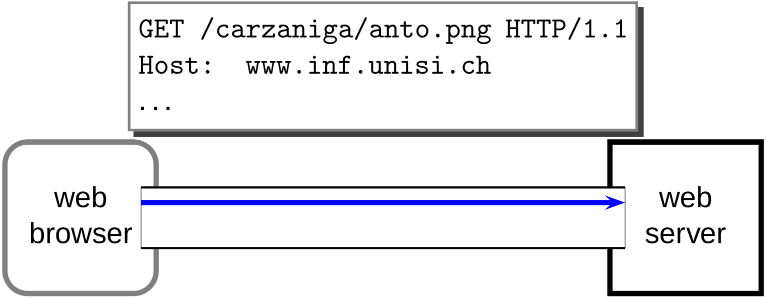
Application



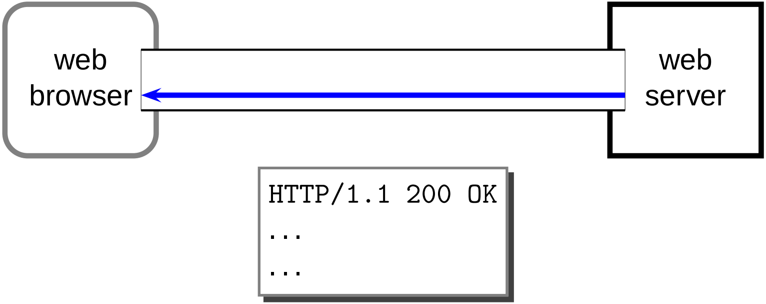
Application



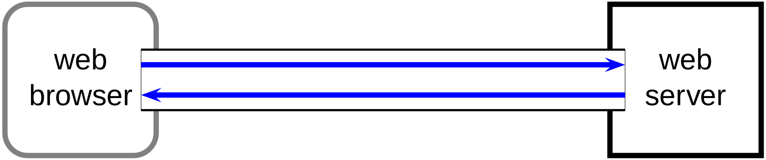
Application



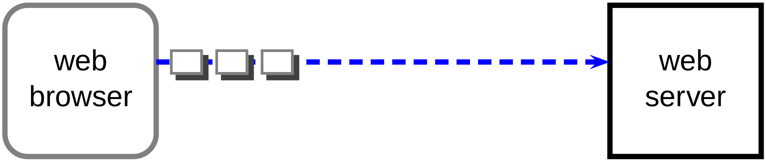
Application



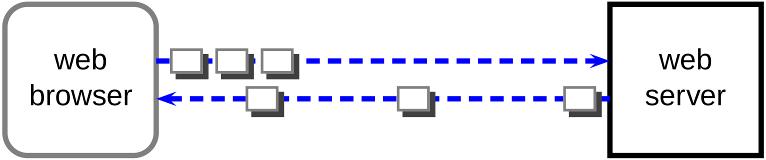
Transport



Transport

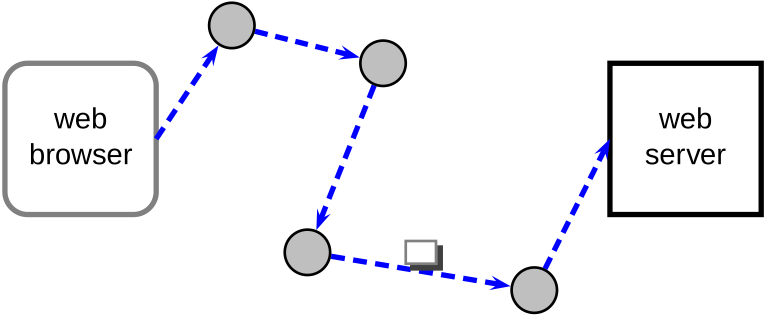


Transport

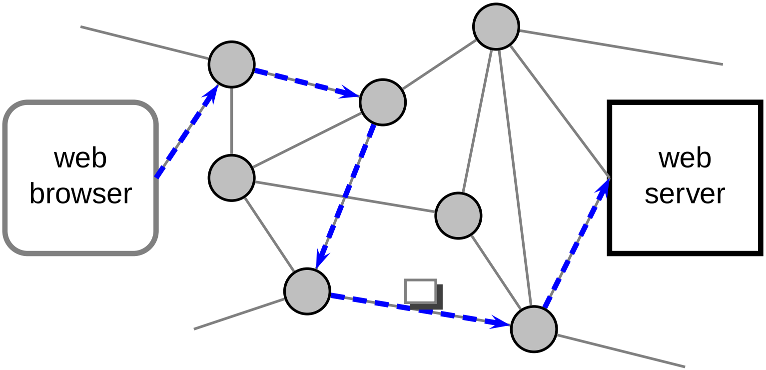


Network

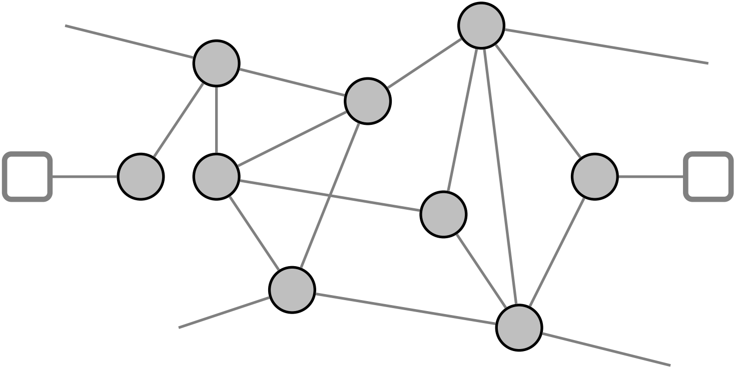
Network



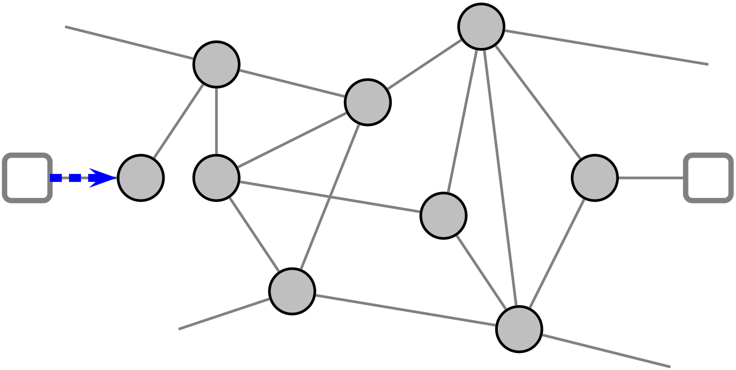
Network



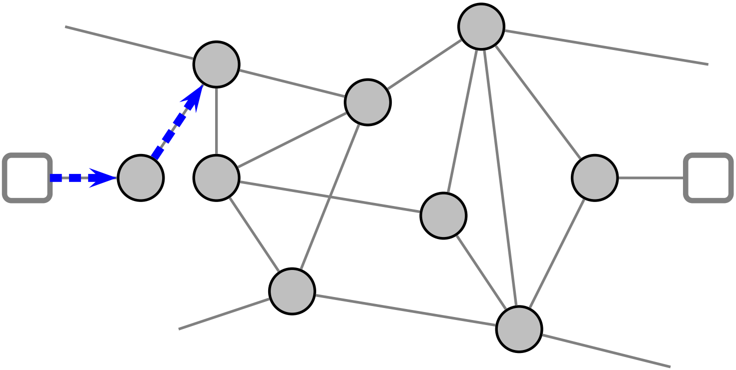
Network



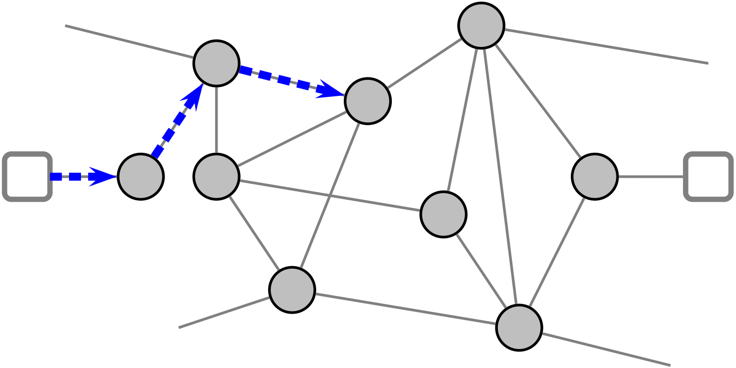
Network



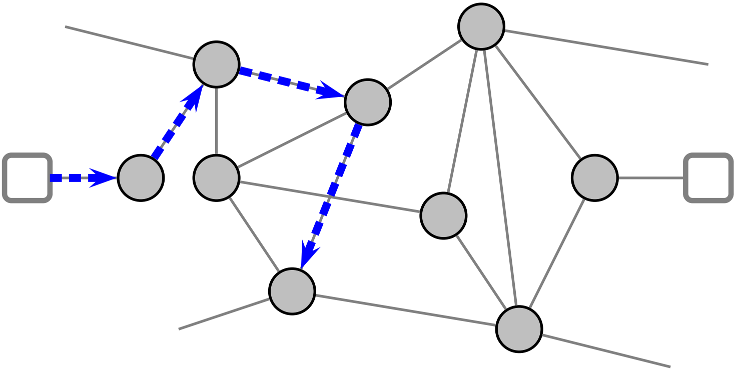
Network



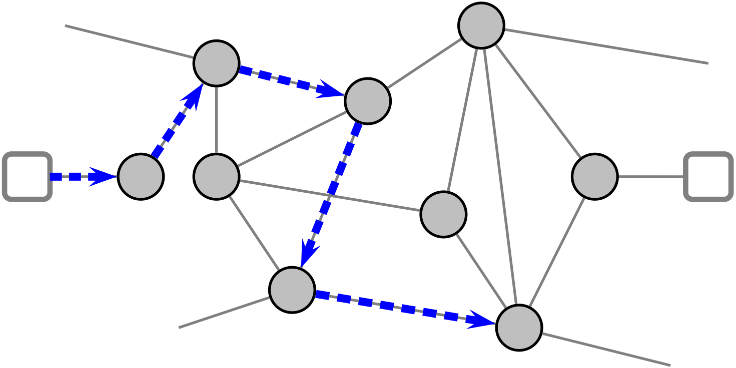
Network



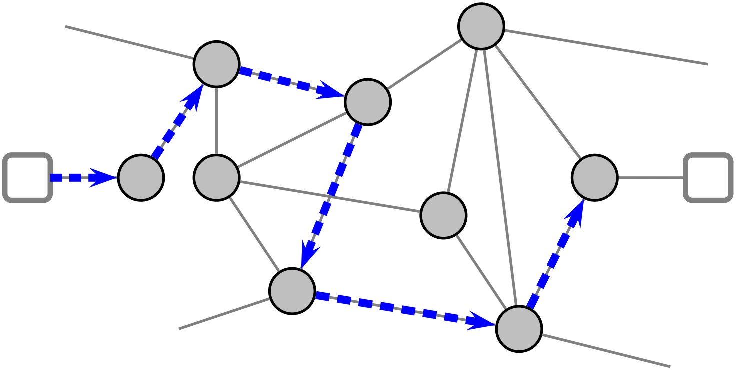
Network



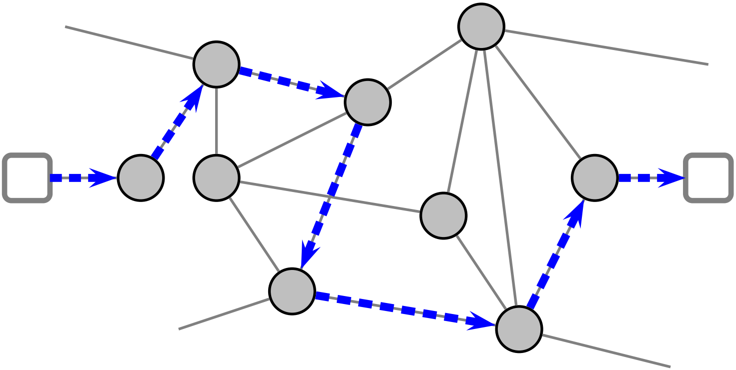
Network



Network

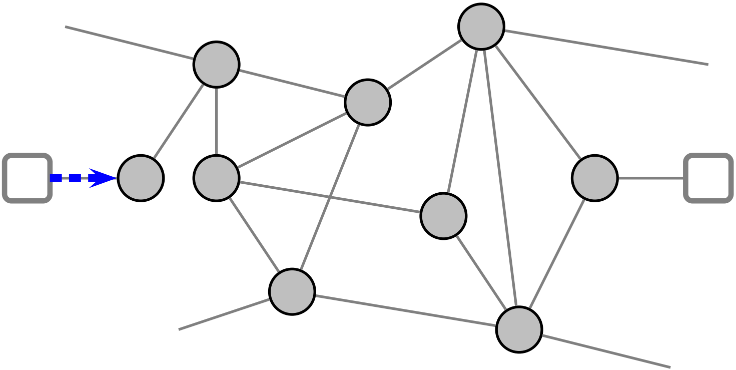


Network



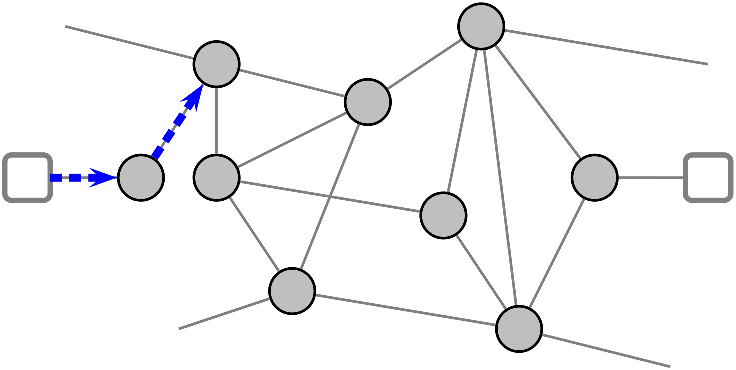
- Potentially *multiple paths* for the same source/destination

Network



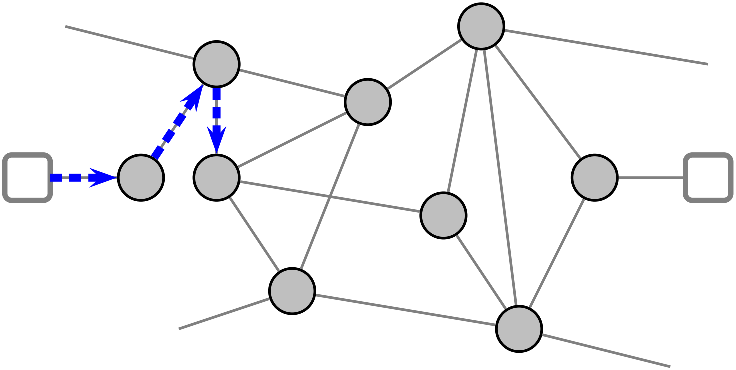
- Potentially *multiple paths* for the same source/destination

Network



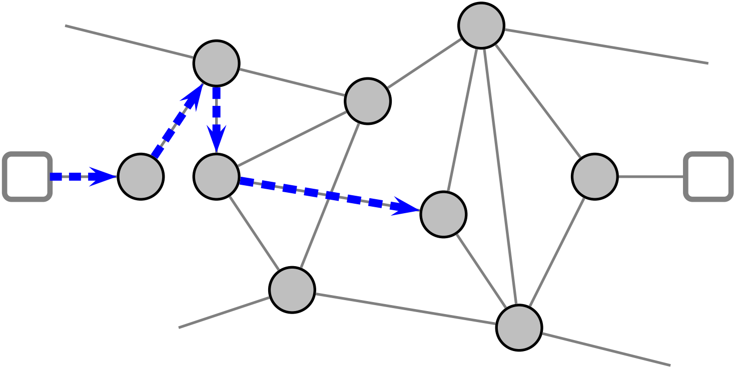
- Potentially *multiple paths* for the same source/destination

Network



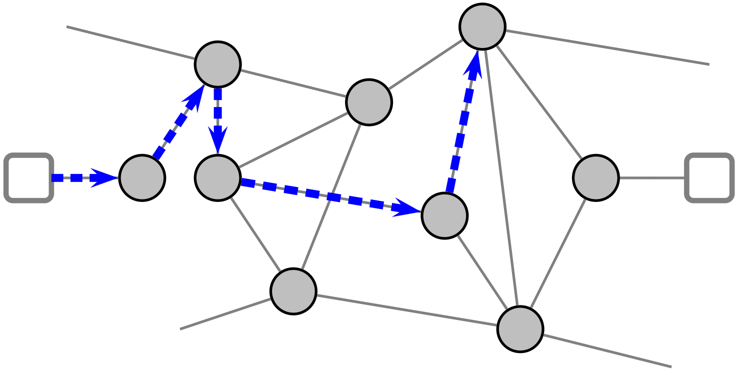
- Potentially *multiple paths* for the same source/destination

Network



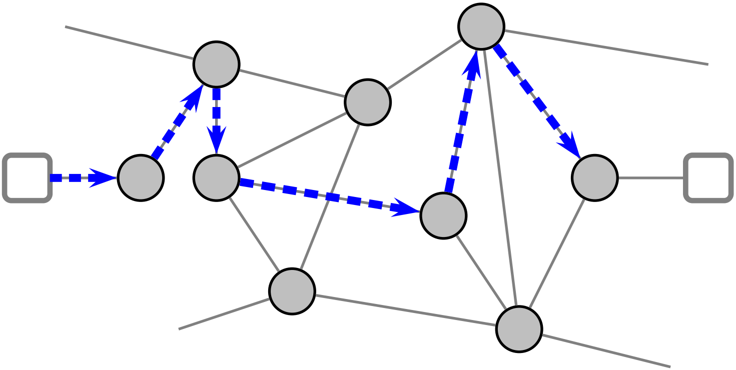
- Potentially *multiple paths* for the same source/destination

Network



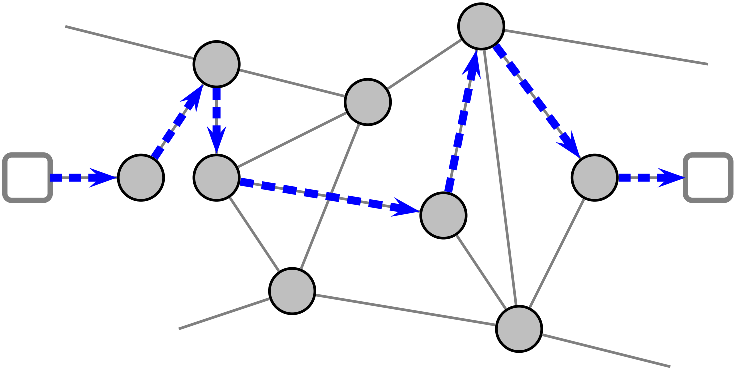
- Potentially *multiple paths* for the same source/destination

Network



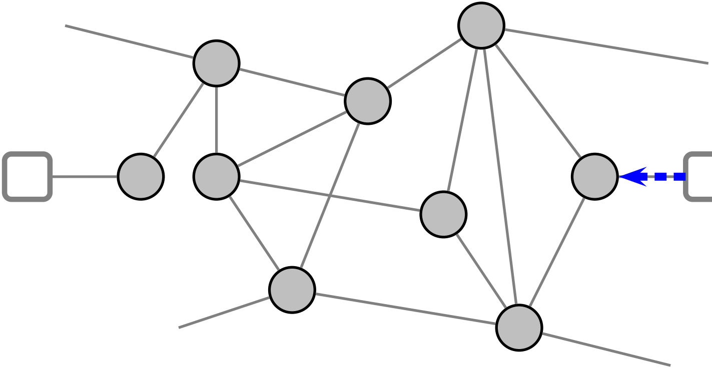
- Potentially *multiple paths* for the same source/destination

Network



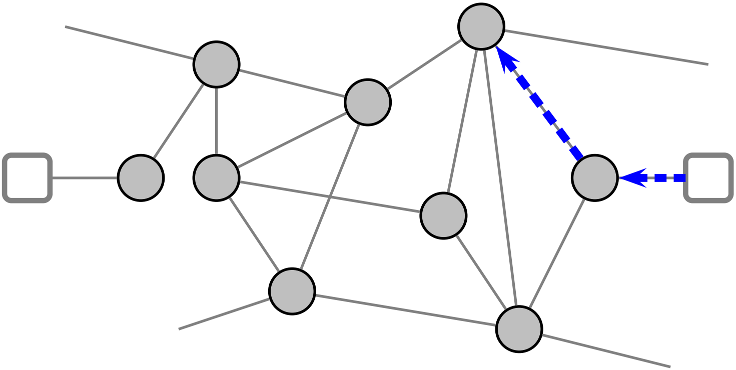
- Potentially *multiple paths* for the same source/destination

Network



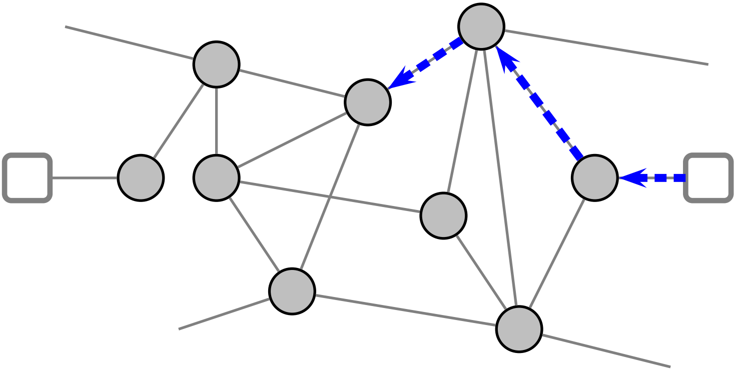
- Potentially *multiple paths* for the same source/destination
- Potentially *asymmetric paths*

Network



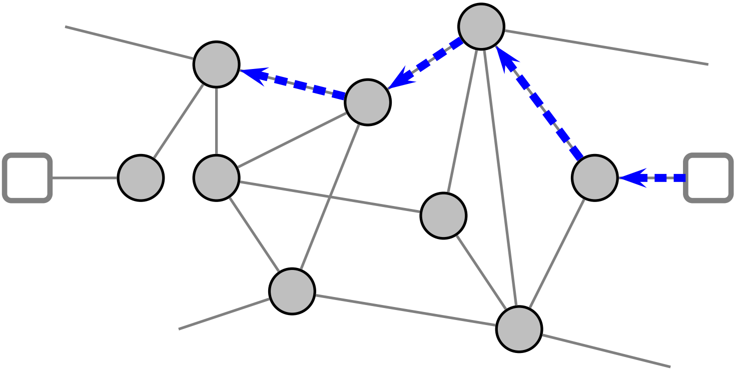
- Potentially *multiple paths* for the same source/destination
- Potentially *asymmetric paths*

Network



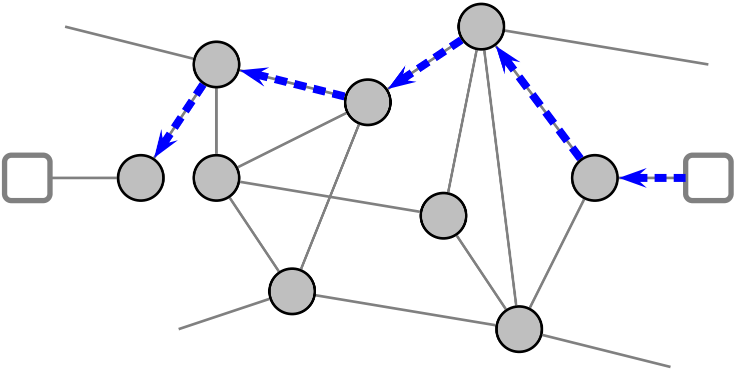
- Potentially *multiple paths* for the same source/destination
- Potentially *asymmetric paths*

Network



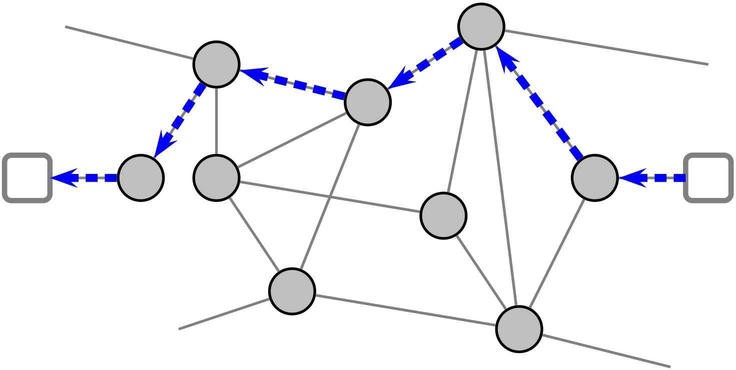
- Potentially *multiple paths* for the same source/destination
- Potentially *asymmetric paths*

Network



- Potentially *multiple paths* for the same source/destination
- Potentially *asymmetric paths*

Network



- Potentially *multiple paths* for the same source/destination
- Potentially *asymmetric paths*

A “Datagram” Network

A “Datagram” Network

- *Packet-switched network*

A “Datagram” Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*

A “Datagram” Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*

- *Connectionless service*

A "Datagram" Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*
- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase

A “Datagram” Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*
- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase
- *“Best-effort” service*

A “Datagram” Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*
- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase
- *“Best-effort” service*
  - delivery guarantee: none

A "Datagram" Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*
- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase
- *"Best-effort" service*
  - delivery guarantee: none
  - maximum latency guarantee: none

A “Datagram” Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*

- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase

- *“Best-effort” service*
  - delivery guarantee: none
  - maximum latency guarantee: none
  - bandwidth guarantee: none

A "Datagram" Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*
- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase
- *"Best-effort" service*
  - delivery guarantee: none
  - maximum latency guarantee: none
  - bandwidth guarantee: none
  - in-order delivery guarantee: none

A “Datagram” Network

- *Packet-switched network*
  - information is transmitted in discrete units called *datagrams*
- *Connectionless service*
  - a datagram is a *self-contained message*
  - treated independently by the network
  - no connection setup/tear-down phase
- *“Best-effort” service*
  - delivery guarantee: none
  - maximum latency guarantee: none
  - bandwidth guarantee: none
  - in-order delivery guarantee: none
  - congestion indication: none

Transport-Layer Value-Added Service

Transport-Layer Value-Added Service

- *Transport-layer multiplexing/demultiplexing*
  - i.e., connecting applications as opposed to hosts

Transport-Layer Value-Added Service

- *Transport-layer multiplexing/demultiplexing*
  - i.e., connecting applications as opposed to hosts

- *Reliable data transfer*
  - i.e., integrity and possibly ordered delivery

Transport-Layer Value-Added Service

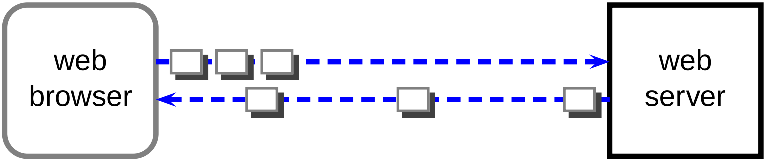
- *Transport-layer multiplexing/demultiplexing*
  - i.e., connecting applications as opposed to hosts
- *Reliable data transfer*
  - i.e., integrity and possibly ordered delivery
- *Connections*
  - i.e., streams
  - can be seen as the same as ordered dalivery

Transport-Layer Value-Added Service

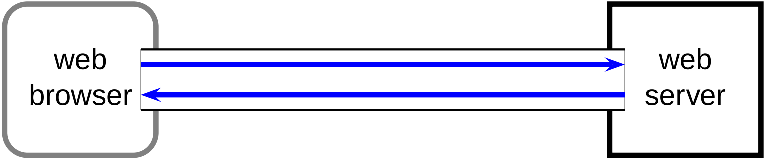
- *Transport-layer multiplexing/demultiplexing*
  - i.e., connecting applications as opposed to hosts
- *Reliable data transfer*
  - i.e., integrity and possibly ordered delivery
- *Connections*
  - i.e., streams
  - can be seen as the same as ordered dalivery
- *Congestion control*
  - i.e., end-to-end traffic (admission) control so as to avoid destructive congestions within the network

Transport

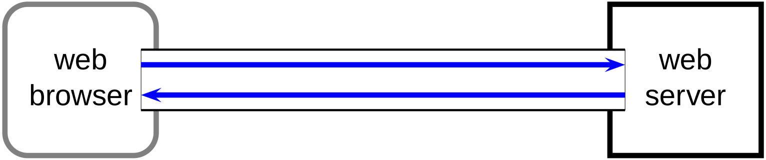
Transport



Transport



Transport



unreliable, datagram $\Longrightarrow$ *reliable “stream”*

Part I

Reliable Streams on Unreliable Networks

Finite-State Machines

Finite-State Machines

- A *finite-state machine (FSM)* is a mathematical abstraction
  - a.k.a., finite-state automaton (FSA), deterministic finite-state automaton (DFA), non-deterministic finite-state automaton (NFA)

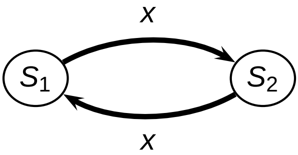
Finite-State Machines

- A *finite-state machine (FSM)* is a mathematical abstraction
  - a.k.a., finite-state automaton (FSA), deterministic finite-state automaton (DFA), non-deterministic finite-state automaton (NFA)
- Very useful to specify and implement network protocols

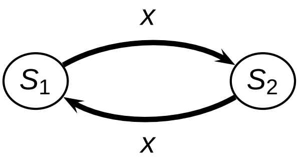
Finite-State Machines

- A *finite-state machine (FSM)* is a mathematical abstraction
  - a.k.a., finite-state automaton (FSA), deterministic finite-state automaton (DFA), non-deterministic finite-state automaton (NFA)
- Very useful to specify and implement network protocols
- Ubiquitous in computer science
  - theory of formal languages
  - compiler design
  - theory of computation
  - text processing
  - behavior specification
  - …

Finite-State Machines

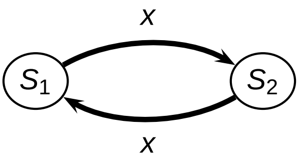


Finite-State Machines



- *States* are represented as *nodes in a graph*

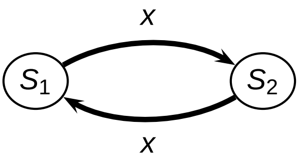
Finite-State Machines



- *States* are represented as *nodes in a graph*

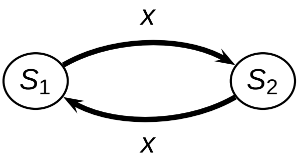
- *Transitions* are represented as *directed edges in the graph*

Finite-State Machines



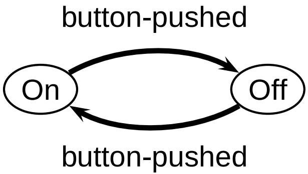
- *States* are represented as *nodes in a graph*
- *Transitions* are represented as *directed edges in the graph*
  - an edge labeled x going from state S_1 to state S_2 says that when the machine is in state S_1 and event x occurs, the machine switches to state S_2

Finite-State Machines

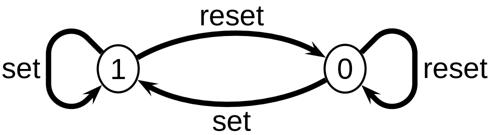


- *States* are represented as *nodes in a graph*

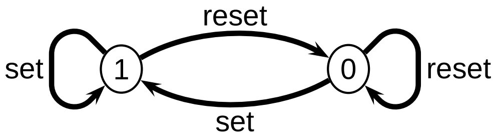
- *Transitions* are represented as *directed edges in the graph*
  - an edge labeled x going from state S_1 to state S_2 says that when the machine is in state S_1 and event x occurs, the machine switches to state S_2

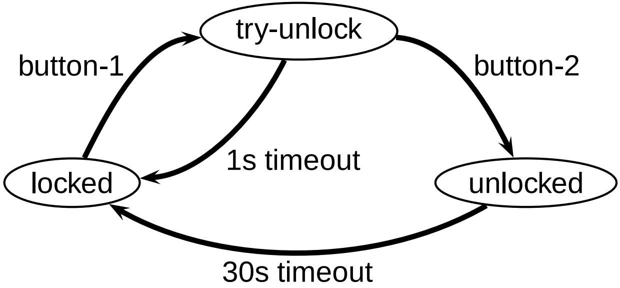


Finite-State Machines

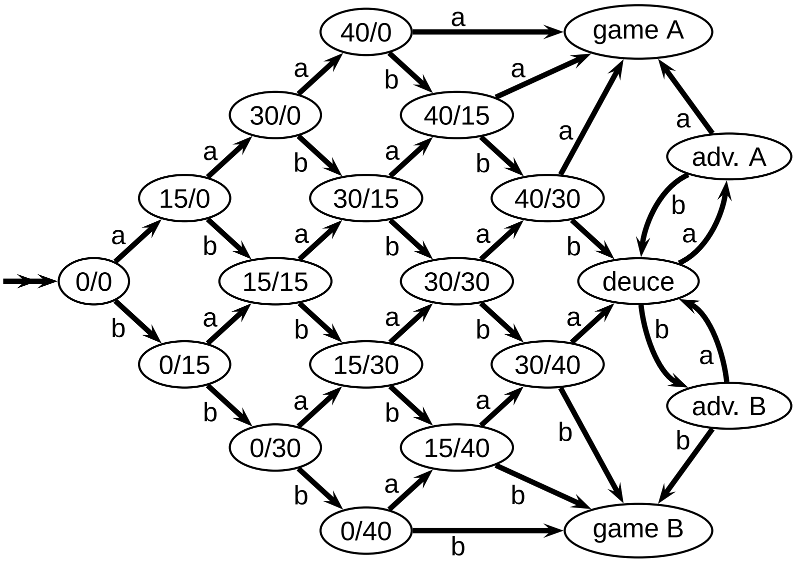


Finite-State Machines





Finite-State Machines



FSMs to Specify Protocols

FSMs to Specify Protocols

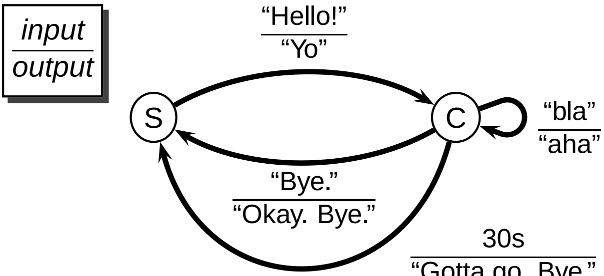
- *States* represent the state of a protocol

FSMs to Specify Protocols

- *States* represent the state of a protocol
- *Transitions* are characterized by an *event/action* label
  - *event:* typically consists of an *input message* or a *timeout*
  - *action:* typically consists of an *output message*

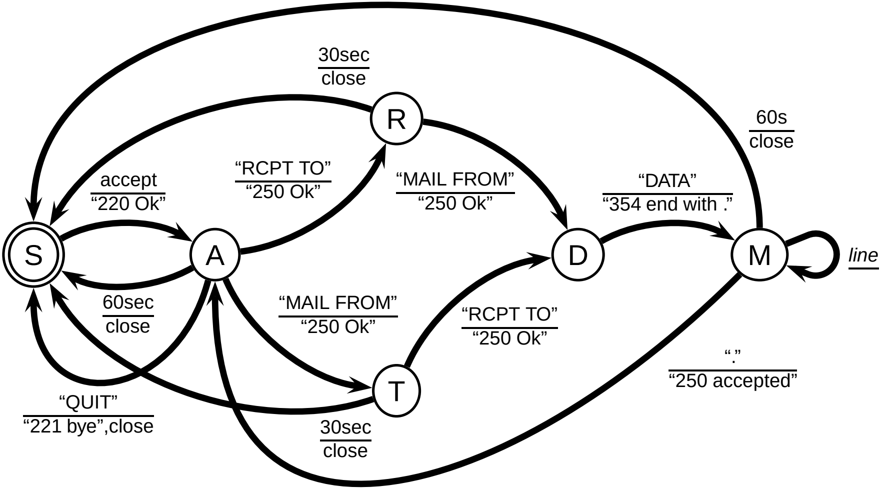
FSMs to Specify Protocols

- *States* represent the state of a protocol
- *Transitions* are characterized by an *event/action* label
  - *event:* typically consists of an *input message* or a *timeout*
  - *action:* typically consists of an *output message*
- E.g., here's a specification of a "simple conversation protocol"



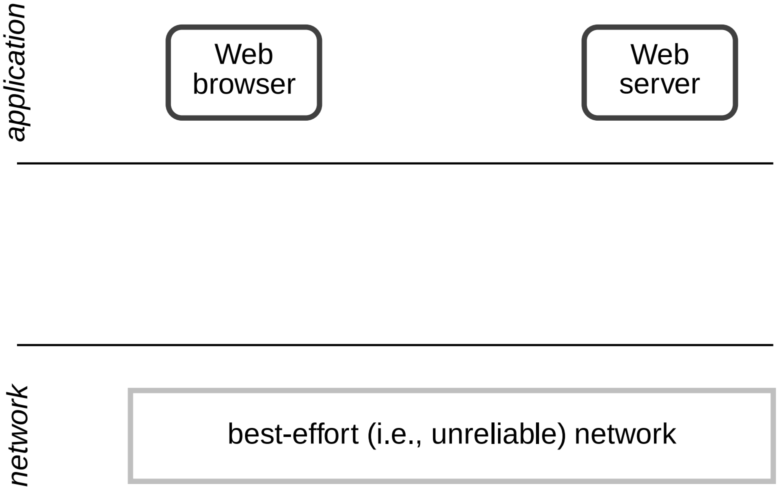
Example

E.g., a subset of a server-side, SMTP-like protocol

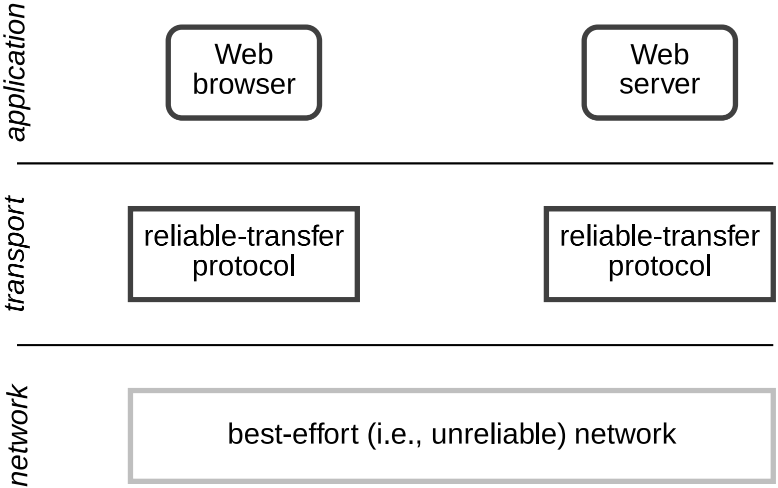


Back to Reliable Data Transfer

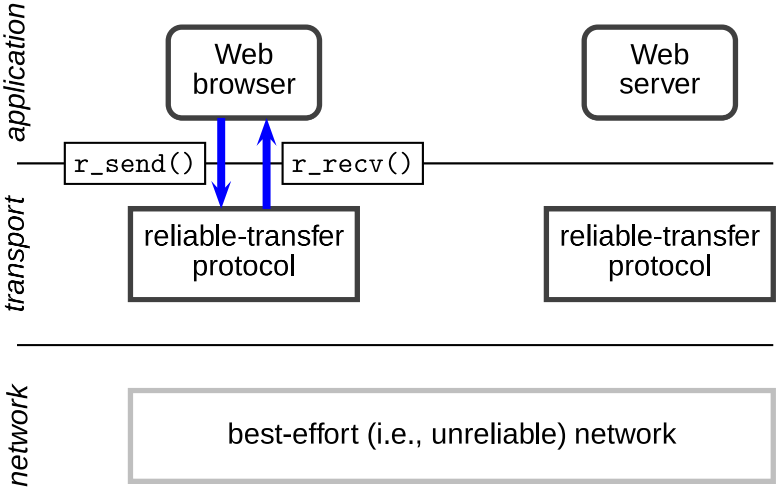
Back to Reliable Data Transfer



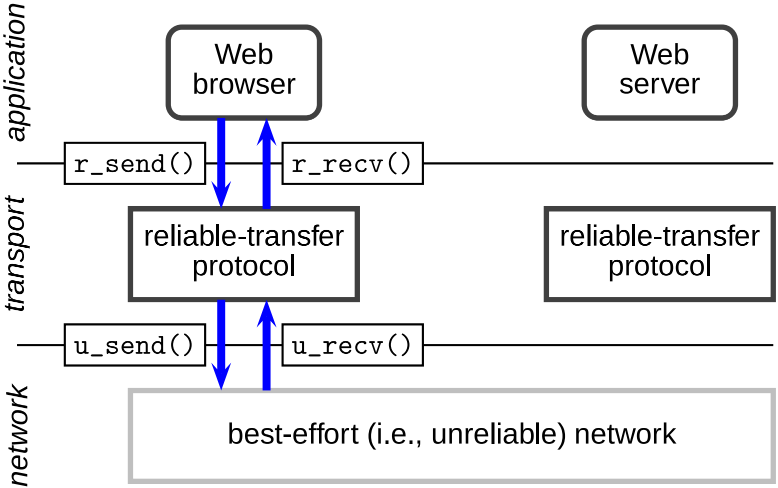
Back to Reliable Data Transfer



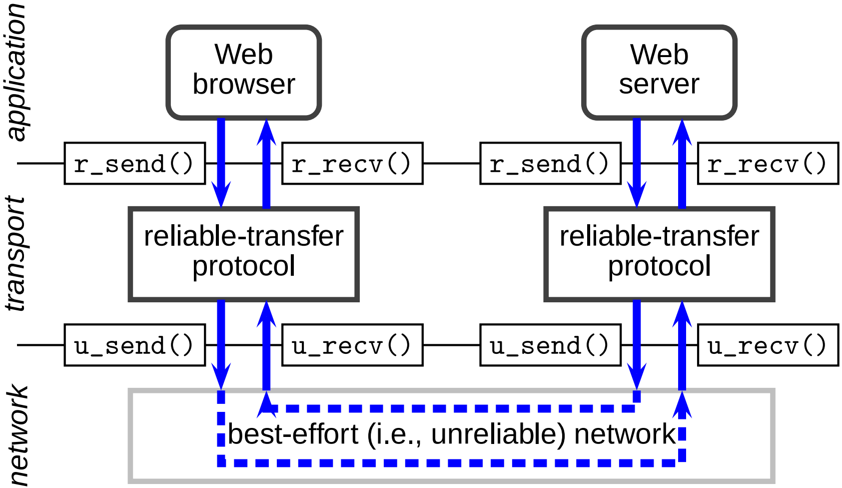
Back to Reliable Data Transfer



Back to Reliable Data Transfer

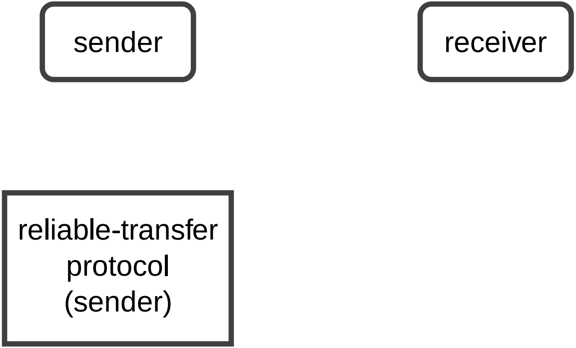


Back to Reliable Data Transfer

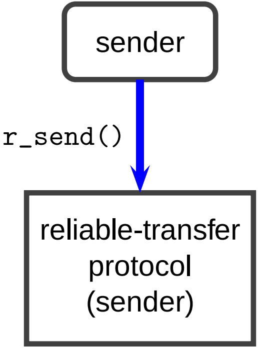


Reliable Data Transfer Model

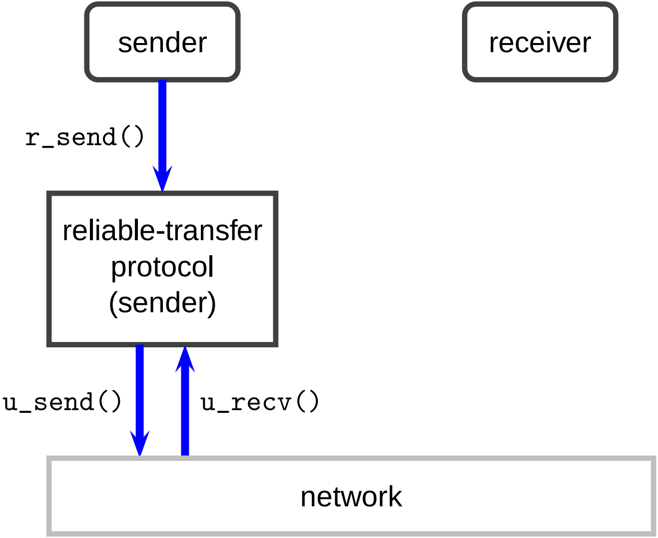
Reliable Data Transfer Model



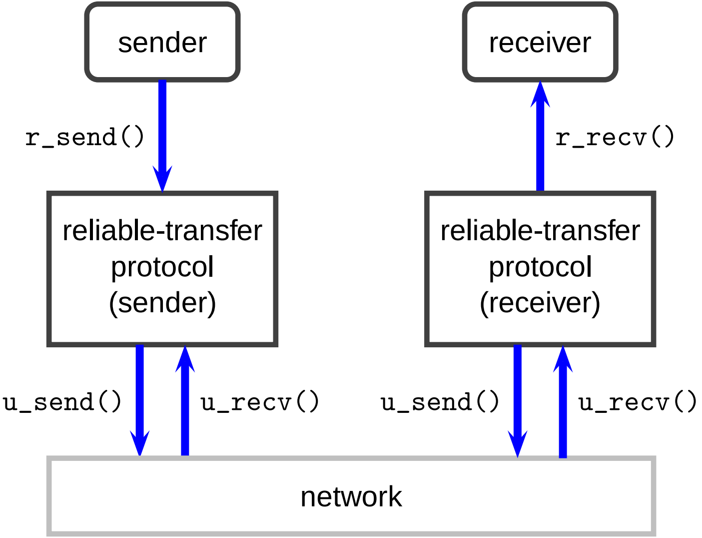
Reliable Data Transfer Model



Reliable Data Transfer Model



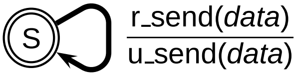
Reliable Data Transfer Model



Baseline Protocol

- Reliable transport protocol that uses a reliable network (obviously a contrived example)

sender

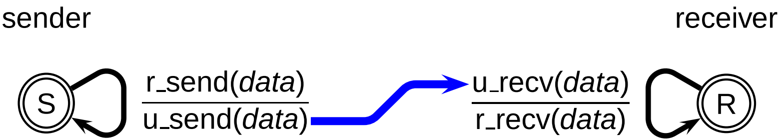


Baseline Protocol

- Reliable transport protocol that uses a reliable network (obviously a contrived example)

Baseline Protocol

- Reliable transport protocol that uses a reliable network (obviously a contrived example)



Noisy Channel

Noisy Channel

- Reliable transport protocol over a network with *bit errors*
  - every so often, a bit will be modified during transmission
    - that is, a bit will be "flipped"
  - however, no packets will be lost

Noisy Channel

- Reliable transport protocol over a network with *bit errors*
  - every so often, a bit will be modified during transmission
    - that is, a bit will be “flipped”
  - however, no packets will be lost
- How do people deal with such situations?
  (Think of a phone call over a noisy line)

Noisy Channel

- Reliable transport protocol over a network with *bit errors*
  - every so often, a bit will be modified during transmission
    - that is, a bit will be "flipped"
  - however, no packets will be lost
- How do people deal with such situations?
  (Think of a phone call over a noisy line)
  - *error detection:* the receiver must be able to know when a received packet is corrupted (i.e., when it contains flipped bits)

Noisy Channel

- Reliable transport protocol over a network with *bit errors*
  - every so often, a bit will be modified during transmission
    - that is, a bit will be "flipped"
  - however, no packets will be lost
- How do people deal with such situations?
  (Think of a phone call over a noisy line)
  - *error detection:* the receiver must be able to know when a received packet is corrupted (i.e., when it contains flipped bits)
  - *receiver feedback:* the receiver must be able to alert the sender that a corrupted packet was received

Noisy Channel

- Reliable transport protocol over a network with *bit errors*
  - every so often, a bit will be modified during transmission
    - that is, a bit will be "flipped"
  - however, no packets will be lost
- How do people deal with such situations?
  (Think of a phone call over a noisy line)
  - *error detection:* the receiver must be able to know when a received packet is corrupted (i.e., when it contains flipped bits)
  - *receiver feedback:* the receiver must be able to alert the sender that a corrupted packet was received
  - *retransmission:* the sender retransmits corrupted packets

Error Detection

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message
    - receiver computes the *xor* of all the bits and concludes that there was an error if the result is not 0 (i.e., if it is 1)

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message
    - receiver computes the *xor* of all the bits and concludes that there was an error if the result is not 0 (i.e., if it is 1)

  Sender:
  message is 1001011011101000

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message
    - receiver computes the *xor* of all the bits and concludes that there was an error if the result is not 0 (i.e., if it is 1)

  Sender:
  message is 1001011011101000 $\Rightarrow$ send 1001011011101000$\underline{0}$

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message
    - receiver computes the *xor* of all the bits and concludes that there was an error if the result is not 0 (i.e., if it is 1)

  Sender:
  message is 1001011011101000 $\Rightarrow$ send 1001011011101000$\underline{0}$

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message
    - receiver computes the *xor* of all the bits and concludes that there was an error if the result is not 0 (i.e., if it is 1)

  Sender:
  message is 1001011011101000 $\Rightarrow$ send 1001011011101000$\underline{0}$

  Receiver:
  receives 10010110101010000

Error Detection

- Key idea: *sending redundant information*
  - e.g., the sender could repeat the message twice
  - error iff the receiver hears two different messages
  - not very efficient: uses twice the number of bits

- *Error-detection codes*
  - e.g., the *parity bit*
    - sender adds one bit that is the *xor* of all the bits in the message
    - receiver computes the *xor* of all the bits and concludes that there was an error if the result is not 0 (i.e., if it is 1)

  Sender:
  message is 10010110111101000 $\Rightarrow$ send 1001011011110100$\underline{0}$

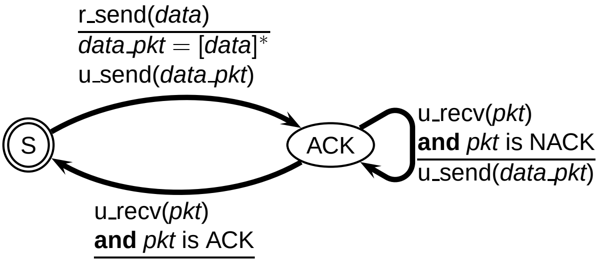
  Receiver:
  receives 100101101010100000 $\Rightarrow$ error!

Noisy Channel

- Sender
  - [*data*]* indicates a packet containing *data* plus an error-detection code (i.e., a checksum)

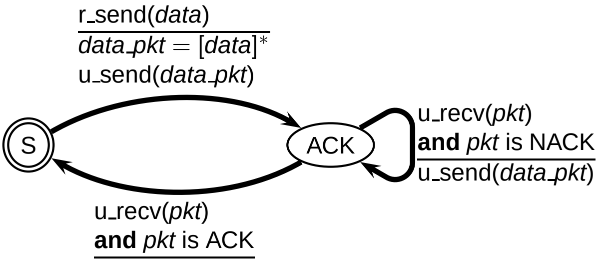
Noisy Channel

- Sender
  - [*data*]* indicates a packet containing *data* plus an error-detection code (i.e., a checksum)

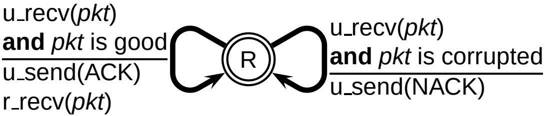


Noisy Channel

- Sender
  - *[data]** indicates a packet containing *data* plus an error-detection code (i.e., a checksum)



- Receiver



Noisy Channel

Noisy Channel

- This protocol is “synchronous” or “stop-and-wait” for each packet
  - i.e., the sender must receive a (positive) acknowledgment before it can take more data from the application layer

Noisy Channel

- This protocol is "synchronous" or "stop-and-wait" for each packet
  - i.e., the sender must receive a (positive) acknowledgment before it can take more data from the application layer

- Does the protocol really work?

Noisy Channel

- This protocol is "synchronous" or "stop-and-wait" for each packet
  - i.e., the sender must receive a (positive) acknowledgment before it can take more data from the application layer
- Does the protocol really work?
- What happens if an error occurs within an ACK/NACK packet?

Dealing With Bad ACKs/NACKs

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: "let's go see Taxi Driver"
  2. receiver hears: "let's . . . Taxi . . . "
  3. receiver says: "Repeat message!"

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”
  5. sender says: “Repeat your ACK please!”
  6. . . .

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”
  5. sender says: “Repeat your ACK please!”
  6. . . .

  Not Good: this protocol doesn’t seem to end

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”
  5. sender says: “Repeat your ACK please!”
  6. . . .

  Not Good: this protocol doesn’t seem to end

- Make ACK/NACK packets so redundant that the sender can always figure out what the message is, even if a few bits are corrupted

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”
  5. sender says: “Repeat your ACK please!”
  6. . . .

  Not Good: this protocol doesn’t seem to end

- Make ACK/NACK packets so redundant that the sender can always figure out what the message is, even if a few bits are corrupted
  - good enough for channels that do not loose messages

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs
  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”
  5. sender says: “Repeat your ACK please!”
  6. . . .

  Not Good: this protocol doesn’t seem to end

- Make ACK/NACK packets so redundant that the sender can always figure out what the message is, even if a few bits are corrupted
  - good enough for channels that do not loose messages

- Assume a NACK and simply retransmit the packet

Dealing With Bad ACKs/NACKs

- Negative acknowledgments for ACKs and NACKs

  1. sender says: “let’s go see Taxi Driver”
  2. receiver hears: “let’s . . . Taxi . . . ”
  3. receiver says: “Repeat message!”
  4. sender hears: “. . . *noise* . . . ”
  5. sender says: “Repeat your ACK please!”
  6. . . .

  Not Good: this protocol doesn’t seem to end

- Make ACK/NACK packets so redundant that the sender can always figure out what the message is, even if a few bits are corrupted
  - good enough for channels that do not loose messages

- Assume a NACK and simply retransmit the packet
  - good idea, but it introduces *duplicate packets* (why?)

Dealing With Duplicate Packets

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission
  1. sender says: “7: let’s go see Taxi Driver”

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission
  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver passes “let’s go see Taxi Driver” to application layer
  4. receiver says: “Got it!” (i.e., ACK)

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission

  1. sender says: "7: let's go see Taxi Driver"
  2. receiver hears: "7: let's go see Taxi Driver"
  3. receiver passes "let's go see Taxi Driver" to application layer
  4. receiver says: "Got it!" (i.e., ACK)
  5. sender hears: "... *noise* ..."

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission

  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver passes “let’s go see Taxi Driver” to application layer
  4. receiver says: “Got it!” (i.e., ACK)
  5. sender hears: “. . . *noise* . . . ”
  6. sender (assuming a NACK) says: “7: let’s go see Taxi Driver”

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission
  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver passes “let’s go see Taxi Driver” to application layer
  4. receiver says: “Got it!” (i.e., ACK)
  5. sender hears: “. . . *noise* . . . ”
  6. sender (assuming a NACK) says: “7: let’s go see Taxi Driver”
  7. receiver hears: “7: let’s go see Taxi Driver”
  8. receiver ignores the packet

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission
  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver passes “let’s go see Taxi Driver” to application layer
  4. receiver says: “Got it!” (i.e., ACK)
  5. sender hears: “. . . *noise* . . . ”
  6. sender (assuming a NACK) says: “7: let’s go see Taxi Driver”
  7. receiver hears: “7: let’s go see Taxi Driver”
  8. receiver ignores the packet

- How many bits do we need for the sequence number?

Dealing With Duplicate Packets

- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission

  1. sender says: "7: let's go see Taxi Driver"
  2. receiver hears: "7: let's go see Taxi Driver"
  3. receiver passes "let's go see Taxi Driver" to application layer
  4. receiver says: "Got it!" (i.e., ACK)
  5. sender hears: "...*noise* ..."
  6. sender (assuming a NACK) says: "7: let's go see Taxi Driver"
  7. receiver hears: "7: let's go see Taxi Driver"
  8. receiver ignores the packet

- How many bits do we need for the sequence number?

  - this is a "stop-and-wait" protocol for each packet, so the receiver needs to distinguish between (1) the next packet and (2) the retransmission of the current packet

Dealing With Duplicate Packets

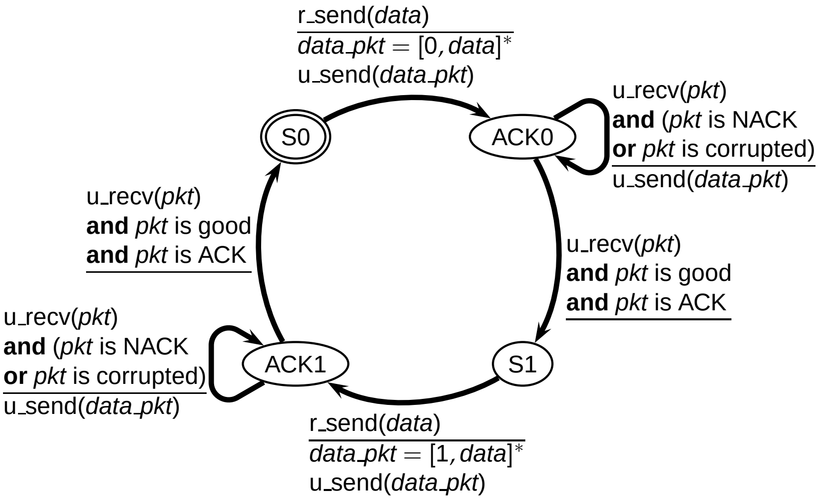
- The sender adds a *sequence number* to each packet so that the receiver can determine whether a packet is a retransmission

  1. sender says: "7: let's go see Taxi Driver"
  2. receiver hears: "7: let's go see Taxi Driver"
  3. receiver passes "let's go see Taxi Driver" to application layer
  4. receiver says: "Got it!" (i.e., ACK)
  5. sender hears: "...*noise* ..."
  6. sender (assuming a NACK) says: "7: let's go see Taxi Driver"
  7. receiver hears: "7: let's go see Taxi Driver"
  8. receiver ignores the packet

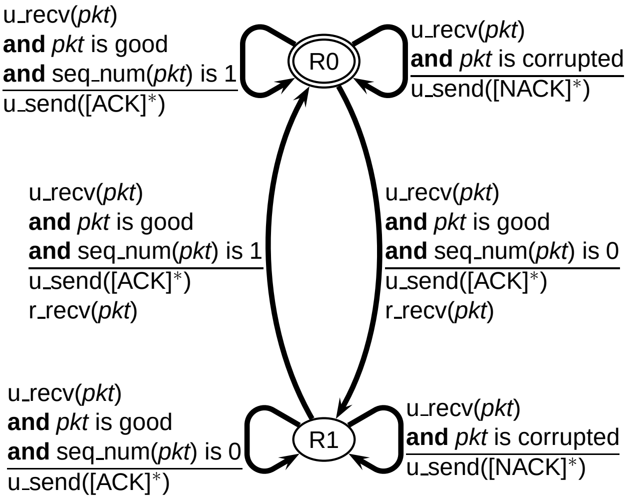
- How many bits do we need for the sequence number?

  - this is a "stop-and-wait" protocol for each packet, so the receiver needs to distinguish between (1) the next packet and (2) the retransmission of the current packet
  - so, one bit is sufficient

Using Sequence Numbers: Sender



Using Sequence Numbers: Receiver



Better Use of ACKs

- Do we really need both ACKs and NACKs?

Better Use of ACKs

- Do we really need both ACKs and NACKs?

- Idea: now that we have sequence numbers, the receiver can convey the semantics of a NACK by sending an ACK for the last good packet it received

Better Use of ACKs

- Do we really need both ACKs and NACKs?

- Idea: now that we have sequence numbers, the receiver can convey the semantics of a NACK by sending an ACK for the last good packet it received

  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver says: “Got it!”
  4. sender hears: “Got it!”
  5. sender says: “8: let’s meet at 8:00PM”
  6. receiver hears: “. . . *noise* . . . ”

Better Use of ACKs

- Do we really need both ACKs and NACKs?

- Idea: now that we have sequence numbers, the receiver can convey the semantics of a NACK by sending an ACK for the last good packet it received

  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver says: “Got it!”
  4. sender hears: “Got it!”
  5. sender says: “8: let’s meet at 8:00PM”
  6. receiver hears: “. . . *noise* . . . ”
  7. receiver now says: “Got 7” (instead of saying “Please, resend”)
  8. sender hears: “Got 7”

Better Use of ACKs

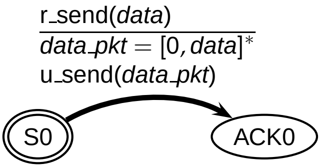
- Do we really need both ACKs and NACKs?

- Idea: now that we have sequence numbers, the receiver can convey the semantics of a NACK by sending an ACK for the last good packet it received

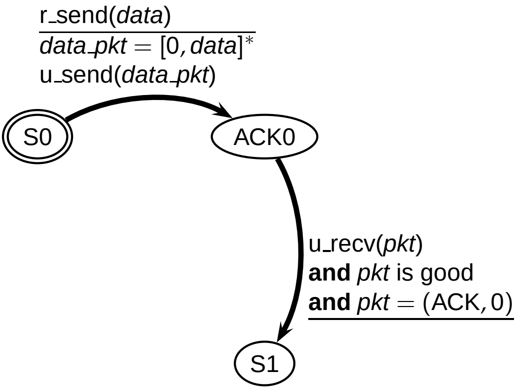
  1. sender says: “7: let’s go see Taxi Driver”
  2. receiver hears: “7: let’s go see Taxi Driver”
  3. receiver says: “Got it!”
  4. sender hears: “Got it!”
  5. sender says: “8: let’s meet at 8:00PM”
  6. receiver hears: “. . . *noise* . . . ”
  7. receiver now says: “Got 7” (instead of saying “Please, resend”)
  8. sender hears: “Got 7”
  9. sender knows that the current message is 8, and therefore repeats: “8: let’s meet at 8:00PM”

ACK-Only Protocol: Sender

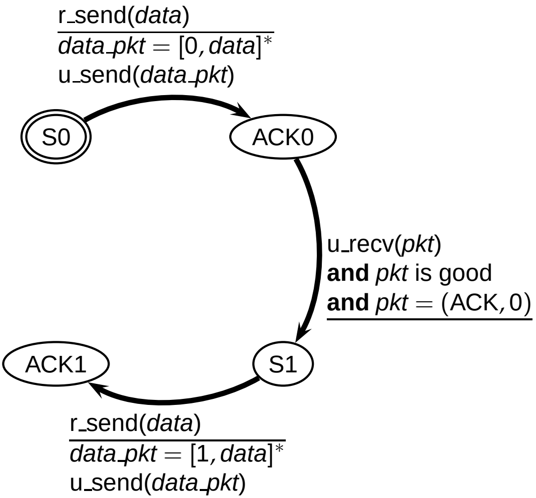
ACK-Only Protocol: Sender



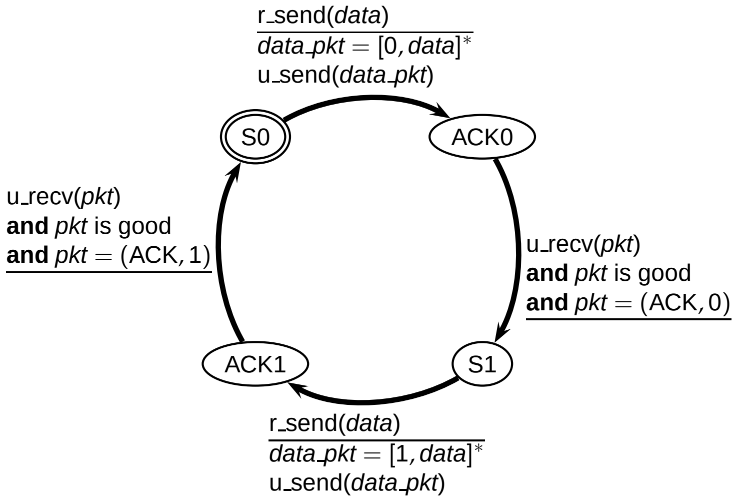
ACK-Only Protocol: Sender



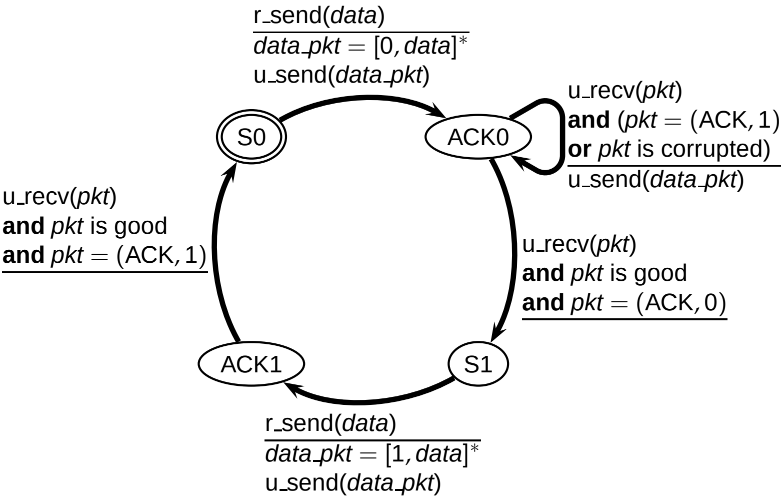
ACK-Only Protocol: Sender



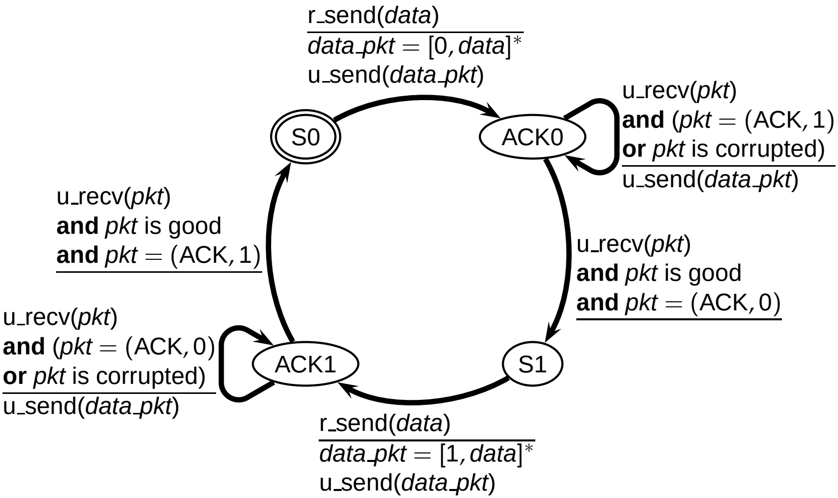
ACK-Only Protocol: Sender



ACK-Only Protocol: Sender

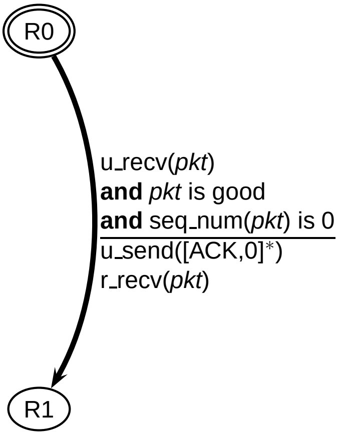


ACK-Only Protocol: Sender

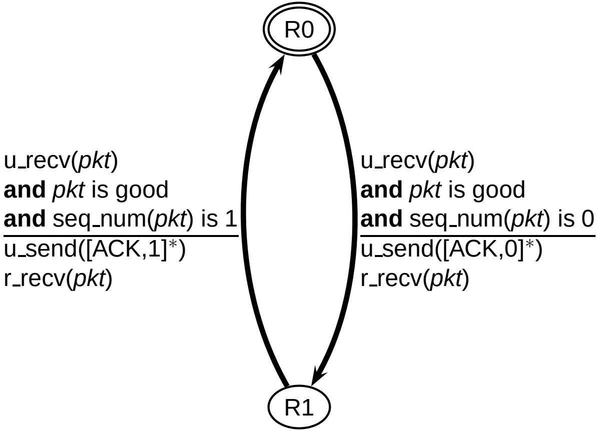


ACK-Only Protocol: Receiver

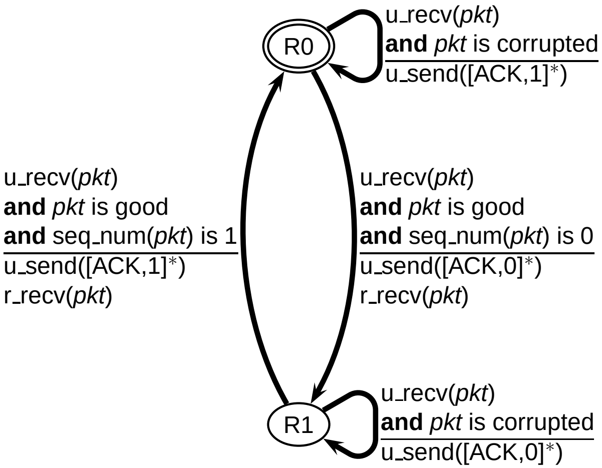
ACK-Only Protocol: Receiver



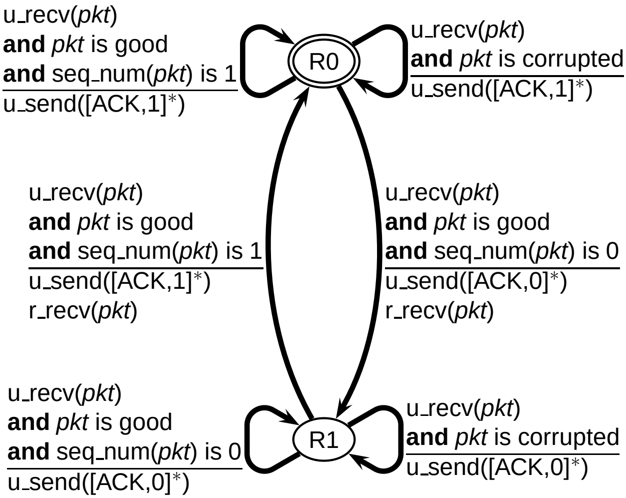
ACK-Only Protocol: Receiver



ACK-Only Protocol: Receiver



ACK-Only Protocol: Receiver



Summary of Principles and Techniques

Summary of Principles and Techniques

- *Error detection codes* (checksums) can be used to detect transmission errors

Summary of Principles and Techniques

- *Error detection codes* (checksums) can be used to detect transmission errors

- *Retransmission* allows us to recover from transmission errors

Summary of Principles and Techniques

- *Error detection codes* (checksums) can be used to detect transmission errors

- *Retransmission* allows us to recover from transmission errors

- *ACKs and NACKs* give feedback to the sender
  - ACKs and NACKs are also “protected” with an error-detection code

Summary of Principles and Techniques

- *Error detection codes* (checksums) can be used to detect transmission errors

- *Retransmission* allows us to recover from transmission errors

- *ACKs and NACKs* give feedback to the sender
  - ACKs and NACKs are also “protected” with an error-detection code
  - corrupted ACKs are interpreded as NACKs, possibly generating duplicate segments

Summary of Principles and Techniques

- *Error detection codes* (checksums) can be used to detect transmission errors

- *Retransmission* allows us to recover from transmission errors

- *ACKs and NACKs* give feedback to the sender
  - ACKs and NACKs are also “protected” with an error-detection code
  - corrupted ACKs are interpreded as NACKs, possibly generating duplicate segments

- *Sequence numbers* allow the receiver to ignore duplicate data segments

Lossy And Noisy Channel

Lossy And Noisy Channel

- Reliable transport protocol over a network that may
  - introduce *bit errors*
  - loose packets

Lossy And Noisy Channel

- Reliable transport protocol over a network that may
  - introduce *bit errors*
  - loose packets

- How do people deal with such situations?
  (Think of radio transmissions over a noisy and shared medium.
  Also, think about what we just did for noisy channels)

Lossy And Noisy Channel

- Reliable transport protocol over a network that may
  - introduce *bit errors*
  - loose packets

- How do people deal with such situations?
  (Think of radio transmissions over a noisy and shared medium.
  Also, think about what we just did for noisy channels)

- *Detection:* the receiver and/or the sender must be able to
  determine that a packet was lost (how?)

Lossy And Noisy Channel

- Reliable transport protocol over a network that may
  - introduce *bit errors*
  - loose packets

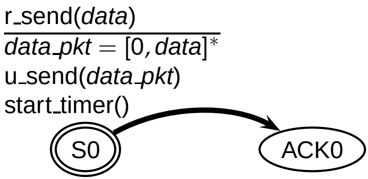
- How do people deal with such situations?
  (Think of radio transmissions over a noisy and shared medium.
  Also, think about what we just did for noisy channels)

- *Detection:* the receiver and/or the sender must be able to determine that a packet was lost (how?)

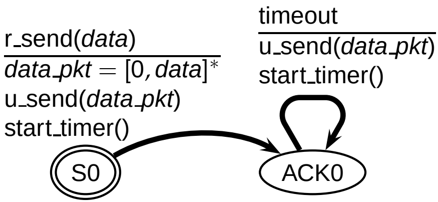
- *ACKs, retransmission, and sequence numbers:* lost packets can be easily treated as corrupted packets

Sender Using Timeouts

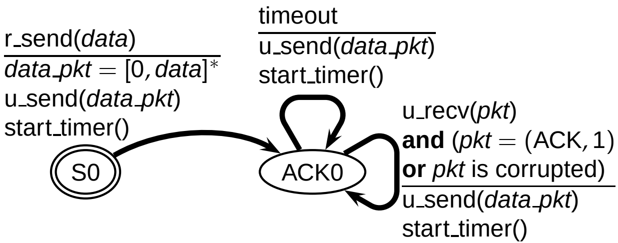
Sender Using Timeouts



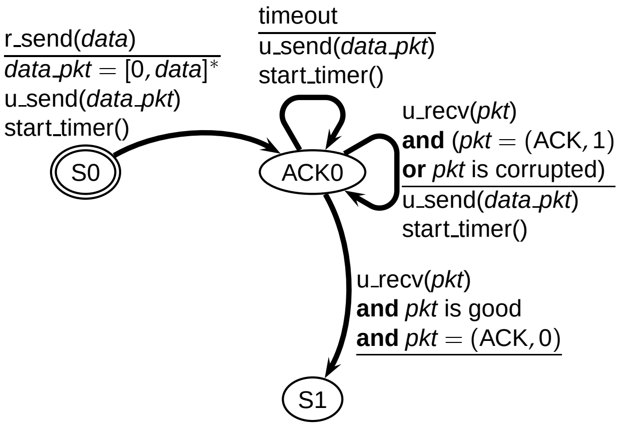
Sender Using Timeouts



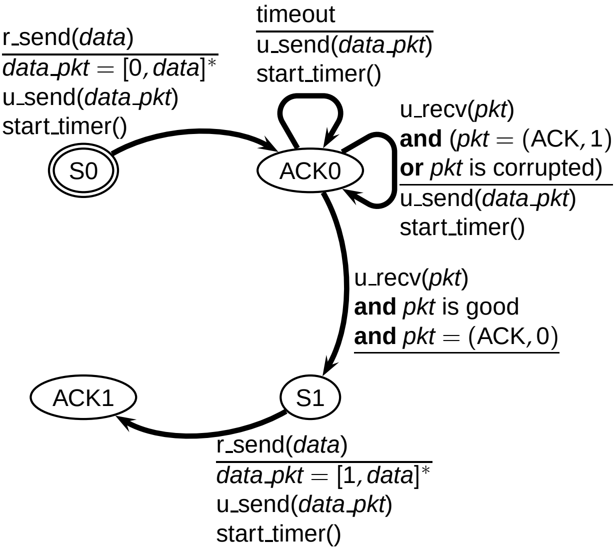
Sender Using Timeouts



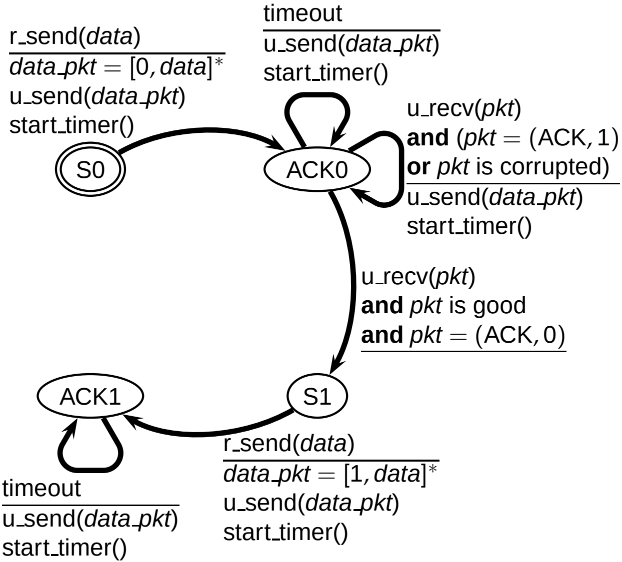
Sender Using Timeouts



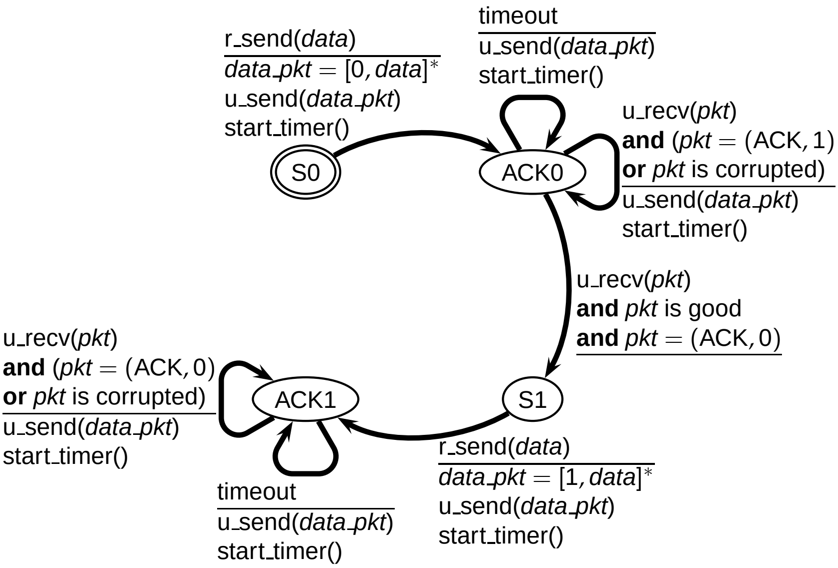
Sender Using Timeouts



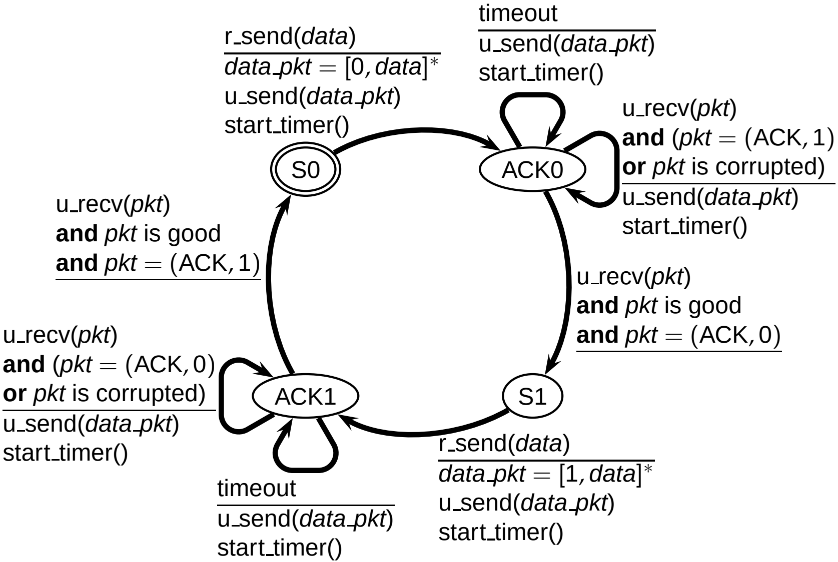
Sender Using Timeouts



Sender Using Timeouts



Sender Using Timeouts



Part II

Efficient and Reliable Streams

Quantifying Data Transfer

- How do we measure the “speed” and “capacity” of a network connection?

Quantifying Data Transfer

- How do we measure the “speed” and “capacity” of a network connection?
- Intuition
  - water moves in a pipeline
  - cars move on a road

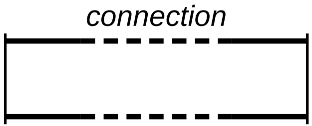
Quantifying Data Transfer

- How do we measure the "speed" and "capacity" of a network connection?
- Intuition
  - water moves in a pipeline
  - cars move on a road
- *Latency*
  - the time it takes for *one bit* to go through the connection (from one end to the other)

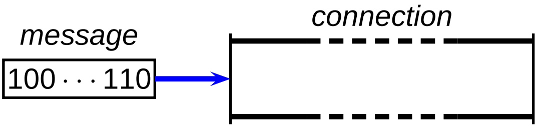
Quantifying Data Transfer

- How do we measure the “speed” and “capacity” of a network connection?
- Intuition
  - water moves in a pipeline
  - cars move on a road
- *Latency*
  - the time it takes for *one bit* to go through the connection (from one end to the other)
- *Throughput*
  - the amount of information that can get into (or out of) the connection in a time unit
  - at “steady-state” we assume zero accumulation of traffic, therefore the input throughput is the same as the output throughput

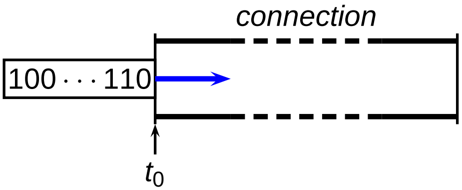
Latency and Throughput



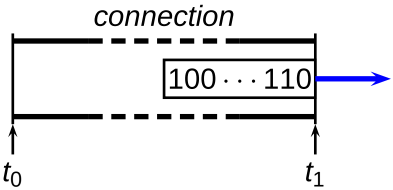
Latency and Throughput



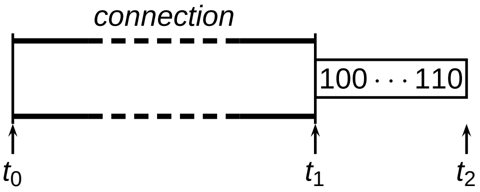
Latency and Throughput



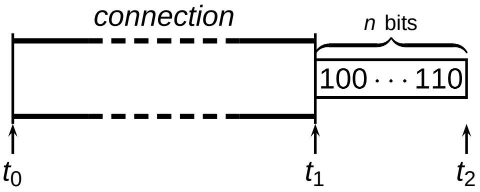
Latency and Throughput



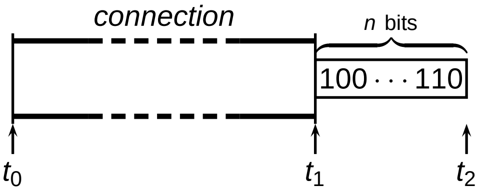
Latency and Throughput



Latency and Throughput

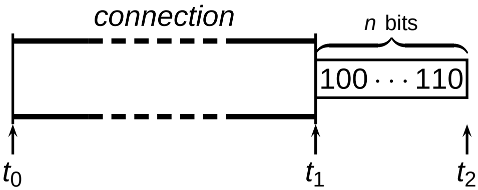


Latency and Throughput



Latency $\quad L = t_1 - t_0 \quad$ sec

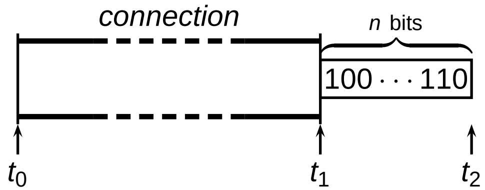
Latency and Throughput



Latency $L = t_1 - t_0$ sec

Throughput $T = \frac{n}{t_2 - t_1}$ bits/sec

Latency and Throughput



Latency	$L = t_1 - t_0$	sec
Throughput	$T = \dfrac{n}{t_2 - t_1}$	bits/sec
Transfer time	$\Delta = L + \dfrac{n}{T}$	sec

Examples

- How long does it take to tranfer a file between, say, Lugano and St. Petersburg?

Examples

- How long does it take to tranfer a file between, say, Lugano and St. Petersburg?

- How big is this file? And *how fast* is our connection?

Examples

- How long does it take to tranfer a file between, say, Lugano and St. Petersburg?

- How big is this file? And *how fast* is our connection?

  E.g., a (short) e-mail message

$$\begin{aligned} S &= 4\text{Kb} \\ L &= 500\text{ms} \\ T &= 1\text{Mb/s} \\ \Delta &= ? \end{aligned}$$

Examples

- How long does it take to tranfer a file between, say, Lugano and St. Petersburg?

- How big is this file? And *how fast* is our connection?

  E.g., a (short) e-mail message

$$
\begin{aligned}
S &= \text{4Kb} \\
L &= \text{500ms} \\
T &= \text{1Mb/s} \\
\Delta &= \text{500ms} + \text{4ms} = \text{504ms}
\end{aligned}
$$

Examples

- How about a big file? (E.g., a CD)

Examples

- How about a big file? (E.g., a CD)

$$\begin{aligned} S &= 400Mb \\ L &= 500ms \\ T &= 1Mb/s \\ \Delta &= ? \end{aligned}$$

Examples

- How about a big file? (E.g., a CD)

$$
\begin{aligned}
S &= 400Mb \\
L &= 500ms \\
T &= 1Mb/s \\
\Delta &= 500ms + 400s = 400.5s = 6'40''
\end{aligned}
$$

Examples

- How about a big file? (E.g., a CD)

$$\begin{aligned} S &= 400Mb \\ L &= 500ms \\ T &= 1Mb/s \\ \Delta &= 500ms + 400s = 400.5s = 6'40'' \end{aligned}$$

- How about a bigger file? (E.g., 10 DVDs)

Examples

- How about a big file? (E.g., a CD)

$$\begin{aligned} S &= 400Mb \\ L &= 500ms \\ T &= 1Mb/s \\ \Delta &= 500ms + 400s = 400.5s = 6'40'' \end{aligned}$$

- How about a bigger file? (E.g., 10 DVDs)

$$\begin{aligned} S &= 40Gb \\ L &= 500ms \\ T &= 1Mb/s \\ \Delta &= ? \end{aligned}$$

Examples

- How about a big file? (E.g., a CD)

$$
\begin{aligned}
S &= 400Mb \\
L &= 500ms \\
T &= 1Mb/s \\
\Delta &= 500ms + 400s = 400.5s = 6'40''
\end{aligned}
$$

- How about a bigger file? (E.g., 10 DVDs)

$$
\begin{aligned}
S &= 40Gb \\
L &= 500ms \\
T &= 1Mb/s \\
\Delta &= \epsilon + 40000s = 11h\ 6'40''
\end{aligned}
$$

Examples

- How about *flying* to St. Petersburgh?

Examples

- How about *flying* to St. Petersburgh?
  - assuming you can carry more or less 100 DVDs in your backpack
  - assuming it takes you four seconds to take the DVDs out of your backpack

Examples

- How about *flying* to St. Petersburgh?
  - assuming you can carry more or less 100 DVDs in your backpack
  - assuming it takes you four seconds to take the DVDs out of your backpack

$$\begin{aligned} S &= 40Gb \\ L &= ? \\ T &= \\ \Delta &= \end{aligned}$$

Examples

- How about *flying* to St. Petersburgh?
  - assuming you can carry more or less 100 DVDs in your backpack
  - assuming it takes you four seconds to take the DVDs out of your backpack

$$
\begin{aligned}
S &= 40Gb \\
L &= 6h \\
T &= ? \\
\Delta &=
\end{aligned}
$$

Examples

- How about *flying* to St. Petersburgh?
  - assuming you can carry more or less 100 DVDs in your backpack
  - assuming it takes you four seconds to take the DVDs out of your backpack

$$\begin{aligned} S &= 40Gb \\ L &= 6h \\ T &= 100Gb/s \\ \Delta &= ? \end{aligned}$$

Examples

- How about *flying* to St. Petersburgh?
  - assuming you can carry more or less 100 DVDs in your backpack
  - assuming it takes you four seconds to take the DVDs out of your backpack

$$
\begin{aligned}
S &= 40Gb \\
L &= 6h \\
T &= 100Gb/s \\
\Delta &= 6h
\end{aligned}
$$

Examples

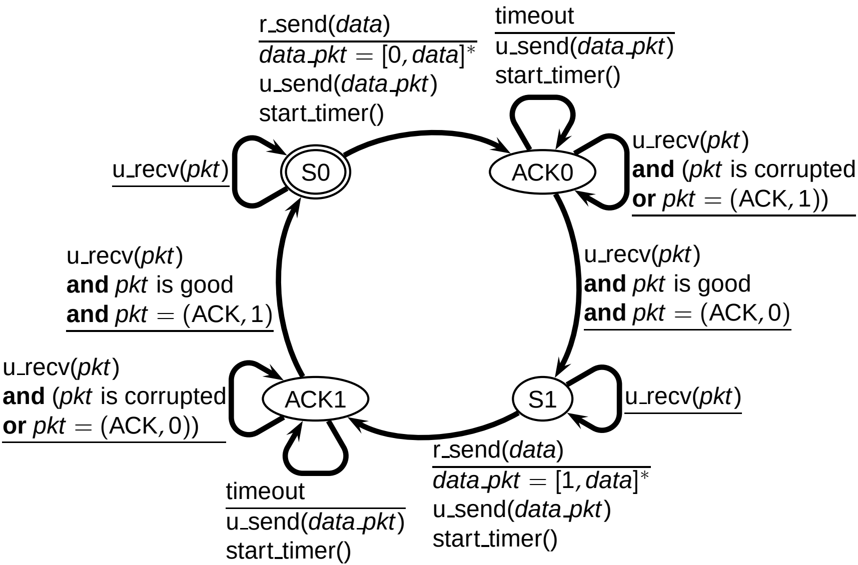
- How about *flying* to St. Petersburgh?
  - assuming you can carry more or less 100 DVDs in your backpack
  - assuming it takes you four seconds to take the DVDs out of your backpack

$$
\begin{aligned}
S &= 40Gb \\
L &= 6h \\
T &= 100Gb/s \\
\Delta &= 6h
\end{aligned}
$$

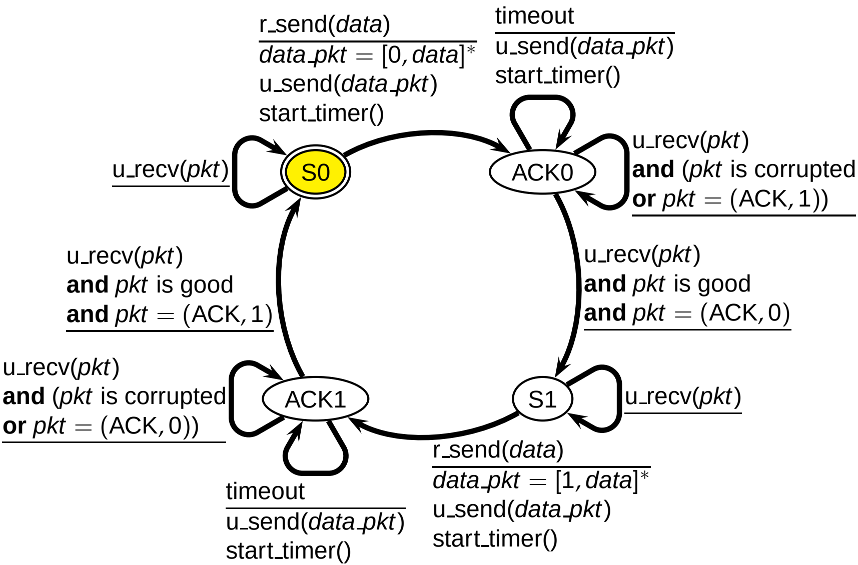
If you need to transfer 10 DVDs from Lugano to St. Petersburg and time is of the essence (and you have plenty of money)... then you're better off talking a plane rather than using the Internet

Back to Reliable Data Tranfer

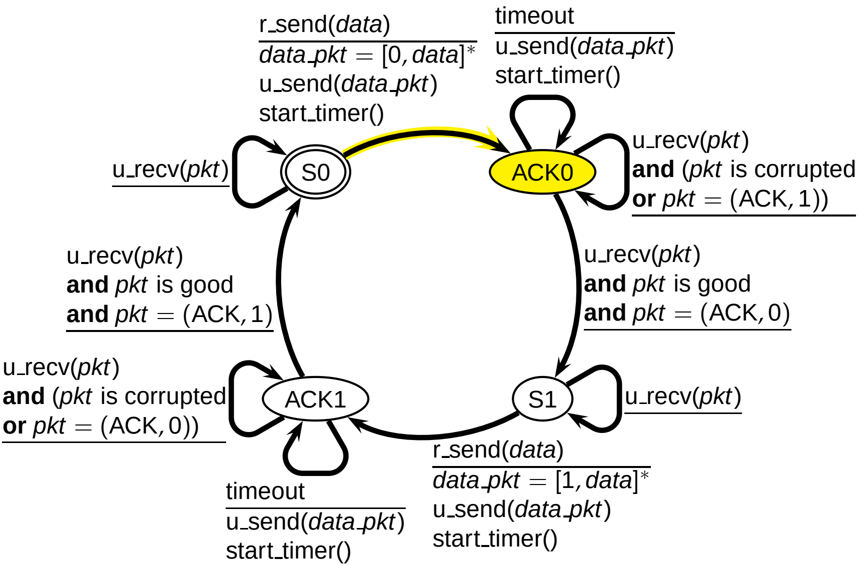
Back to Reliable Data Tranfer



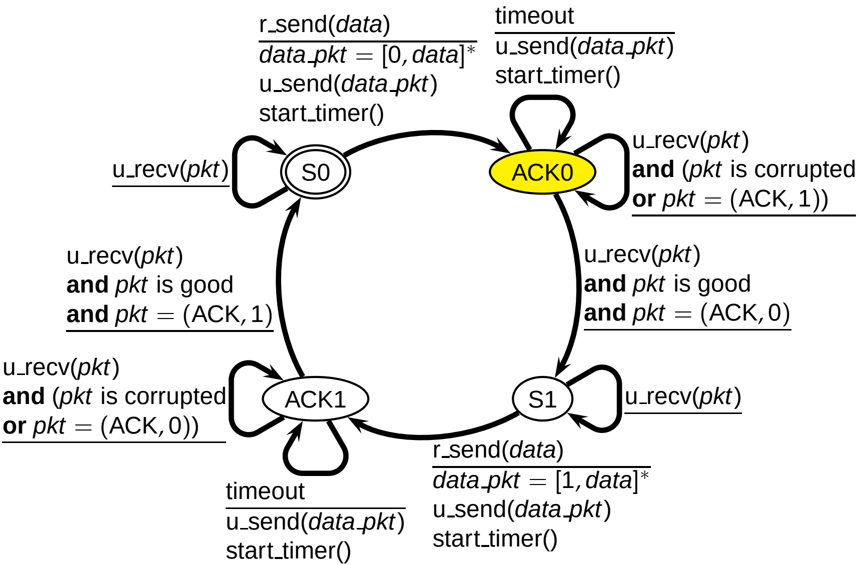
Back to Reliable Data Tranfer



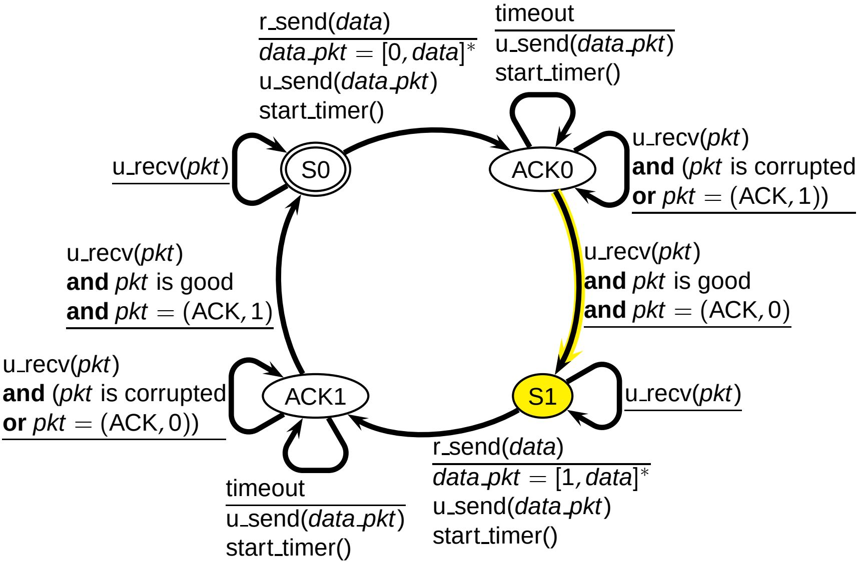
Back to Reliable Data Tranfer



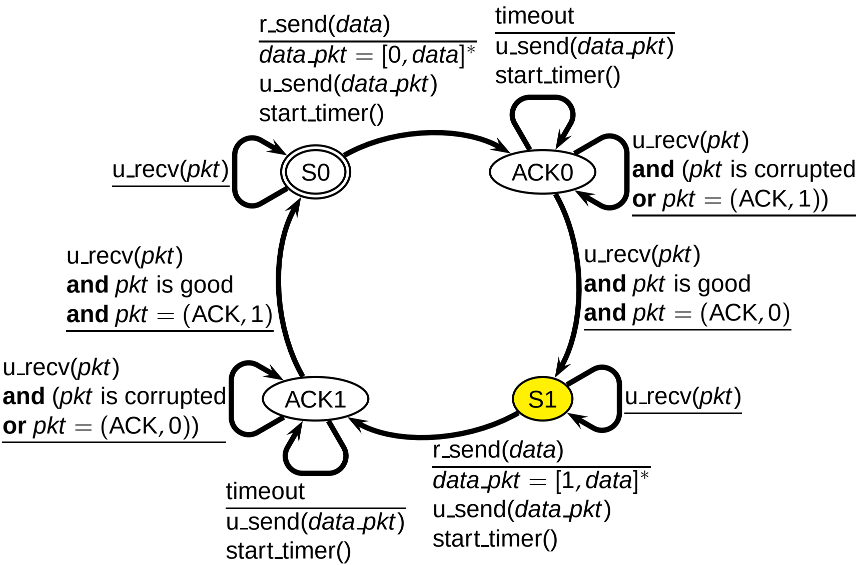
Back to Reliable Data Tranfer



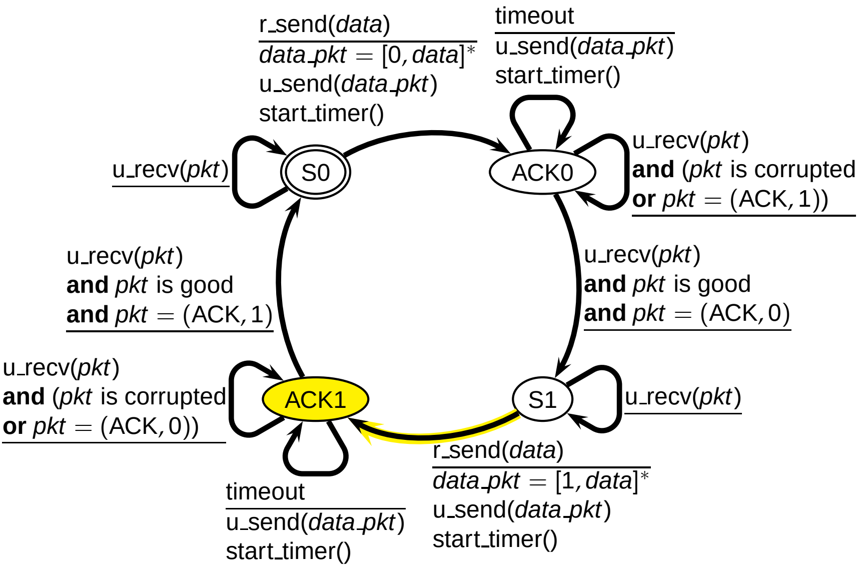
Back to Reliable Data Tranfer



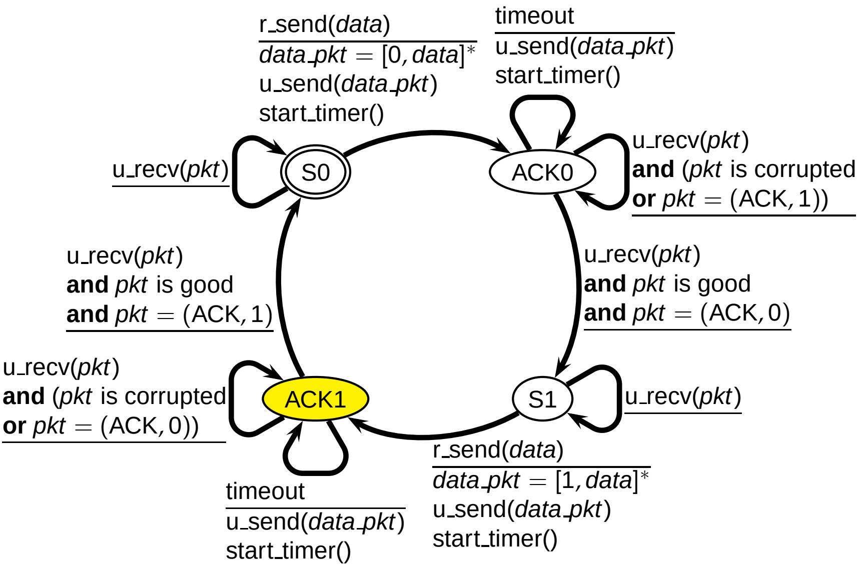
Back to Reliable Data Tranfer



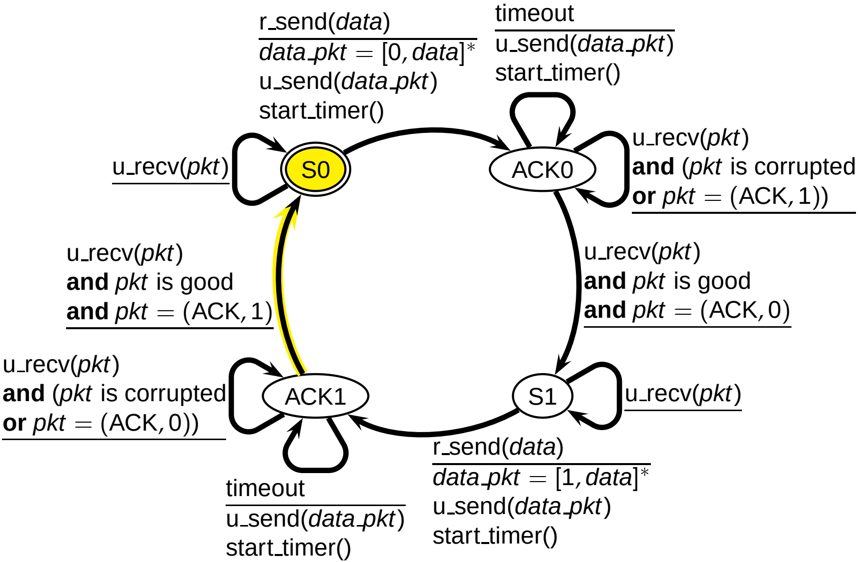
Back to Reliable Data Tranfer



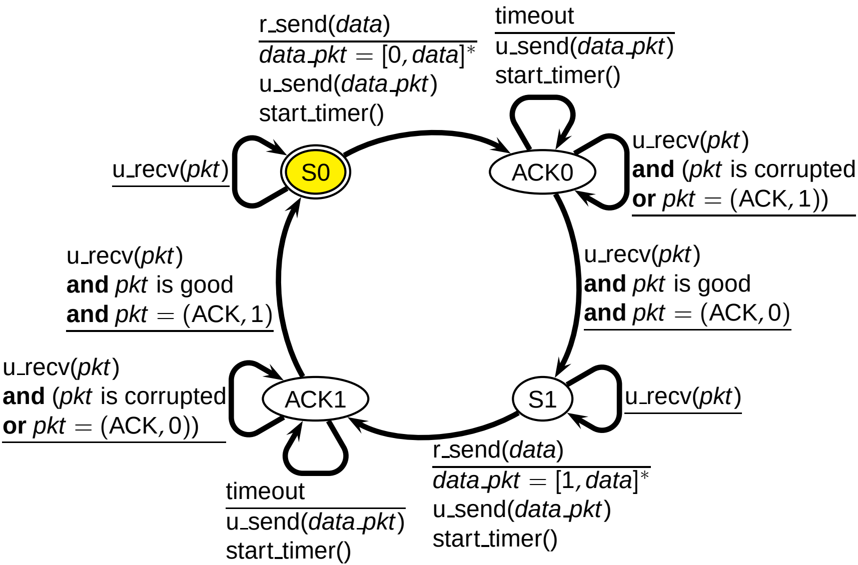
Back to Reliable Data Tranfer



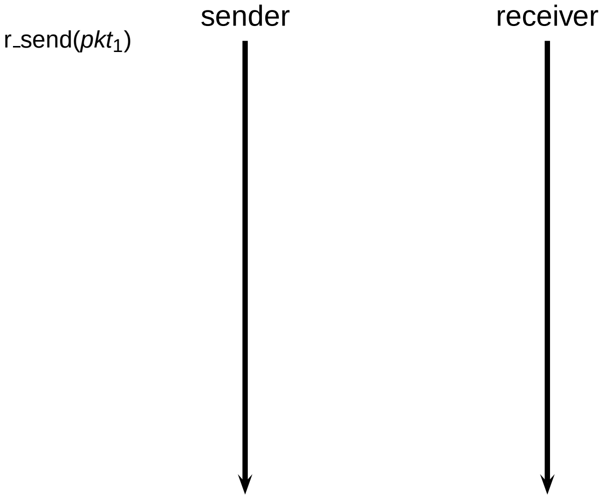
Back to Reliable Data Tranfer



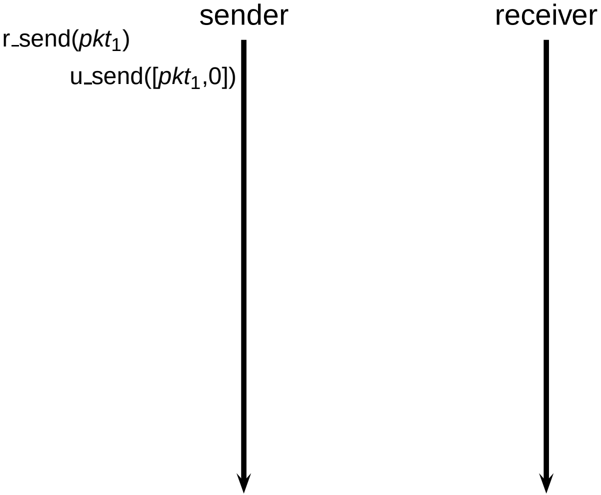
Back to Reliable Data Tranfer



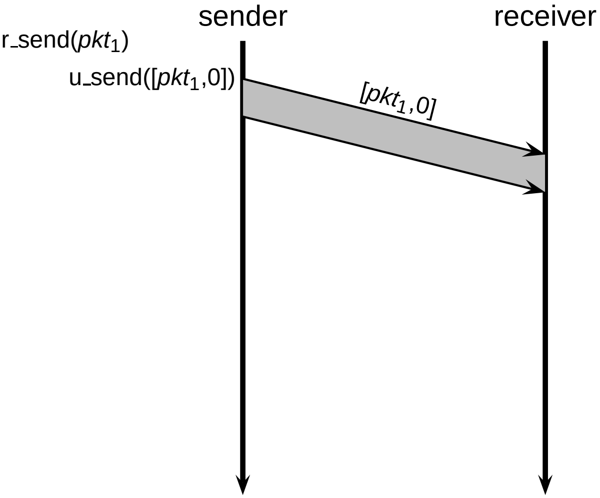
Network Usage



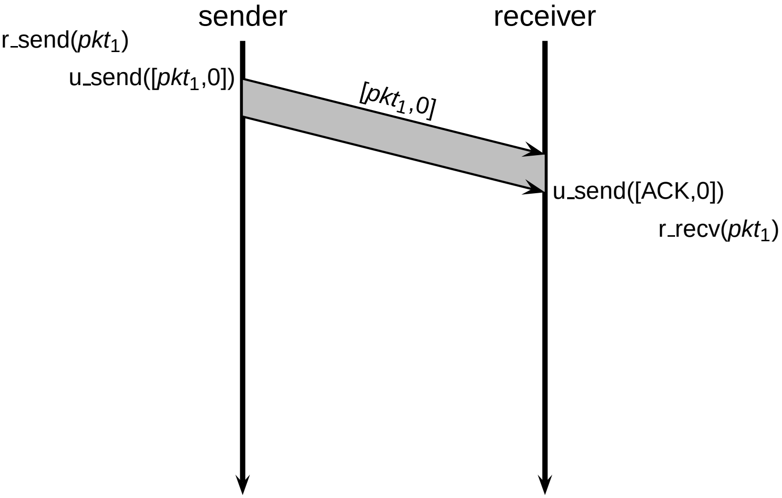
Network Usage



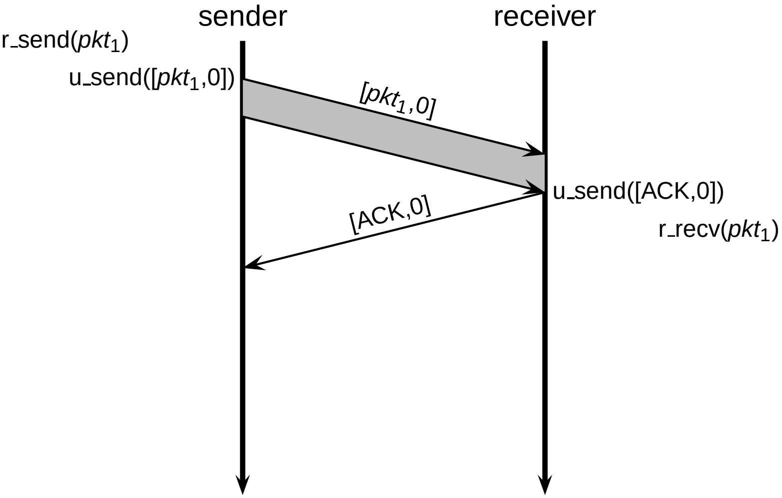
Network Usage



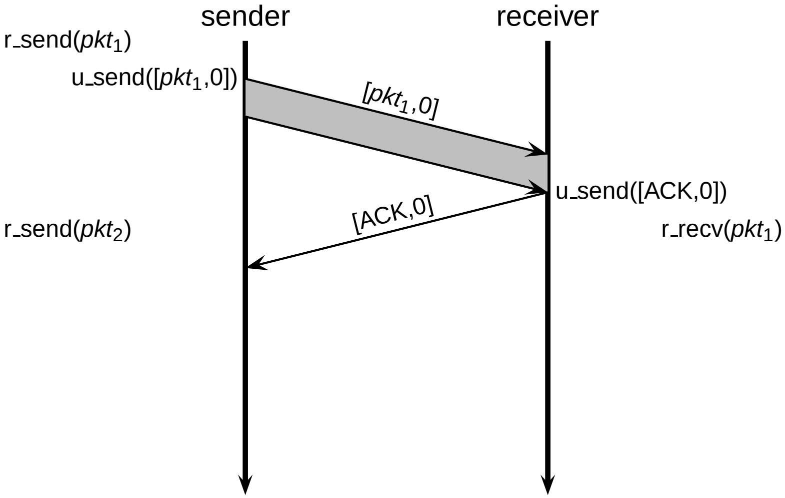
Network Usage



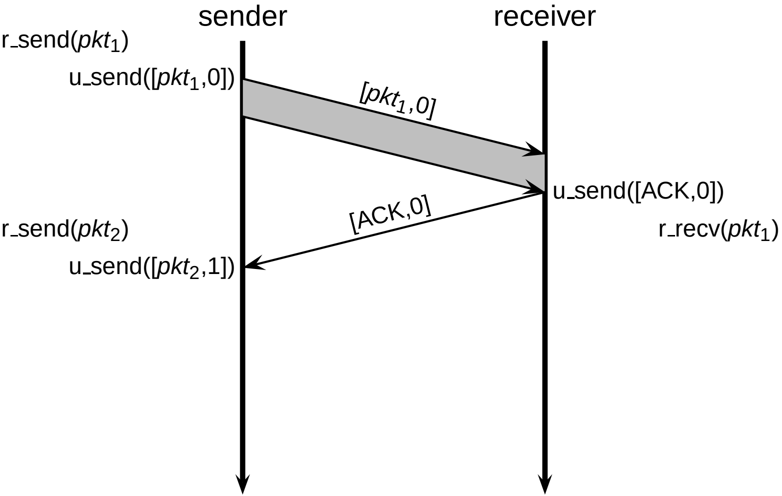
Network Usage



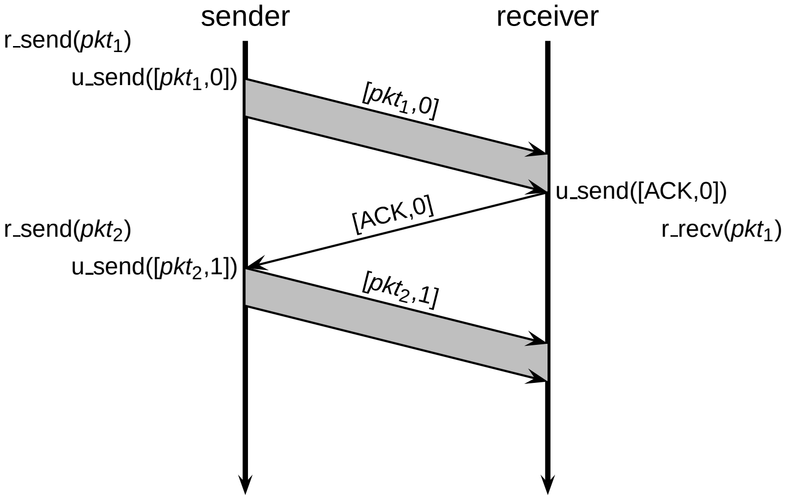
Network Usage



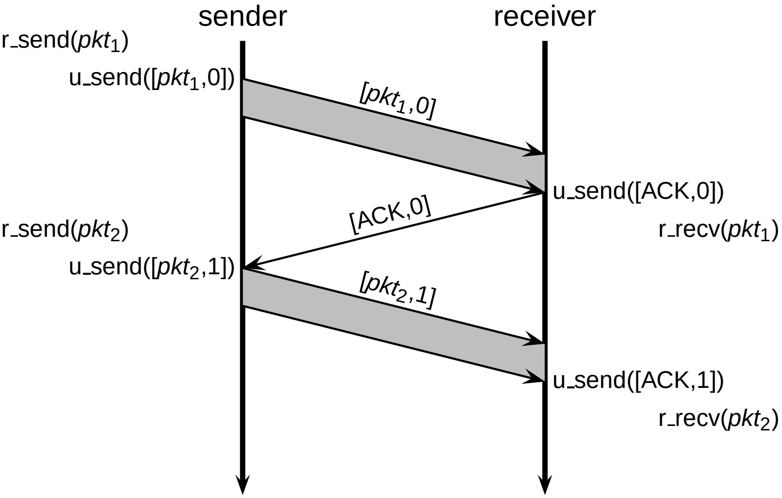
Network Usage



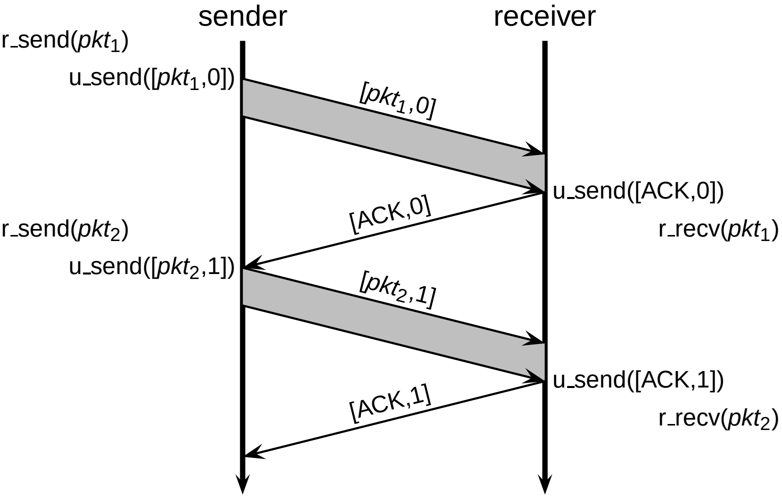
Network Usage



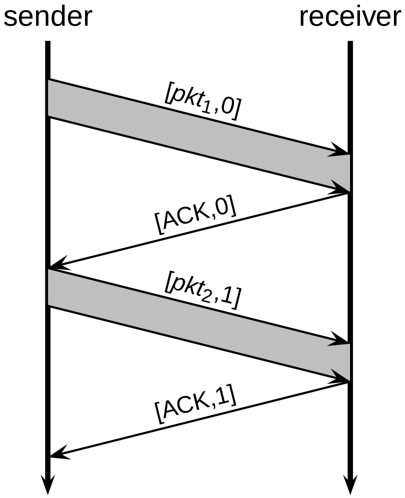
Network Usage



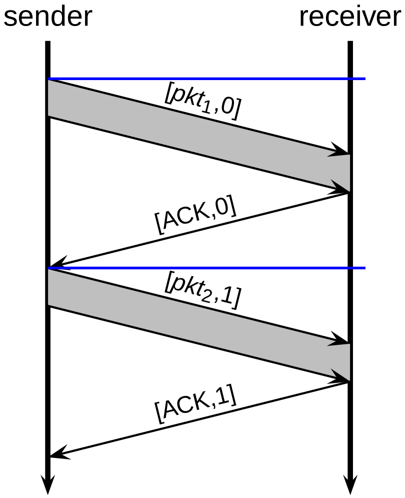
Network Usage



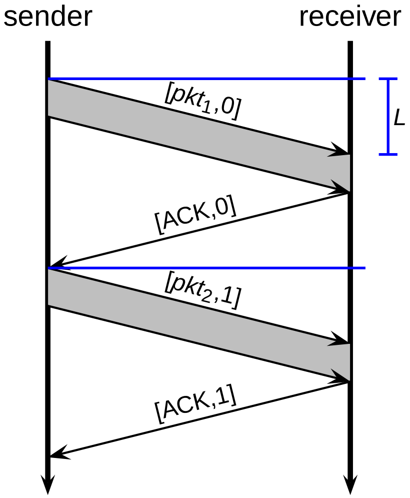
Network Usage



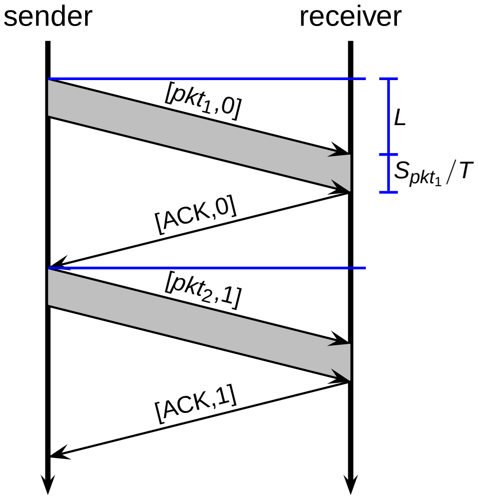
Network Usage



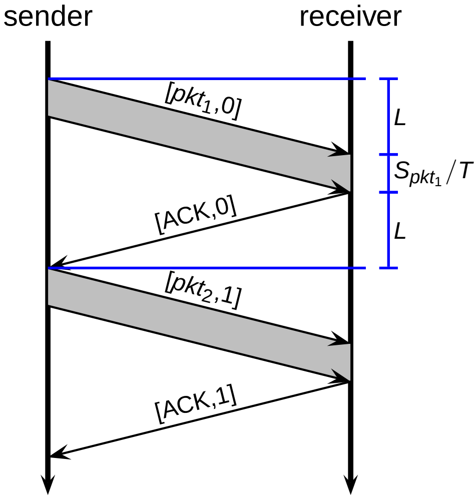
Network Usage



Network Usage



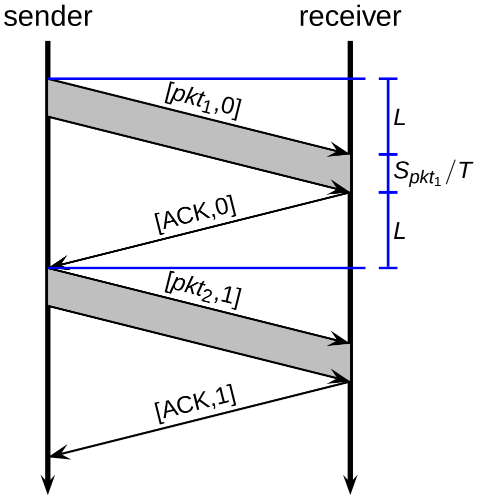
Network Usage



Network Usage

utilization factor

$$U = \frac{S_{pkt}/T}{2L+S_{pkt}/T}$$

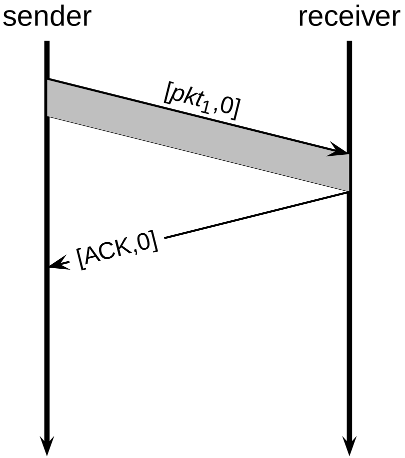


Improving Network Usage

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

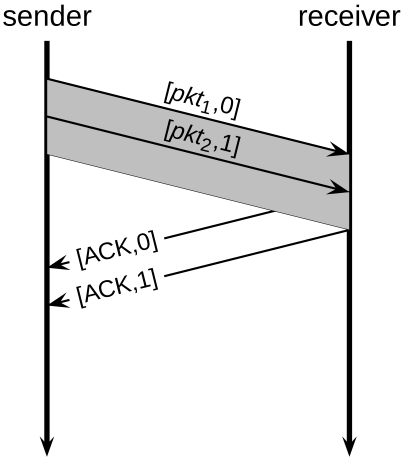
Improving Network Usage

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



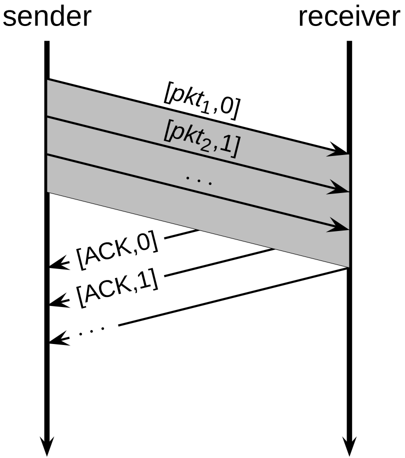
Improving Network Usage

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



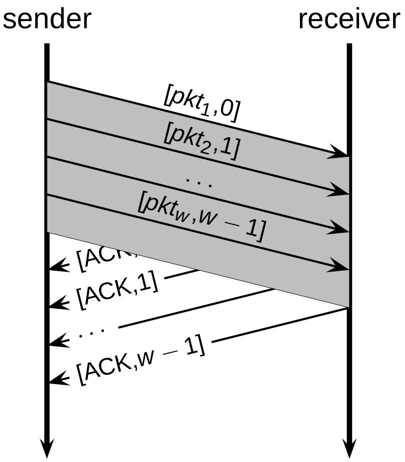
Improving Network Usage

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



Improving Network Usage

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



Go-Back-N

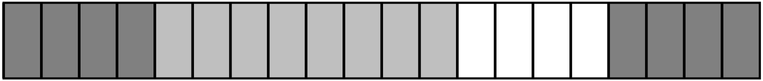
- *Idea:* *the sender transmits multiple packets without waiting for an acknowledgement*

Go-Back-N

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

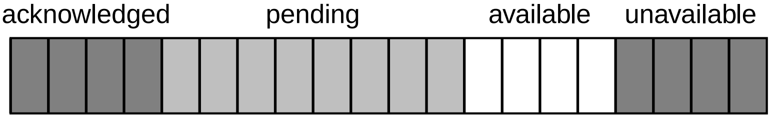
Go-Back-N

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

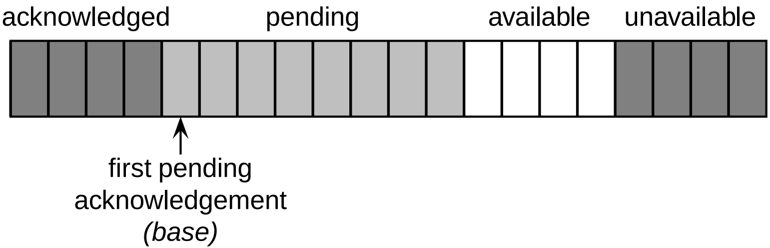
- *Idea: the sender transmits multiple packets without waiting for an acknowledgement*
- The sender has up to W unacknowledged packets in the pipeline
  - the sender's state machine gets very complex
  - we represent the sender's state with its queue of acknowledgements



Go-Back-N

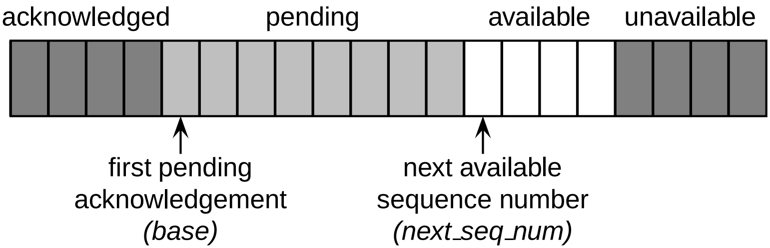
- *Idea: the sender transmits multiple packets without waiting for an acknowledgement*

- The 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



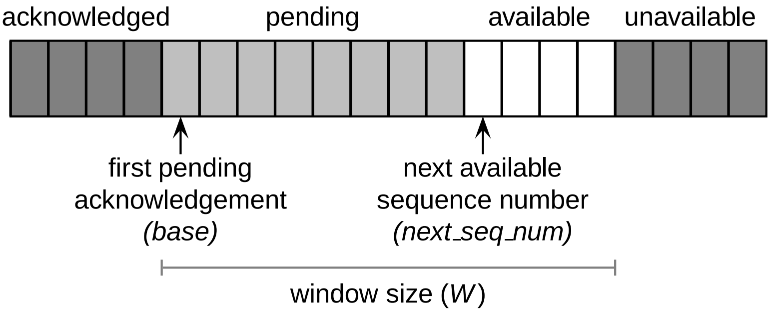
Go-Back-N

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

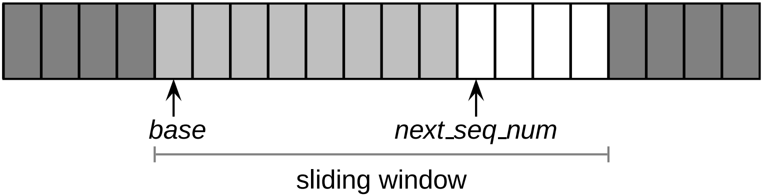


Go-Back-N

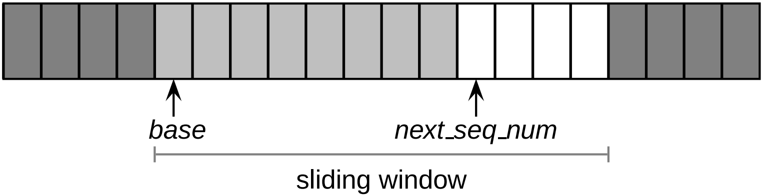
- *Idea: the sender transmits multiple packets without waiting for an acknowledgement*
- The sender has up to W unacknowledged packets in the pipeline
  - the sender's state machine gets very complex
  - we represent the sender's state with its queue of acknowledgements



Sliding Window Protocol: Sender

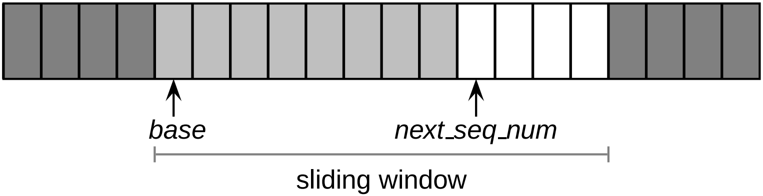


Sliding Window Protocol: Sender



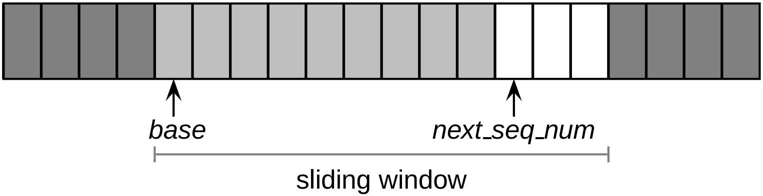
- r_send(pkt_1)

Sliding Window Protocol: Sender



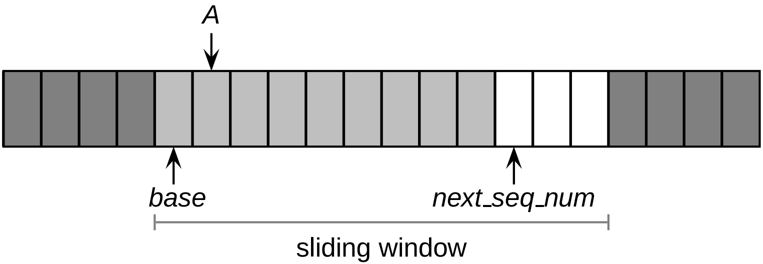
- r_send(pkt_1)
  - u_send([pkt_1,*next_seq_num*])

Sliding Window Protocol: Sender



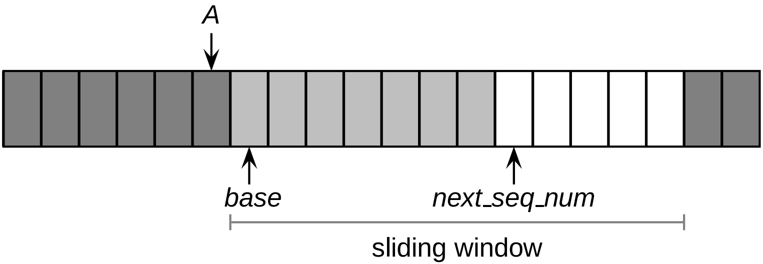
- r_send(pkt_1)
  - u_send([pkt_1,*next_seq_num*])
  - $next\_seq\_num \leftarrow next\_seq\_num + 1$

Sliding Window Protocol: Sender



- r_send(pkt_1)
  - u_send([pkt_1,*next_seq_num*])
  - $next\_seq\_num \leftarrow next\_seq\_num + 1$
- u_recv([ACK,*A*])

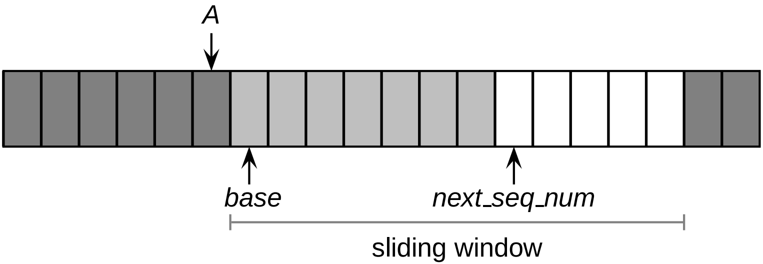
Sliding Window Protocol: Sender



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

- u_recv([ACK,A])
  - $base \leftarrow A + 1$

Sliding Window Protocol: Sender



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

- u_recv([ACK,A])
  - $base \leftarrow A + 1$
  - 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

- $\dfrac{\textit{init}}{\begin{array}{l}\textit{base} \leftarrow 1 \\ \textit{next\_seq\_num} \leftarrow 1\end{array}}$

Sliding Window Protocol: Sender

- *init*

  $base \leftarrow 1$
  $next\_seq\_num \leftarrow 1$

- r_send(*data*)

  if ($next\_seq\_num < base + W$) {
     $pkt[next\_seq\_num] \leftarrow [next\_seq\_num, data]^*$
     u_send($pkt[next\_seq\_num]$)
     if ($next\_seq\_num = base$) {
        start_timer()
     }
     $next\_seq\_num \leftarrow next\_seq\_num + 1$
  } **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 $\leftarrow$ *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

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

Sliding Window Protocol: Receiver

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

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

Sliding Window Protocol: Receiver

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

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

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

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*

Comments

- Concepts
  - *sequence numbers*
  - *sliding window*
  - *cumulative acknowledgements*
  - *checksums*, *timeouts*, and *sender-initiated retransmission*
- Advantages: *simple*
  - the sender maintains *two counters* and a *one timer*
  - the receiver maintains *one counter*

Comments

- Concepts
  - *sequence numbers*
  - *sliding window*
  - *cumulative acknowledgements*
  - *checksums*, *timeouts*, and *sender-initiated retransmission*
- Advantages: *simple*
  - the sender maintains *two counters* and a *one timer*
  - the receiver maintains *one counter*
- Disadvantages: *not optimal*, *not adaptive*

Comments

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

- Advantages: *simple*
  - the sender maintains *two counters* and a *one timer*
  - the receiver maintains *one counter*

- 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*
  - the sender maintains *two counters* and a *one timer*
  - the receiver maintains *one counter*
- 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

$$
\begin{aligned}
S &= \textit{stream} \\
L &= 500ms \\
T &= 1Mb/s \\
W &= ?
\end{aligned}
$$

Performance Analysis

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

$$
\begin{aligned}
S &= \mathit{stream} \\
L &= 500ms \\
T &= 1Mb/s \\
W &= ?
\end{aligned}
$$

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

$$
\begin{aligned}
S_{pkr} &= 1Kb \\
L &= 500ms \\
T &= 1Mb/s \\
W &= \frac{2L \times T}{S_{pkt}} = 1000
\end{aligned}
$$

Performance Analysis

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

Performance Analysis

- The RTT–throughput product $(2L \times T)$ is the crucial factor
  - $W \times S_{pkt} \leq 2L \times T$
    - why $W \times S_{pkt} > 2L \times T$ doesn't make much sense?

Performance Analysis

- The RTT–throughput product $(2L \times T)$ is the crucial factor
  - $W \times S_{pkt} \leq 2L \times T$
    - why $W \times S_{pkt} > 2L \times T$ doesn't make much sense?
  - maximum channel utilization when $W \times S_{pkt} = 2L \times T$
  - $2L \times T$ 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

$$
\begin{aligned}
S_{pkr} &= 1Kb \\
L &= 500ms \\
T &= 1Mb/s \\
W &= \frac{T \times L}{S_{pkt}} = 1000
\end{aligned}
$$

Problems with Go-Back-N

- Let's consider a fully utilized connection

$$\begin{aligned} S_{pkr} &= 1Kb \\ L &= 500ms \\ T &= 1Mb/s \\ W &= \frac{T \times L}{S_{pkt}} = 1000 \end{aligned}$$

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

Problems with Go-Back-N

- Let's consider a fully utilized connection

$$
\begin{aligned}
S_{pkr} &= 1Kb \\
L &= 500ms \\
T &= 1Mb/s \\
W &= \frac{T \times L}{S_{pkt}} = 1000
\end{aligned}
$$

- 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

$$\begin{aligned} S_{pkr} &= 1Kb \\ L &= 500ms \\ T &= 1Mb/s \\ W &= \frac{T \times L}{S_{pkt}} = 1000 \end{aligned}$$

- What happens if the first packet (or acknowledgement) is lost?
- Sender retransmits the entire content of its buffers
  - $W \times S_{pkt} = 2L \times T = 1Mb$
  - 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

$$\begin{aligned} S_{pkr} &= 1Kb \\ L &= 500ms \\ T &= 1Mb/s \\ W &= \frac{T \times L}{S_{pkt}} = 1000 \end{aligned}$$

- What happens if the first packet (or acknowledgement) is lost?
- Sender retransmits the entire content of its buffers
  - $W \times S_{pkt} = 2L \times T = 1Mb$
  - 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

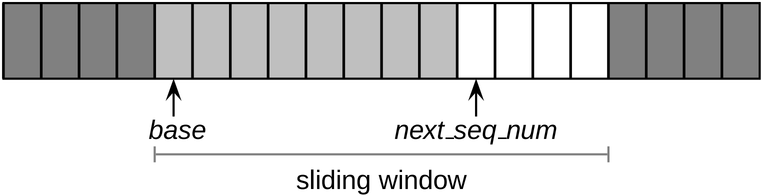
Selective Repeat

- 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

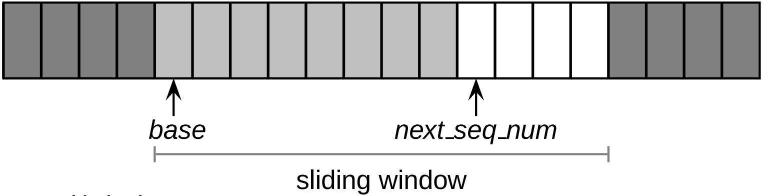
Selective Repeat

- 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

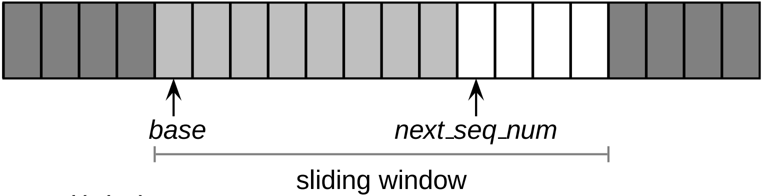


Selective Repeat: Sender



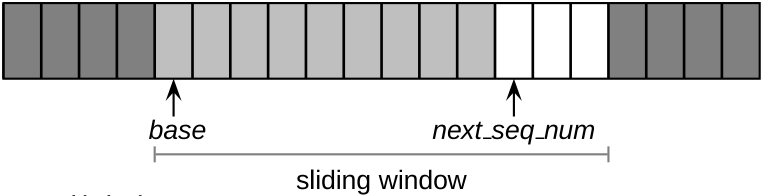
- r_send(pkt_1)

Selective Repeat: Sender



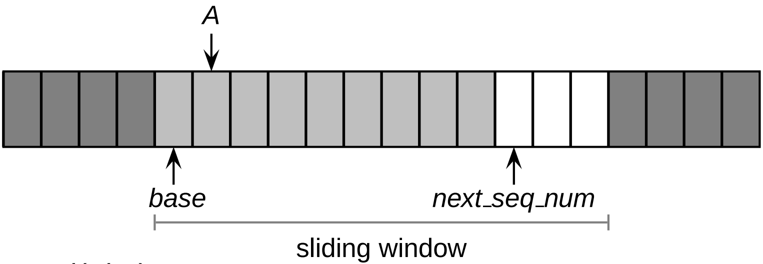
- r_send(pkt_1)
  - u_send([pkt_1,*next_seq_num*])
  - start_timer(*next_seq_num*)

Selective Repeat: Sender



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

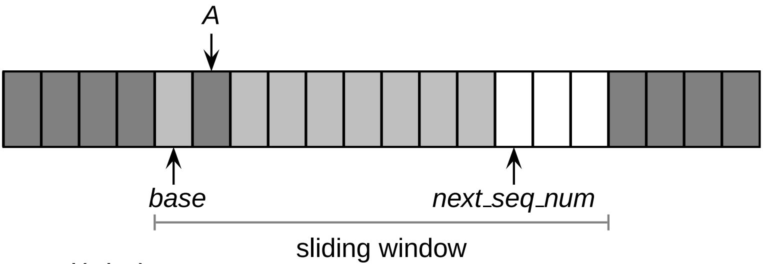
Selective Repeat: Sender



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

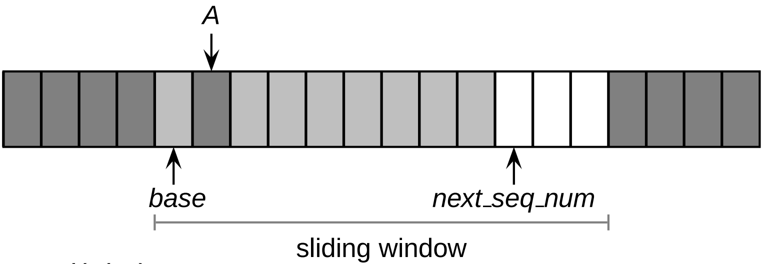
- u_recv([ACK,*A*])

Selective Repeat: Sender



- r_send(pkt_1)
  - u_send([pkt_1,$next\_seq\_num$])
  - start_timer($next\_seq\_num$)
  - $next\_seq\_num \leftarrow next\_seq\_num + 1$
- u_recv([ACK,A])
  - $acks[A] \leftarrow 1$ *// remember that A was ACK'd*

Selective Repeat: Sender

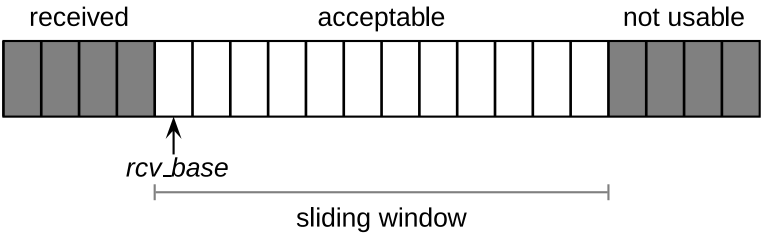


- r_send(pkt_1)
  - u_send([pkt_1,*next_seq_num*])
  - start_timer(*next_seq_num*)
  - $next\_seq\_num \leftarrow next\_seq\_num + 1$

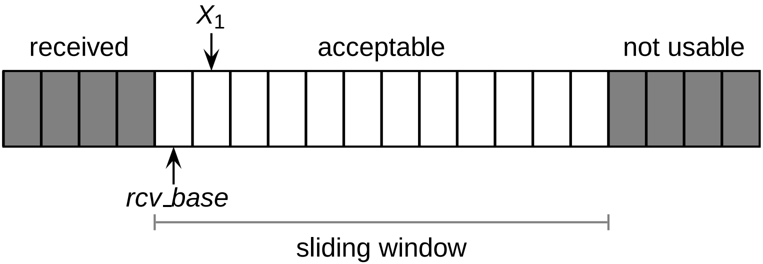
- u_recv([ACK,*A*])
  - $acks[A] \leftarrow 1$ *// remember that A was ACK'd*
  - acknewledgements are no longer "cumulative"

Selective Repeat: Receiver

Selective Repeat: Receiver

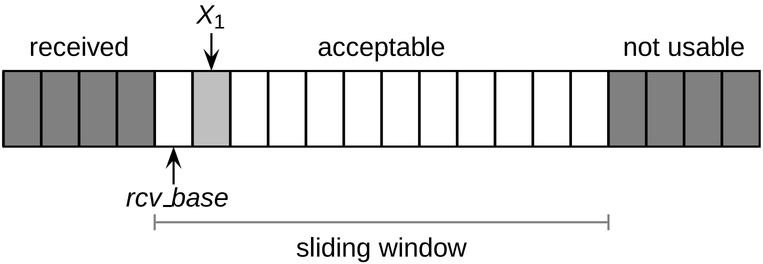


Selective Repeat: Receiver



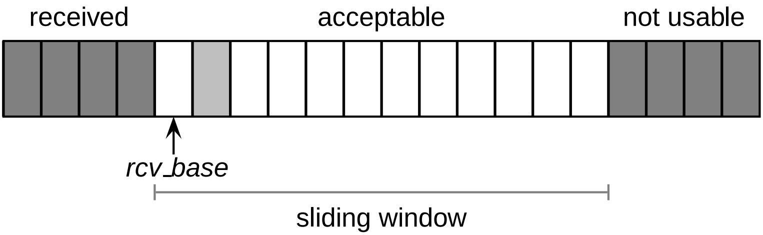
- u_recv([pkt_1,X_1]) **and** $rcv\_base \leq X_1 < rcv\_base + W$

Selective Repeat: Receiver

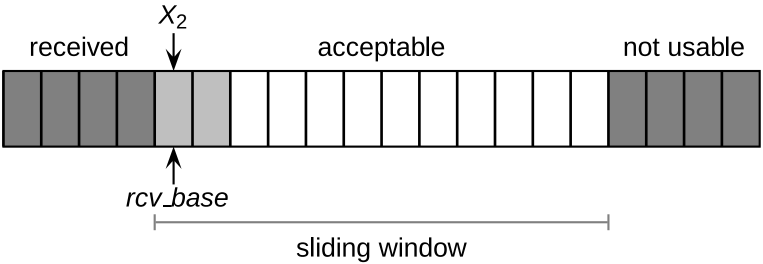


- u_recv([pkt_1,X_1]) **and** $rcv\_base \leq X_1 < rcv\_base + W$
  - $buffer[X_1] \leftarrow pkt_1$
  - u_send([$ACK, X_1]^*$)   *// no longer a "cumulative" ACK*

Selective Repeat: Receiver

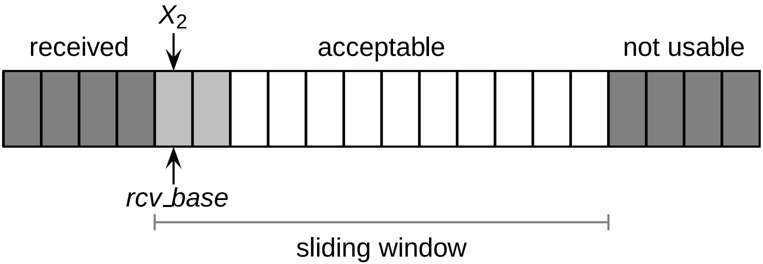


Selective Repeat: Receiver



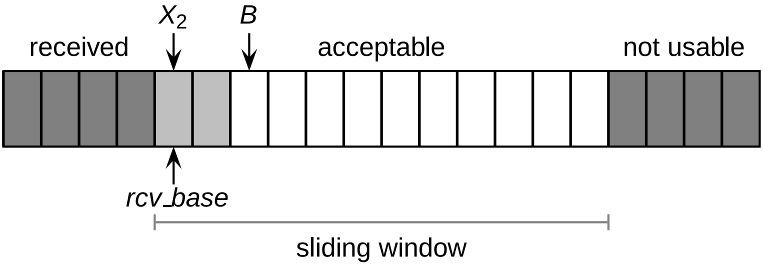
- u_recv([pkt_2,X_2]) **and** $rcv\_base \leq X_2 < rcv\_base + W$
  - $buffer[X_2] \leftarrow pkt_2$
  - u_send([ACK, X_2]*)

Selective Repeat: Receiver



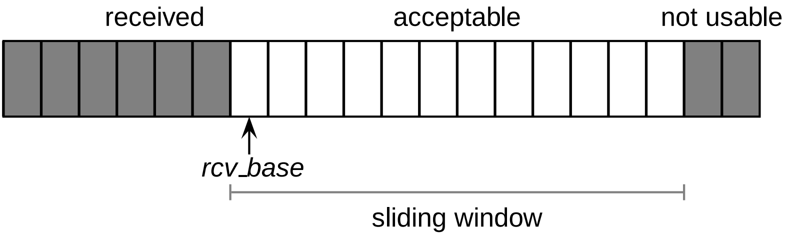
- u_recv([pkt_2,X_2]) **and** $rcv\_base \leq X_2 < rcv\_base + W$
  - $buffer[X_2] \leftarrow pkt_2$
  - u_send($[ACK, X_2]^*$)
  - **if** $(X_2 = rcv\_base)$ {

Selective Repeat: Receiver



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

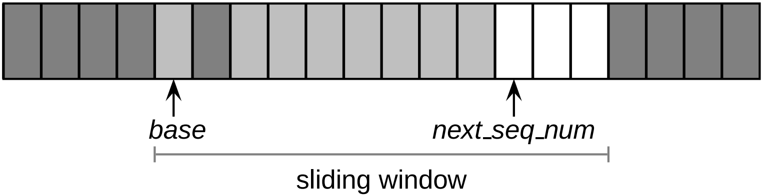
Selective Repeat: Receiver



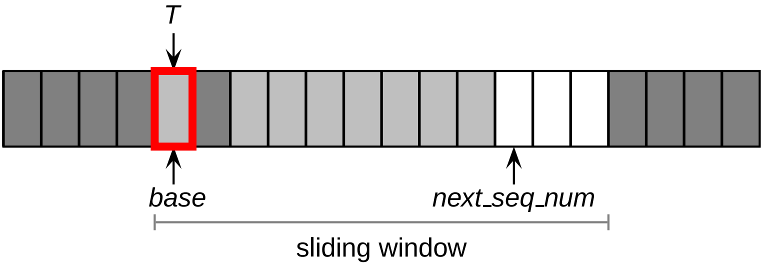
- u_recv([pkt_2,X_2]) **and** $rcv\_base \leq X_2 < rcv\_base + W$
  - $buffer[X_2] \leftarrow pkt_2$
  - u_send([ACK, X_2]*)
  - **if** $(X_2 = rcv\_base)$ {
    $B \leftarrow first\_missing\_seq\_num()$
    foreach i **in** $rcv\_base \ldots B - 1$ {
    r_recv($buffer[i]$) }
    $rcv\_base \leftarrow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

Selective Repeat: Sender

Selective Repeat: Sender

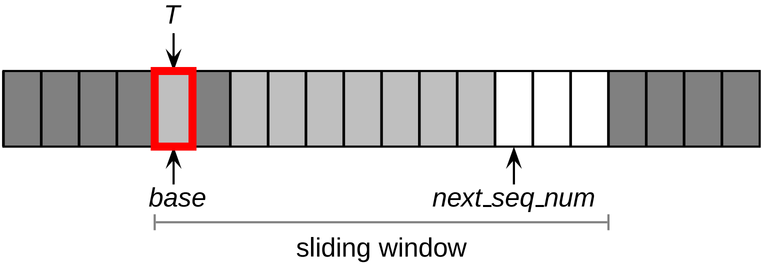


Selective Repeat: Sender



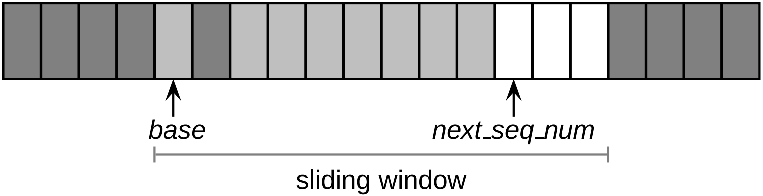
- Timeout for sequence number *T*

Selective Repeat: Sender

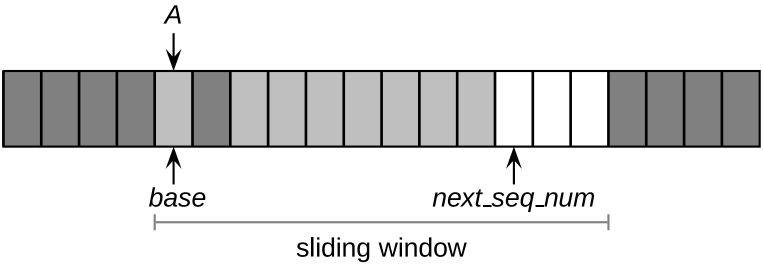


- Timeout for sequence number T
  - u_send($[pkt[T], T]^*$)

Selective Repeat: Sender

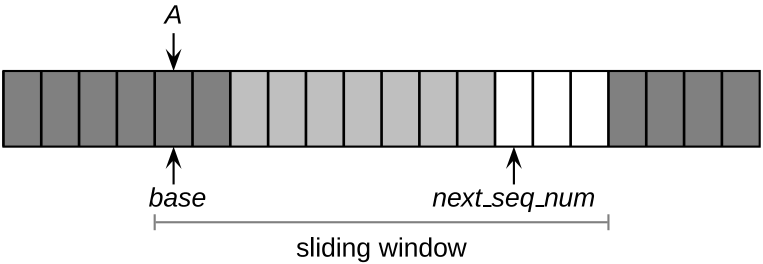


Selective Repeat: Sender



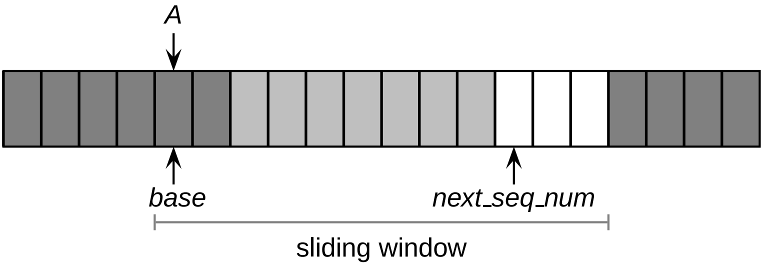
- u_recv([ACK,*A*])

Selective Repeat: Sender



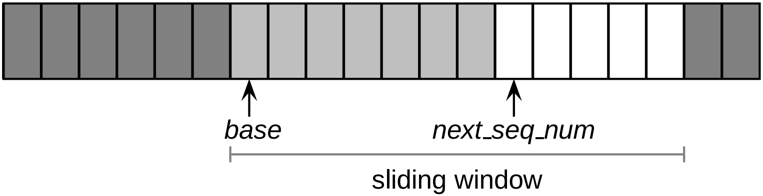
- u_recv([ACK,A])
  - $acks[A] \leftarrow 1$

Selective Repeat: Sender



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

Selective Repeat: Sender



- u_recv([ACK,A])
  - $acks[A] \leftarrow 1$
  - **if** $(A = base)$ $\{$
    $base \leftarrow first\_missing\_ack\_num()$ $\}$

Part III

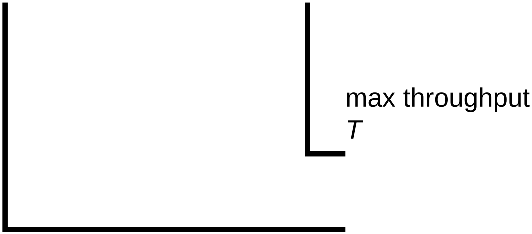
Congestion Control

Understanding Congestion

- A router behaves a lot like a kitchen sink

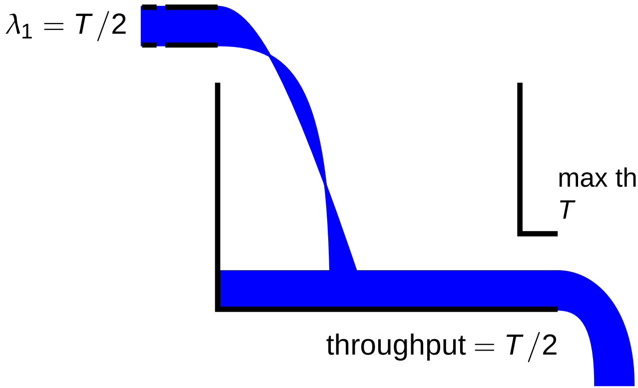
Understanding Congestion

- A router behaves a lot like a kitchen sink



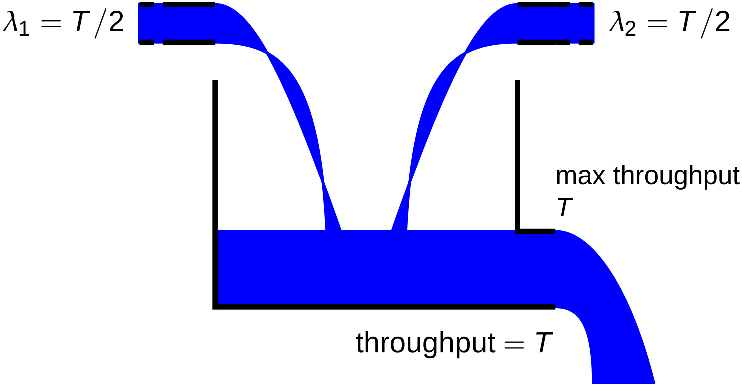
Understanding Congestion

- A router behaves a lot like a kitchen sink



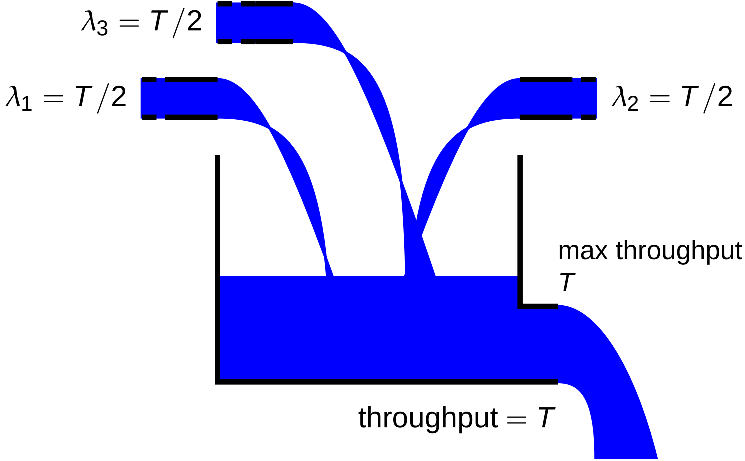
Understanding Congestion

- A router behaves a lot like a kitchen sink



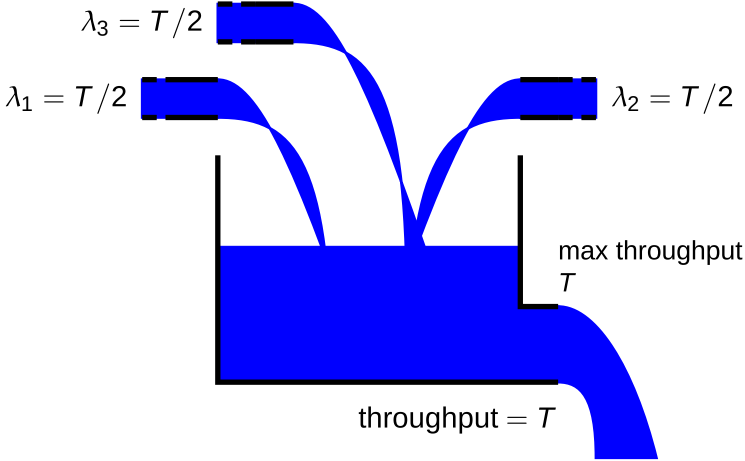
Understanding Congestion

- A router behaves a lot like a kitchen sink



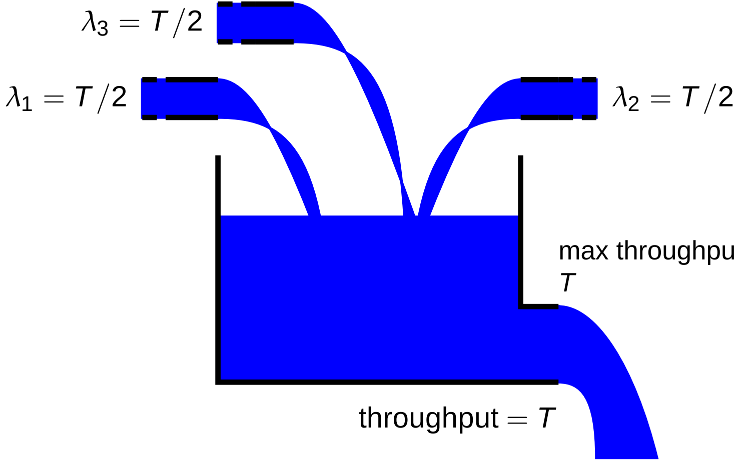
Understanding Congestion

- A router behaves a lot like a kitchen sink



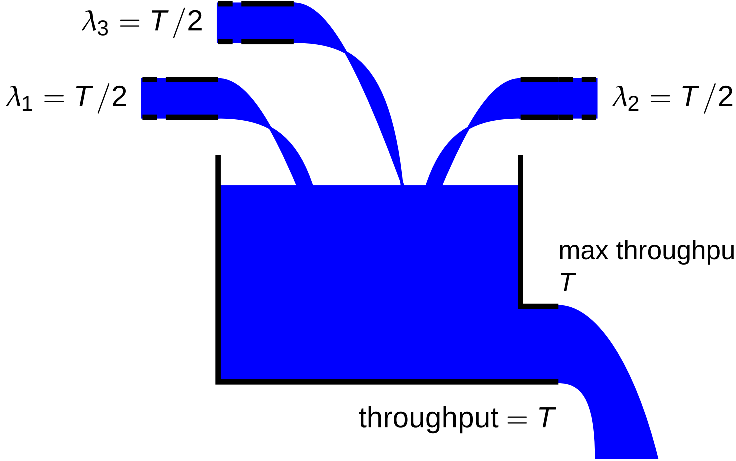
Understanding Congestion

- A router behaves a lot like a kitchen sink



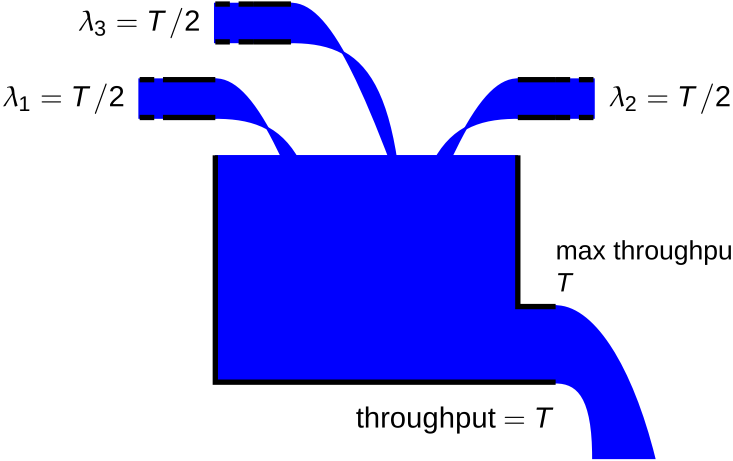
Understanding Congestion

- A router behaves a lot like a kitchen sink



Understanding Congestion

- A router behaves a lot like a kitchen sink



Queuing Delay

Queuing Delay

- Total latency is the sum of link latency, processing time, and the time that a packet spends in the input queue

$$L = \Delta_{TX} + \Delta_{CPU} + \Delta_q \quad \text{where } \Delta_q = |q|/T$$

Queuing Delay

- Total latency is the sum of link latency, processing time, and the time that a packet spends in the input queue

$$L = \Delta_{TX} + \Delta_{CPU} + \Delta_q \quad \text{where } \Delta_q = |q|/T$$

- *Ideal case:* constant input data rate

$$\lambda_{in} < T$$

In this case the $\Delta_q = 0$, because $|q| = 0$

Queuing Delay

- Total latency is the sum of link latency, processing time, and the time that a packet spends in the input queue

$$L = \Delta_{TX} + \Delta_{CPU} + \Delta_q \quad \text{where } \Delta_q = |q|/T$$

- *Ideal case:* constant input data rate

$$\lambda_{in} < T$$

In this case the $\Delta_q = 0$, because $|q| = 0$

- *Extreme case:* constant input data rate

$$\lambda_{in} > T$$

In this case $|q| = (\lambda_{in} - T)t$ and therefore

$$\Delta_q = \frac{\lambda_{in} - T}{T} t$$

Queuing Delay

Queuing Delay

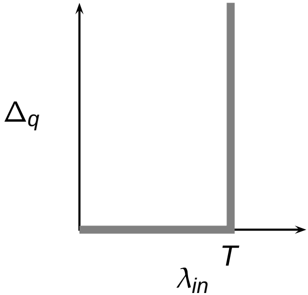
- Steady-state queuing delay

$$\Delta_q = \begin{cases} 0 & \lambda_{in} < T \\ \frac{\lambda_{in} - T}{T} t & \lambda_{in} > T \end{cases}$$

Queuing Delay

- Steady-state queuing delay

$$\Delta_q = \begin{cases} 0 & \lambda_{in} < T \\ \frac{\lambda_{in} - T}{T} t & \lambda_{in} > T \end{cases}$$

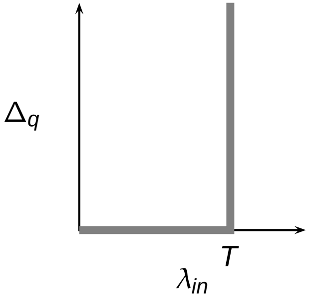


ideal input flow
λ_{in} constant

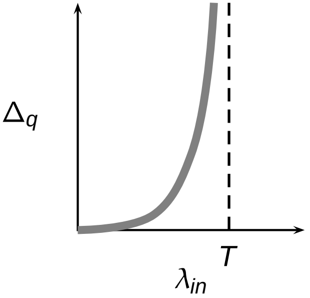
Queuing Delay

- Steady-state queuing delay

$$\Delta_q = \begin{cases} 0 & \lambda_{in} < T \\ \frac{\lambda_{in} - T}{T} t & \lambda_{in} > T \end{cases}$$



ideal input flow
λ_{in} constant



realistic input flow
λ_{in} variable

Queuing Delay

Queuing Delay

- *Conclusion:* as the input rate λ_{in} approaches the maximum throughput T, packets will experience very long delays

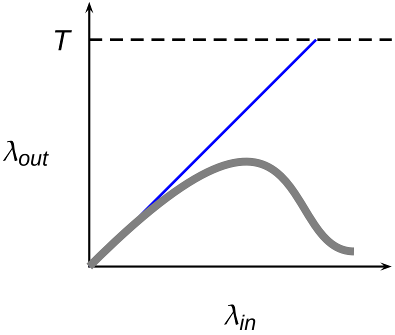
Queuing Delay

- *Conclusion:* as the input rate λ_{in} approaches the maximum throughput T, packets will experience very long delays
- More realistic assumptions and models
  - finite queue length (buffers) in routers
  - effects of retransmission overhead

Queuing Delay

- *Conclusion:* as the input rate λ_{in} approaches the maximum throughput T, packets will experience very long delays

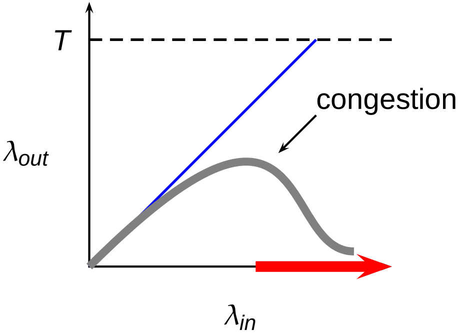
- More realistic assumptions and models
  - finite queue length (buffers) in routers
  - effects of retransmission overhead



Queuing Delay

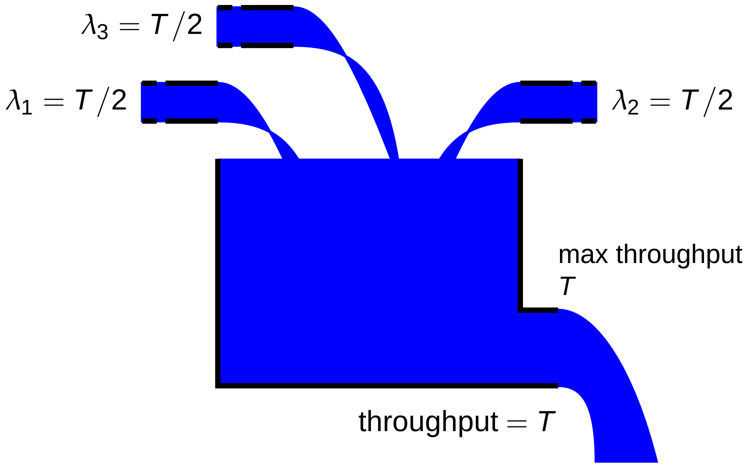
- *Conclusion:* as the input rate λ_{in} approaches the maximum throughput T, packets will experience very long delays

- More realistic assumptions and models
  - finite queue length (buffers) in routers
  - effects of retransmission overhead



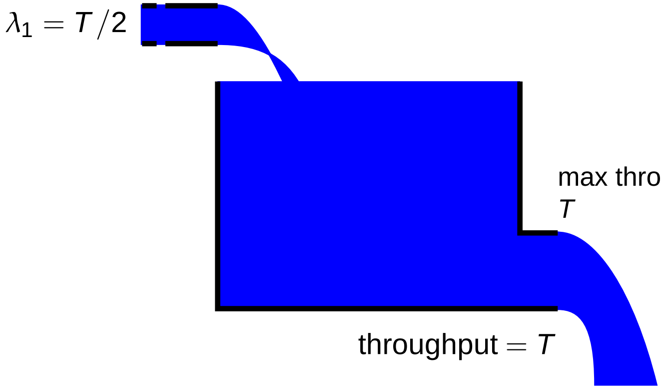
What to Do?

- What to do when the network is congested and queues are full?



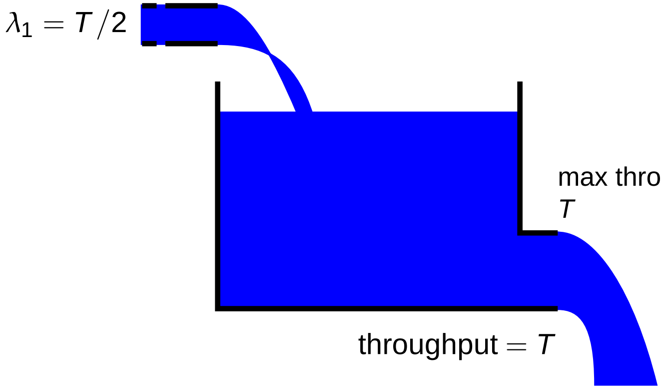
What to Do?

- What to do when the network is congested and queues are full?



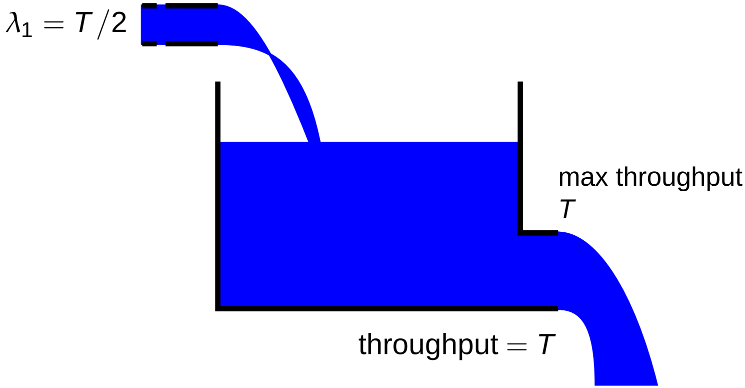
What to Do?

- What to do when the network is congested and queues are full?



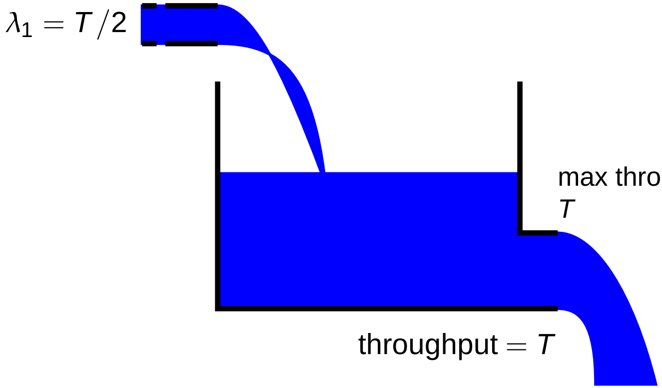
What to Do?

- What to do when the network is congested and queues are full?



What to Do?

- What to do when the network is congested and queues are full?



Part IV

Brief Overview of TCP

Transmission Control Protocol

- The Internet's primary transport protocol
  - defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581

Transmission Control Protocol

- The Internet's primary transport protocol
  - defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581
- Connection-oriented service
  - endpoints "shake hands" to establish a connection
  - not a circuit-switched connection, nor a virtual circuit

Transmission Control Protocol

- The Internet's primary transport protocol
  - defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581
- Connection-oriented service
  - endpoints "shake hands" to establish a connection
  - not a circuit-switched connection, nor a virtual circuit
- Full-duplex service
  - both endpoints can both send and receiver, at the same time

Preliminary Definitions

Preliminary Definitions

- *TCP segment:* envelope for TCP data
  - TCP data are sent within TCP segments
  - TCP segments are usually sent within an IP packet

Preliminary Definitions

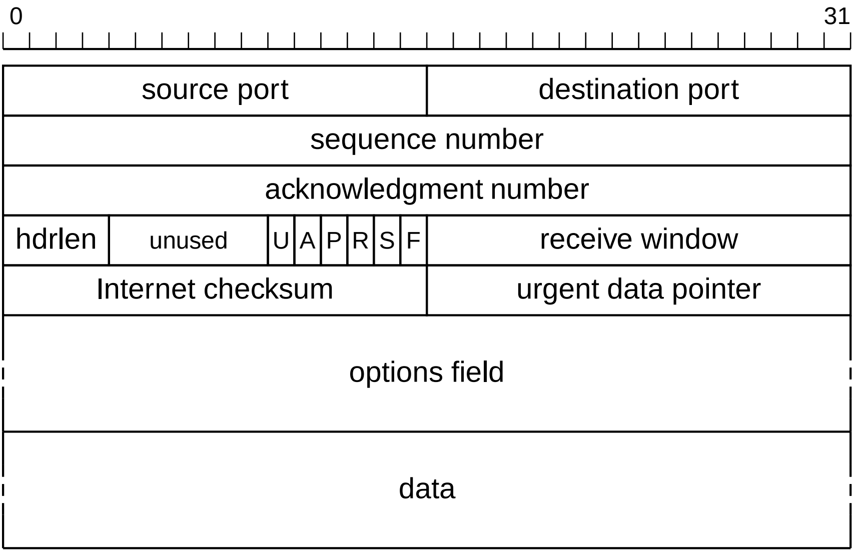
- *TCP segment:* envelope for TCP data
  - TCP data are sent within TCP segments
  - TCP segments are usually sent within an IP packet

- *Maximum segment size (MSS):* maximum amount of application data transmitted in a single segment
  - typically related to the MTU of the connection, to avoid network-level fragmentation (we'll talk about all of this later)

Preliminary Definitions

- *TCP segment:* envelope for TCP data
  - TCP data are sent within TCP segments
  - TCP segments are usually sent within an IP packet
- *Maximum segment size (MSS):* maximum amount of application data transmitted in a single segment
  - typically related to the MTU of the connection, to avoid network-level fragmentation (we'll talk about all of this later)
- *Maximum transmission unit (MTU):* largest link-layer frame available to the sender host
  - *path MTU:* largest link-layer frame that can be sent on all links from the sender host to the receiver host

TCP Segment Format



TCP Header Fields

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers

- *Sequence number:* (32-bit) used to implement reliable data transfer

- *Acknowledgment number:* (32-bit) used to implement reliable data transfer

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer
- *Receive window:* (16-bit) size of the “window” on the receiver end

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer
- *Receive window:* (16-bit) size of the “window” on the receiver end
- *Header length:* (4-bit) size of the TCP header in 32-bit words

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer
- *Receive window:* (16-bit) size of the “window” on the receiver end
- *Header length:* (4-bit) size of the TCP header in 32-bit words
- *Optional and variable-length options field:* may be used to negotiate protocol parameters

TCP Header Fields

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown
- *PSH flag:* (1-bit) “push” flag, used to solicit the receiver to pass the data to the application immediately

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown
- *PSH flag:* (1-bit) "push" flag, used to solicit the receiver to pass the data to the application immediately
- *URG flag:* (1-bit) "urgent" flag, used to inform the receiver that the sender has marked some data as "urgent". The location of this urgent data is marked by the *urgent data pointer* field

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown
- *PSH flag:* (1-bit) “push” flag, used to solicit the receiver to pass the data to the application immediately
- *URG flag:* (1-bit) “urgent” flag, used to inform the receiver that the sender has marked some data as “urgent”. The location of this urgent data is marked by the *urgent data pointer* field
- *Checksum:* (16-bit) used to detect transmission errors

Sequence Numbers

Sequence Numbers

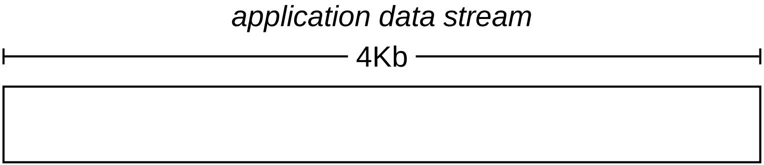
- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before

Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*

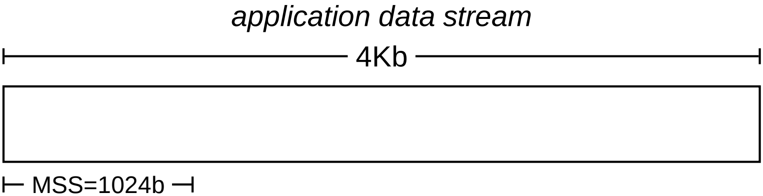
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*



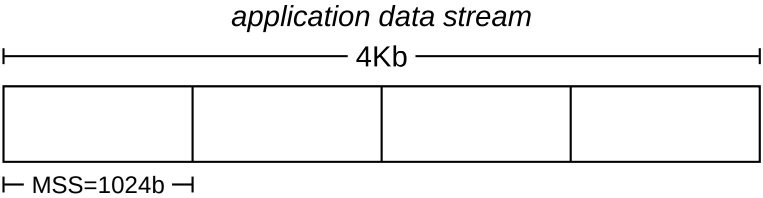
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*



Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*

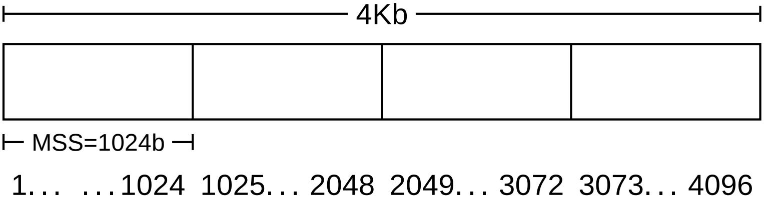


Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before

- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*

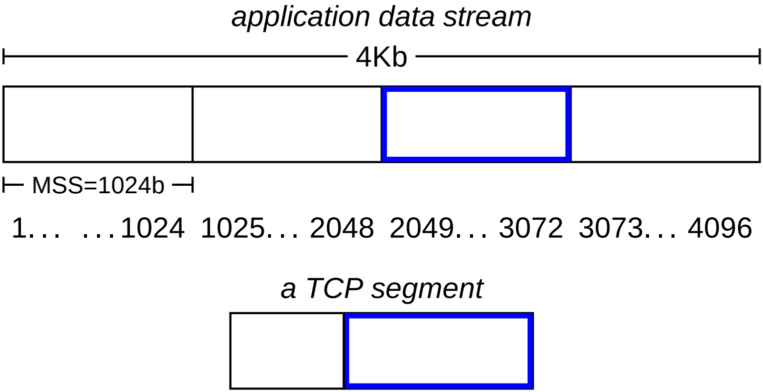
application data stream



Sequence Numbers

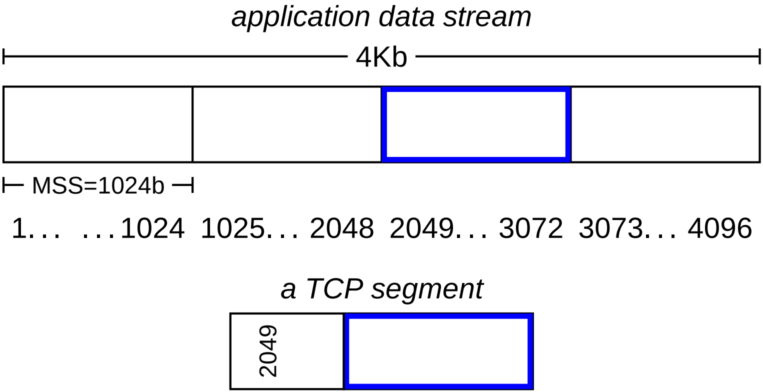
- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before

- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*



Sequence Numbers

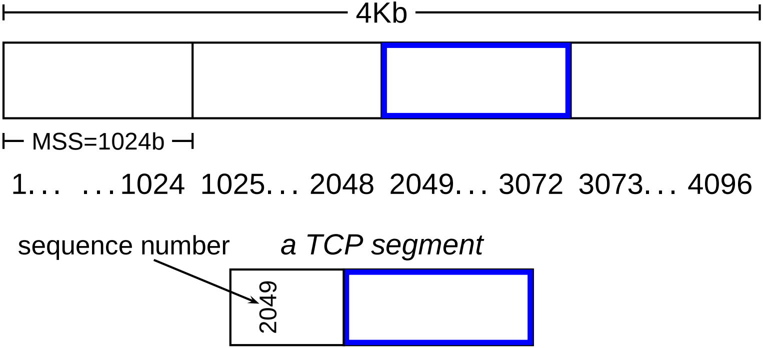
- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*



Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The *sequence number* in a TCP segment indicates *the sequence number of the first byte carried by that segment*

application data stream



Acknowledgment Numbers

Acknowledgment Numbers

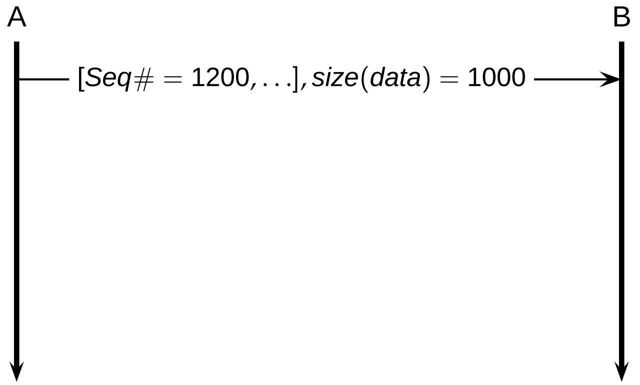
- An *acknowledgment number* represents the *first sequence number not yet seen by the receiver*
  - *TCP acknowledgments are cumulative*

Acknowledgment Numbers

- An *acknowledgment number* represents the *first sequence number not yet seen by the receiver*
  - *TCP acknowledgments are cumulative*

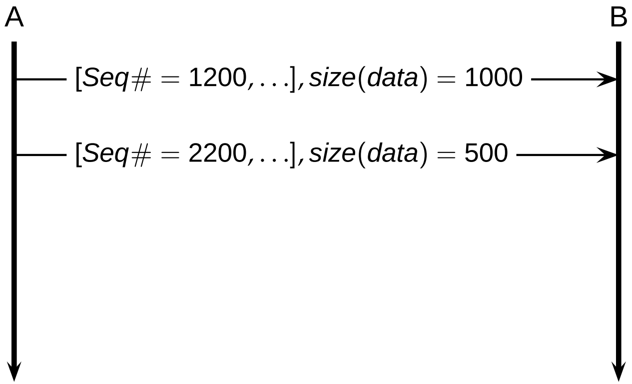
Acknowledgment Numbers

- An *acknowledgment number* represents the *first sequence number not yet seen by the receiver*
  - *TCP acknowledgments are cumulative*



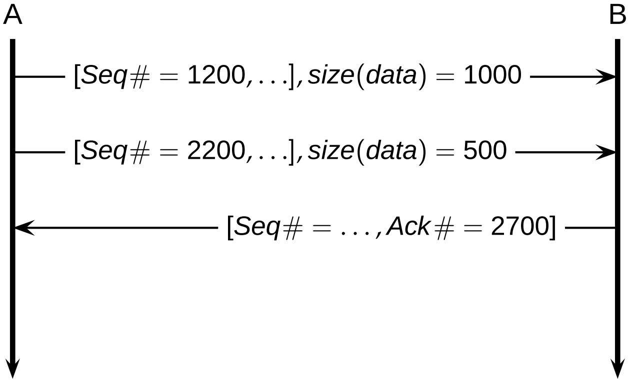
Acknowledgment Numbers

- An *acknowledgment number* represents the *first sequence number not yet seen by the receiver*
  - *TCP acknowledgments are cumulative*



Acknowledgment Numbers

- An *acknowledgment number* represents the *first sequence number not yet seen by the receiver*
  - *TCP acknowledgments are cumulative*



Sequence Numbers and ACK Numbers

Sequence Numbers and ACK Numbers

- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

Sequence Numbers and ACK Numbers

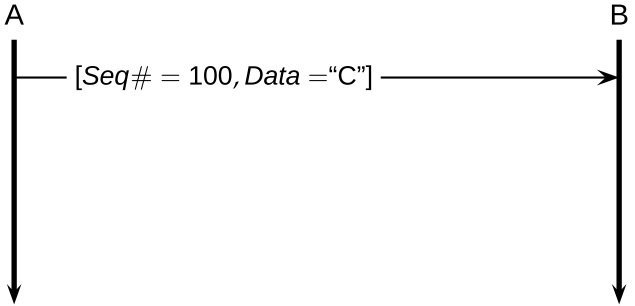
- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

E.g., consider a simple “Echo” application:

Sequence Numbers and ACK Numbers

- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

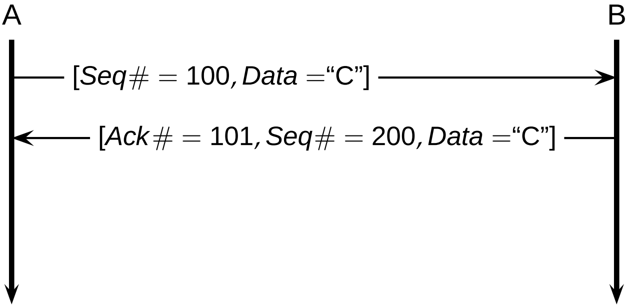
E.g., consider a simple “Echo” application:



Sequence Numbers and ACK Numbers

- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

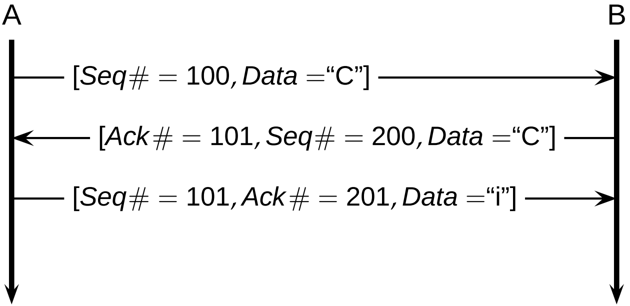
E.g., consider a simple "Echo" application:



Sequence Numbers and ACK Numbers

- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

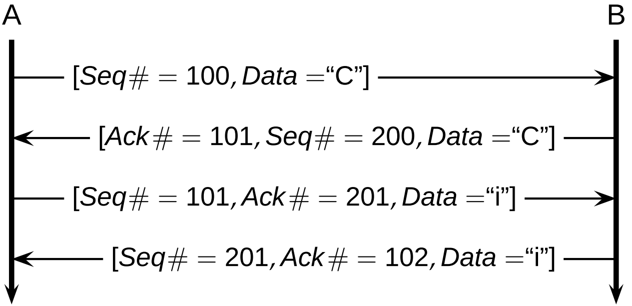
E.g., consider a simple “Echo” application:



Sequence Numbers and ACK Numbers

- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

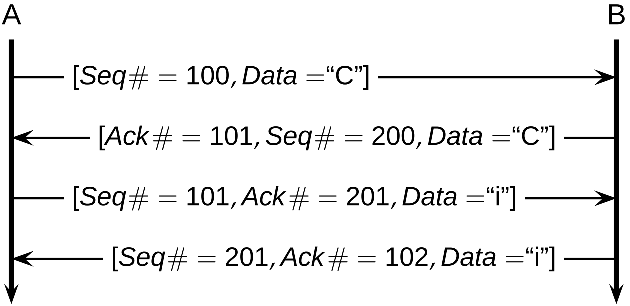
E.g., consider a simple “Echo” application:



Sequence Numbers and ACK Numbers

- Notice that a TCP connection consists of is a *full-duplex* link
  - therefore, there are *two streams*
  - two different sequence numbers

E.g., consider a simple “Echo” application:



- Acknowledgments are “piggybacked” on data segments

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\rightarrow$ lost packet $\rightarrow$ retransmission

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\rightarrow$ lost packet $\rightarrow$ retransmission

- How long to wait for acknowledgments?

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\rightarrow$ lost packet $\rightarrow$ retransmission
- How long to wait for acknowledgments?
- Retransmission timeouts should be larger than the round-trip time $RTT = 2L$
  - as close as possible to the RTT

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\rightarrow$ lost packet $\rightarrow$ retransmission
- How long to wait for acknowledgments?
- Retransmission timeouts should be larger than the round-trip time $RTT = 2L$
  - as close as possible to the RTT
- TCP controls its timeout by continuously *estimating the current RTT*

Round-Trip Time Estimation

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once
- Given a single sample S at any given time
- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once
- Given a single sample S at any given time
- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

  - RFC 2988 recommends $\alpha = 0.125$

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once
- Given a single sample S at any given time
- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

  - RFC 2988 recommends $\alpha = 0.125$
- TCP also measures the *variability of RTT*

$$\overline{DevRTT} = (1-\beta)\overline{DevRTT}' + \beta|\overline{RTT}' - S|$$

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once

- Given a single sample S at any given time

- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

  - RFC 2988 recommends $\alpha = 0.125$

- TCP also measures the *variability of RTT*

$$\overline{DevRTT} = (1-\beta)\overline{DevRTT}' + \beta|\overline{RTT}' - S|$$

  - RFC 2988 recommends $\beta = 0.25$

Timeout Value

Timeout Value

- The timeout interval T must be larger than the RTT
  - so as to avoid unnecessary retransmission

- However, T should not be too far from RTT
  - so as to detect (and retransmit) lost segments as quickly as possible

Timeout Value

- The timeout interval T must be larger than the RTT
  - so as to avoid unnecessary retransmission
- However, T should not be too far from RTT
  - so as to detect (and retransmit) lost segments as quickly as possible
- TCP sets its timeouts using the estimated RTT ($\overline{RTT}$) and the variability estimate $\overline{DevRTT}$:

$$T = \overline{RTT} + 4\overline{DevRTT}$$

Reliable Data Transfer (Sender)

A simplified TCP sender

- r_send(*data*)

  ---

  if (timer not running)

      start_timer()

  u_send([*data*,*next_seq_num*])

  $next\_seq\_num \leftarrow next\_seq\_num + length(data)$

Reliable Data Transfer (Sender)

A simplified TCP sender

- r_send(*data*)

  if (timer not running)
      start_timer()
  u_send([*data*,*next_seq_num*])
  $next\_seq\_num \leftarrow next\_seq\_num + length(data)$

- timeout

  u_send(pending segment with smallest sequence number)
  start_timer()

Reliable Data Transfer (Sender)

A simplified TCP sender

- r_send(*data*)

  if (timer not running)
  start_timer()
  u_send([*data*,*next_seq_num*])
  $next\_seq\_num \leftarrow next\_seq\_num + length(data)$

- timeout

  u_send(pending segment with smallest sequence number)
  start_timer()

- u_recv([ACK,*y*])

  if ($y > base$)
  $base \leftarrow y$
  if ($\exists$ pending segments)
  start_timer()
  else . . .

Acknowledgment Generation (Receiver)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK

- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK

- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK

- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)

- Arrival of out of order segment with higher-than-expected sequence number (gap detected)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK

- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)

- Arrival of out of order segment with higher-than-expected sequence number (gap detected)
  - *Duplicate ACK:* immediately send duplicate ACK

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)
- Arrival of out of order segment with higher-than-expected sequence number (gap detected)
  - *Duplicate ACK:* immediately send duplicate ACK
- Arrival of segment that (partially or completely) fills a gap in the received data

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment. If that doesn't happen, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)
- Arrival of out of order segment with higher-than-expected sequence number (gap detected)
  - *Duplicate ACK:* immediately send duplicate ACK
- Arrival of segment that (partially or completely) fills a gap in the received data
  - *Immediate ACK:* immediately send ACK if the packet start at the lower end of the gap

Reaction to ACKs (Sender)

Reaction to ACKs (Sender)

- u_recv([ACK,*y*])

  ---
  if (*y* $>$ *base*)
      base $\leftarrow$ *y*
      if ($\exists$ pending segments)
          start_timer()

Reaction to ACKs (Sender)

- u_recv([ACK,y])

if ($y > base$)
  $base \leftarrow y$
  if ($\exists$ pending segments)
    start_timer()
else
  $ack\_counter[y] \leftarrow ack\_counter[y] + 1$
  if ($ack\_counter[y] = 3$)
    u_send(segment with sequence number y)

Connection Setup

Connection Setup

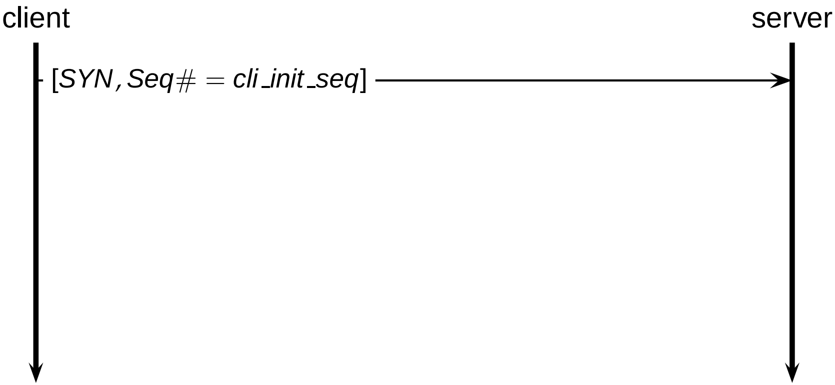
Three-way handshake

Connection Setup

Three-way handshake

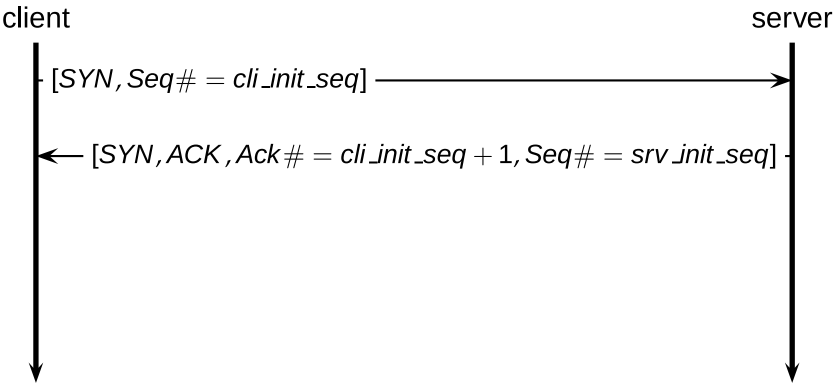
Connection Setup

Three-way handshake



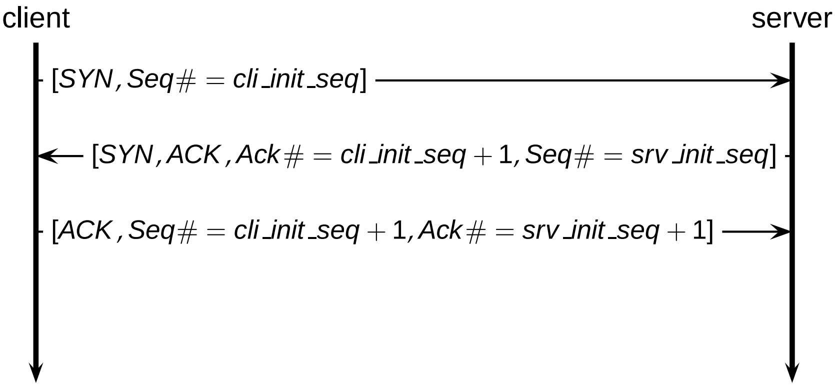
Connection Setup

Three-way handshake



Connection Setup

Three-way handshake



Connection Shutdown

"This is it."

"Okay, Bye now."

"Bye."

Connection Shutdown

“This is it.”

“Okay, Bye now.”

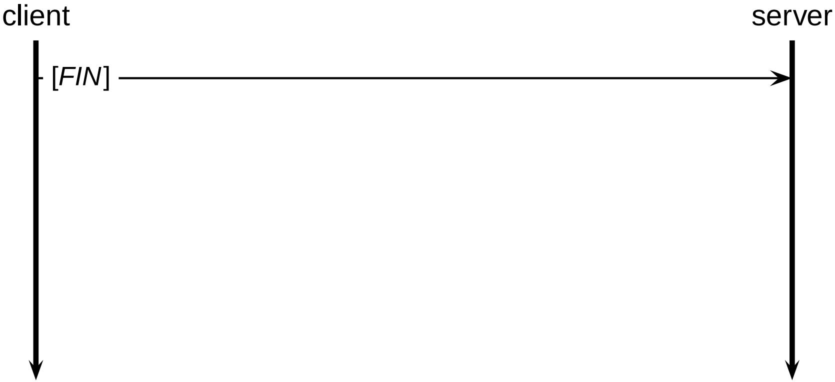
“Bye.”

Connection Shutdown

"This is it."

"Okay, Bye now."

"Bye."

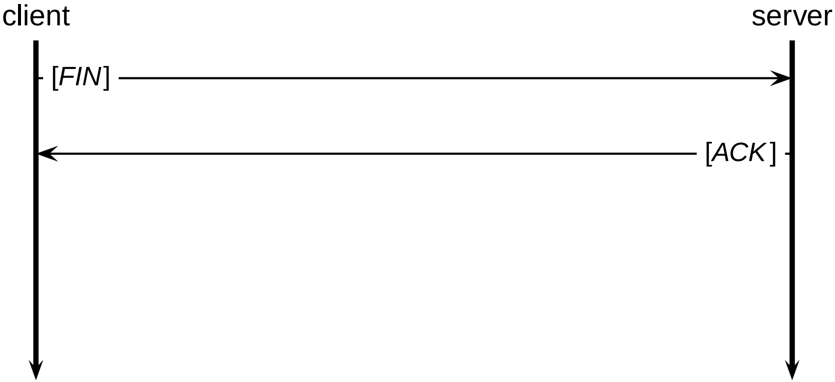


Connection Shutdown

"This is it."

"Okay, Bye now."

"Bye."

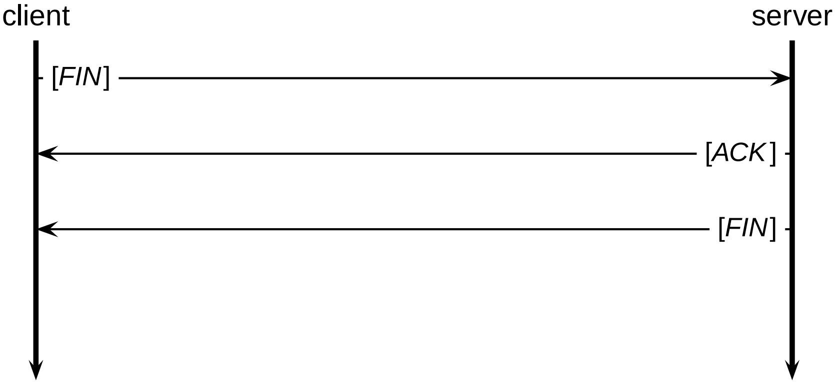


Connection Shutdown

"This is it."

"Okay, Bye now."

"Bye."

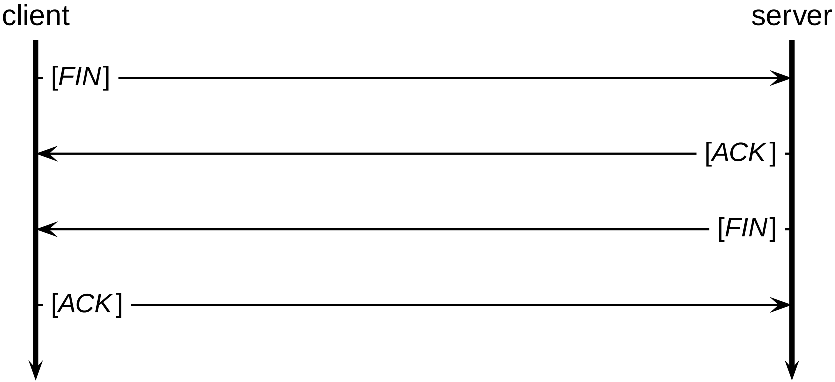


Connection Shutdown

"This is it."

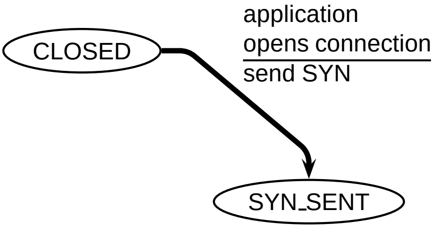
"Okay, Bye now."

"Bye."

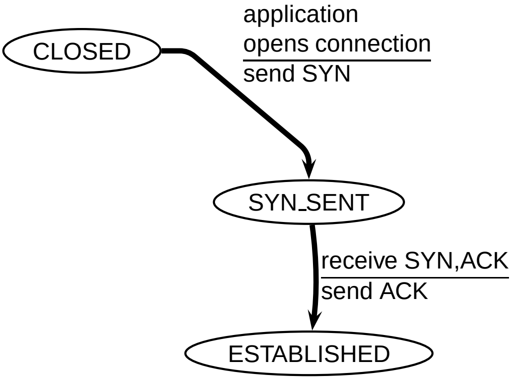


The TCP State Machine (Client)

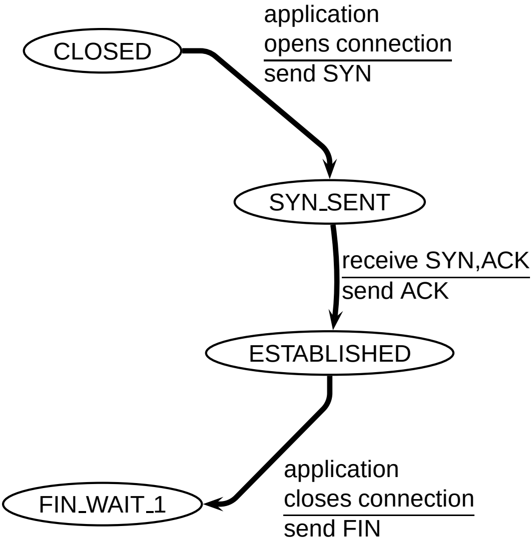
The TCP State Machine (Client)



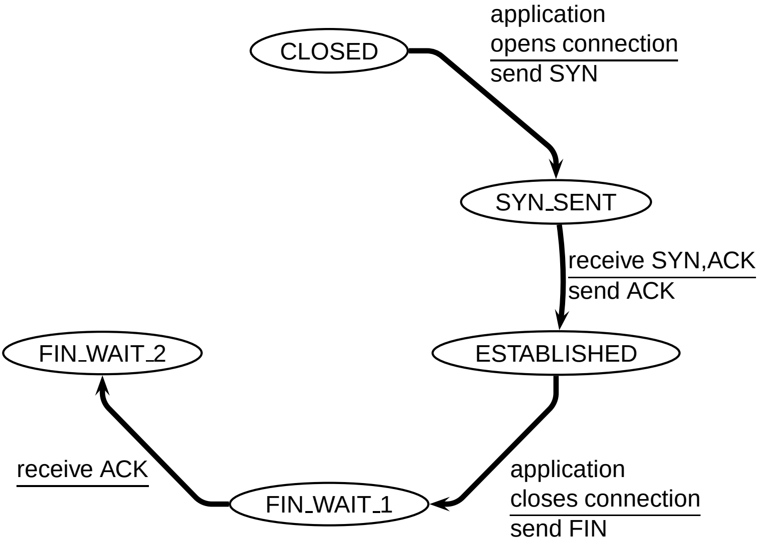
The TCP State Machine (Client)



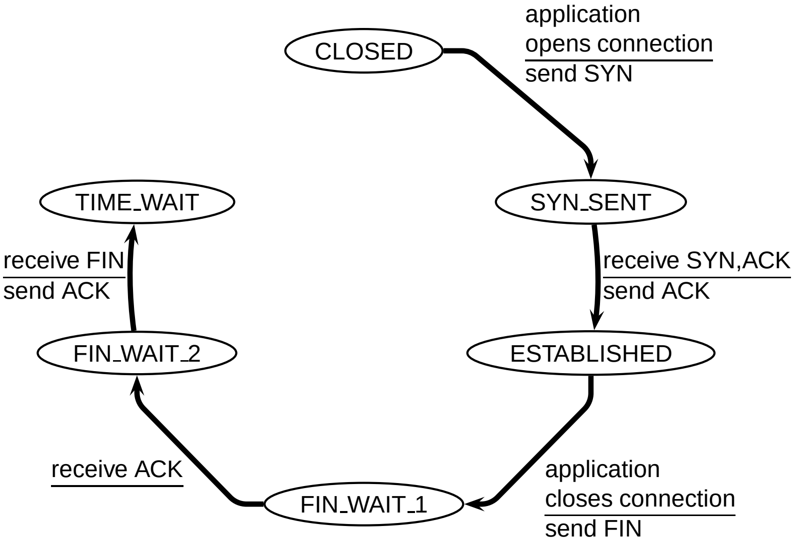
The TCP State Machine (Client)



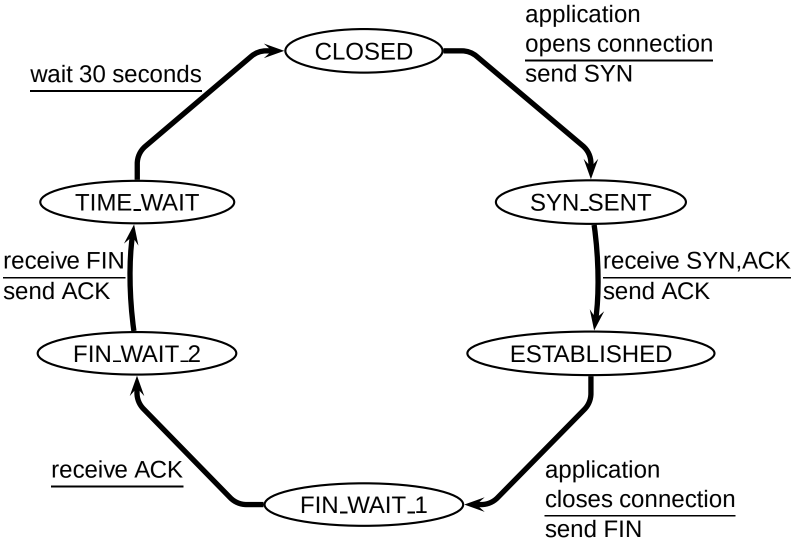
The TCP State Machine (Client)



The TCP State Machine (Client)

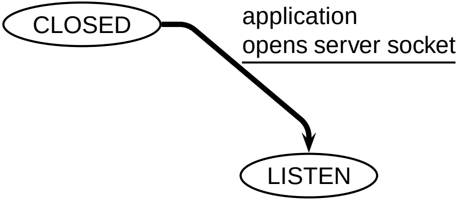


The TCP State Machine (Client)

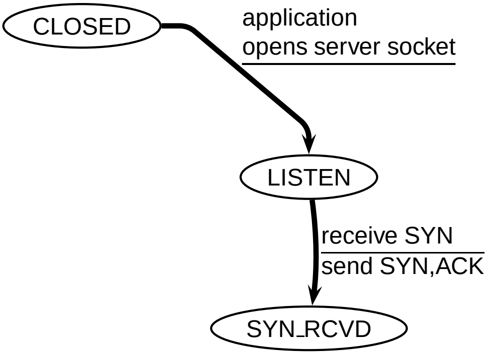


The TCP State Machine (Server)

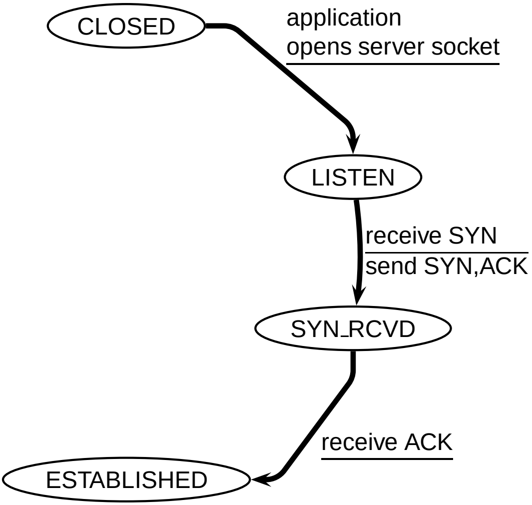
The TCP State Machine (Server)



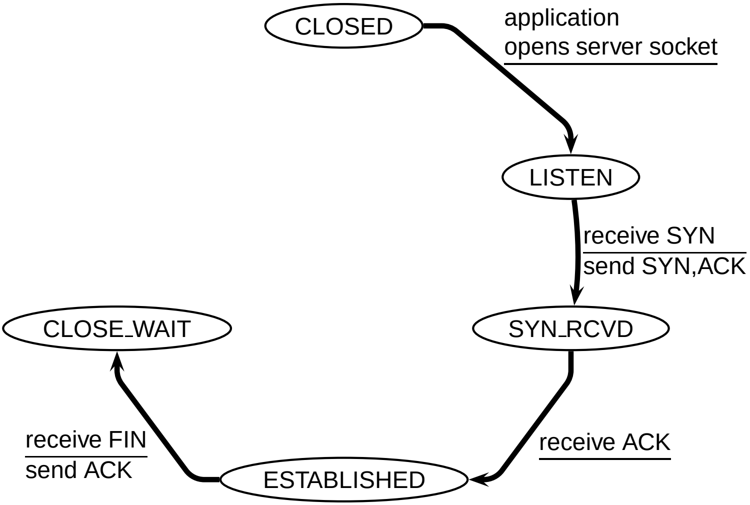
The TCP State Machine (Server)



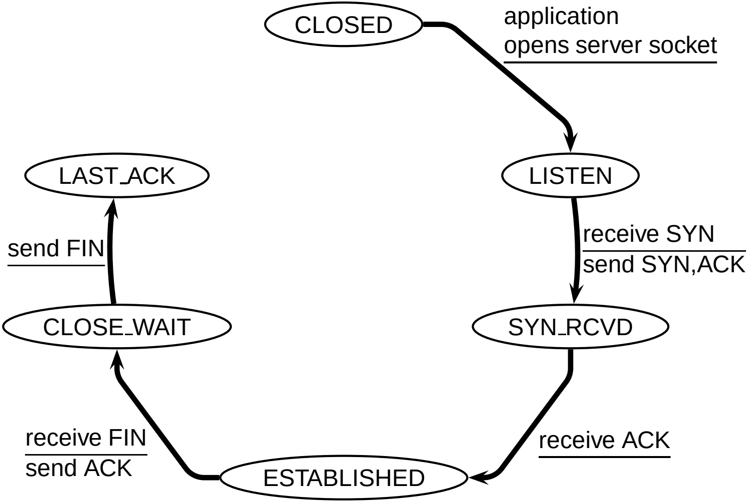
The TCP State Machine (Server)



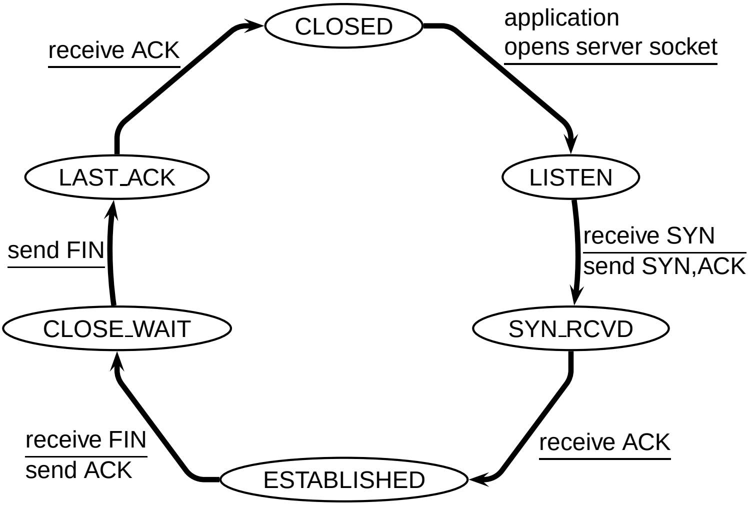
The TCP State Machine (Server)



The TCP State Machine (Server)



The TCP State Machine (Server)



Part V

Congestion Control in TCP

Congestion Control (in TCP)

Congestion Control (in TCP)

- Approach: *the sender limits its output rate according to the status of the network*
  - the sender output rate becomes (part of) the input rate for the network (λ_{in})

Congestion Control (in TCP)

- Approach: *the sender limits its output rate according to the status of the network*
  - the sender output rate becomes (part of) the input rate for the network (λ_{in})

- *Issues*

Congestion Control (in TCP)

- Approach: *the sender limits its output rate according to the status of the network*
  - the sender output rate becomes (part of) the input rate for the network (λ_{in})

- *Issues*
  - how does the sender “measure” the status of the network?
    - i.e., how does the sender detect congestion?

Congestion Control (in TCP)

- Approach: *the sender limits its output rate according to the status of the network*
  - the sender output rate becomes (part of) the input rate for the network (λ_{in})

- *Issues*
  - how does the sender "measure" the status of the network?
    - i.e., how does the sender detect congestion?
  - how does the sender effectively limit its output rate?

Congestion Control (in TCP)

- Approach: *the sender limits its output rate according to the status of the network*
  - the sender output rate becomes (part of) the input rate for the network (λ_{in})

- *Issues*
  - how does the sender "measure" the status of the network?
    - i.e., how does the sender detect congestion?
  - how does the sender effectively limit its output rate?
  - how should the sender "modulate" its output rate?
    - i.e., what algorithm should the sender use to decrease or increase its output rate?

Detecting Congestion

Detecting Congestion

- If all traffic is correctly acknowledged, then the sender assumes (quite correctly) that there is no congestion

Detecting Congestion

- If all traffic is correctly acknowledged, then the sender assumes (quite correctly) that there is no congestion

- Congestion means that queue overflow in one or more routers between the sender and the receiver
  - the visible effect is that some segments are dropped

Detecting Congestion

- If all traffic is correctly acknowledged, then the sender assumes (quite correctly) that there is no congestion
- Congestion means that queue overflow in one or more routers between the sender and the receiver
  - the visible effect is that some segments are dropped
- Therefore the server assumes that the network is congested when it detects a segment loss
  - time out (i.e., no ACK)
  - multiple acknowledgements (i.e., NACK)

Congestion Window

- The sender maintains a *congestion window* W

Congestion Window

- The sender maintains a *congestion window* W
- The congestion window limits the amount of bytes that the sender pushes into the network before blocking waiting for acknowledgments

Congestion Window

- The sender maintains a *congestion window* W

- The congestion window limits the amount of bytes that the sender pushes into the network before blocking waiting for acknowledgments

$$\mathit{LastByteSent} - \mathit{LastByteAcked} \leq W$$

where

$$W = \min(\mathit{CongestionWindow}, \mathit{ReceiverWindow})$$

Congestion Window

- The sender maintains a *congestion window* W

- The congestion window limits the amount of bytes that the sender pushes into the network before blocking waiting for acknowledgments

$$LastByteSent - LastByteAcked \leq W$$

where

$$W = \min(CongestionWindow, ReceiverWindow)$$

- The resulting maximum output rate is roughly

$$\lambda = \frac{W}{2L}$$

Congestion Control

- How does TCP “modulate” its output rate?

Congestion Control

- How does TCP “modulate” its output rate?
- *Additive-increase and multiplicative-decrease*

Congestion Control

- How does TCP “modulate” its output rate?
- *Additive-increase and multiplicative-decrease*
- *Slow start*

Congestion Control

- How does TCP “modulate” its output rate?
- *Additive-increase and multiplicative-decrease*
- *Slow start*
- *Reaction to timeout events*

Additive-Increase/Multiplicative-Decrease

Additive-Increase/Multiplicative-Decrease

- *How W is reduced:* at every *loss* event, TCP halves the congestion window

Additive-Increase/Multiplicative-Decrease

- *How W is reduced:* at every *loss* event, TCP halves the congestion window
  - e.g., suppose the window size W is currently 20Kb, and a loss is detected
  - TCP reduces W to 10Kb

Additive-Increase/Multiplicative-Decrease

- *How W is reduced:* at every *loss* event, TCP halves the congestion window
  - e.g., suppose the window size W is currently 20Kb, and a loss is detected
  - TCP reduces W to 10Kb

- *How W is increased:* at every (good) acknowledgment, TCP increments W by $1MSS/W$, so as to increase W by MSS every round-trip time $2L$. This process is called *congestion avoidance*

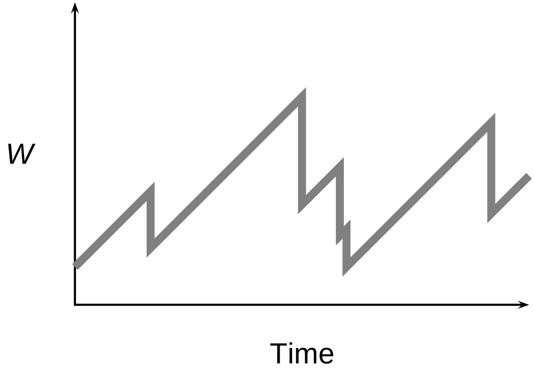
Additive-Increase/Multiplicative-Decrease

- *How W is reduced:* at every *loss* event, TCP halves the congestion window
  - e.g., suppose the window size W is currently 20Kb, and a loss is detected
  - TCP reduces W to 10Kb

- *How W is increased:* at every (good) acknowledgment, TCP increments W by $1MSS/W$, so as to increase W by MSS every round-trip time $2L$. This process is called *congestion avoidance*
  - e.g., suppose $W = 14600$ and $MSS = 1460$, then the sender increases W to 16060 after 10 acknowledgments

Additive-Increase/Multiplicative-Decrease

- Window size W over time



Slow Start

- What is the initial value of W?

Slow Start

- What is the initial value of W?

- The initial value of W is MSS, which is quite low for modern networks

Slow Start

- What is the initial value of W?

- The initial value of W is MSS, which is quite low for modern networks

- In order to get quickly to a good throughput level, TCP increases its sending rate exponentially for its first growth phase

Slow Start

- What is the initial value of W?

- The initial value of W is MSS, which is quite low for modern networks

- In order to get quickly to a good throughput level, TCP increases its sending rate exponentially for its first growth phase

- After experiencing the first loss, TCP cuts W in half and proceeds with its linear push

Slow Start

- What is the initial value of W?

- The initial value of W is MSS, which is quite low for modern networks

- In order to get quickly to a good throughput level, TCP increases its sending rate exponentially for its first growth phase

- After experiencing the first loss, TCP cuts W in half and proceeds with its linear push

- This process is called *slow start*, because of the small initial value of W

Timeouts vs. NACKs

- As we know, three duplicate ACKs are interpreted as a NACK

Timeouts vs. NACKs

- As we know, three duplicate ACKs are interpreted as a NACK
- Both timeouts and NACKs signal a loss, but they say different things about the status of the network

Timeouts vs. NACKs

- As we know, three duplicate ACKs are interpreted as a NACK
- Both timeouts and NACKs signal a loss, but they say different things about the status of the network
- A *timeout indicates congestion*

Timeouts vs. NACKs

- As we know, three duplicate ACKs are interpreted as a NACK
- Both timeouts and NACKs signal a loss, but they say different things about the status of the network
- A *timeout indicates congestion*
- Three (duplicate) ACKs suggest that the network is still able to deliver segments along that path

Timeouts vs. NACKs

- As we know, three duplicate ACKs are interpreted as a NACK
- Both timeouts and NACKs signal a loss, but they say different things about the status of the network
- A *timeout indicates congestion*
- Three (duplicate) ACKs suggest that the network is still able to deliver segments along that path
- So, TCP reacts differently to a timeout and to a triple duplicate ACKs

Timeouts vs. NACKs

Assuming the current window size is $W = \overline{W}$

Timeouts vs. NACKs

Assuming the current window size is $W = \overline{W}$

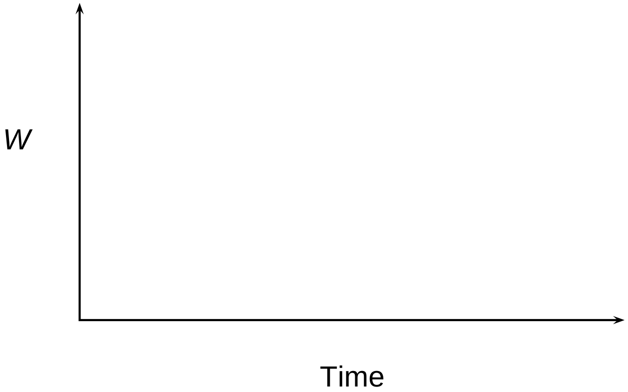
- *Timeout*
  - go back to $W = MSS$
  - run *slow start* until W reaches $\overline{W}/2$
  - then proceed with *congestion avoidance*

Timeouts vs. NACKs

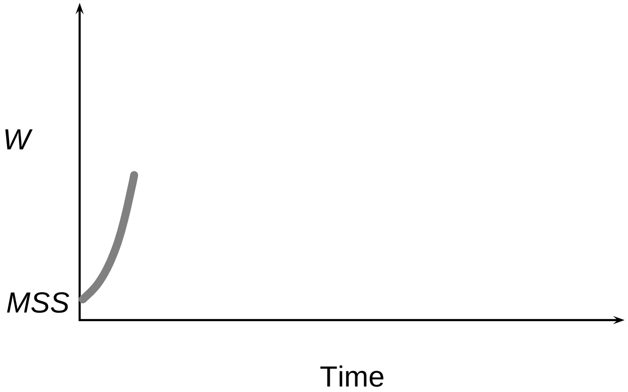
Assuming the current window size is $W = \overline{W}$

- *Timeout*
  - go back to $W = MSS$
  - run *slow start* until W reaches $\overline{W}/2$
  - then proceed with *congestion avoidance*
- *NACK*
  - cut W in half: $W = \overline{W}/2$
  - run *congestion avoidance*, ramping up W linearly
  - This is called *fast recovery*

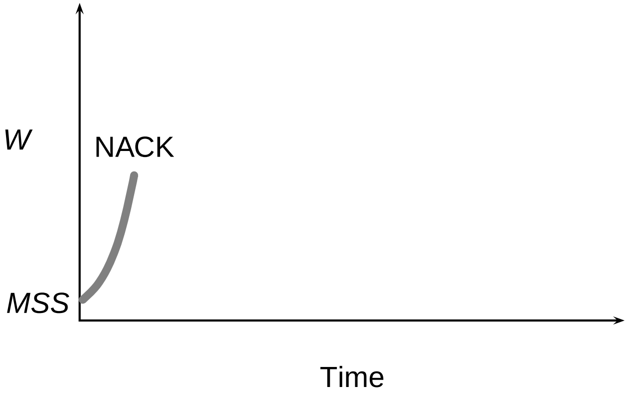
Sender Behavior



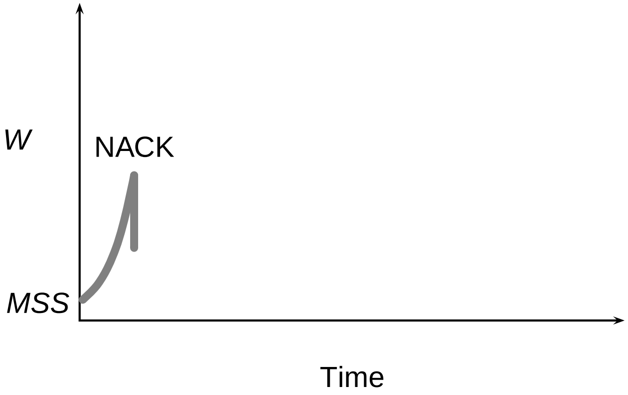
Sender Behavior



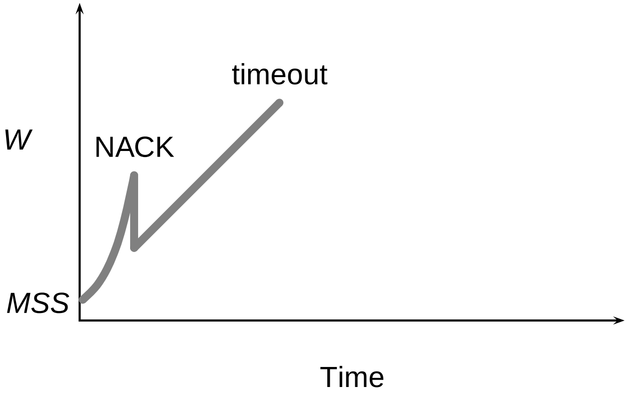
Sender Behavior



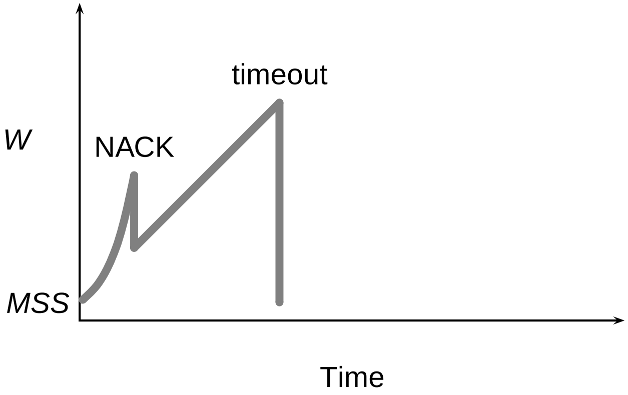
Sender Behavior



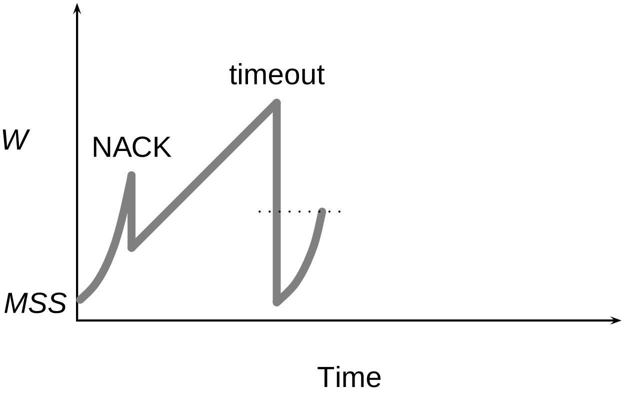
Sender Behavior



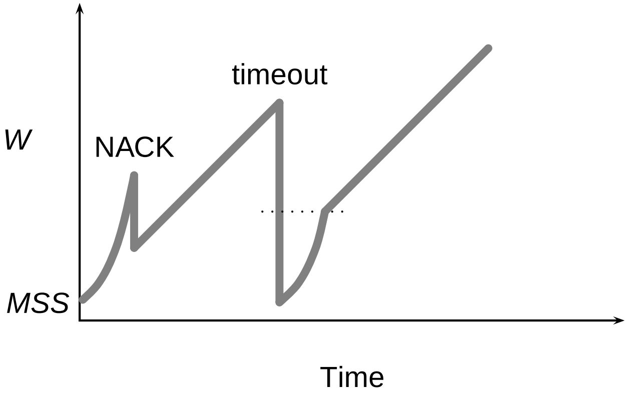
Sender Behavior



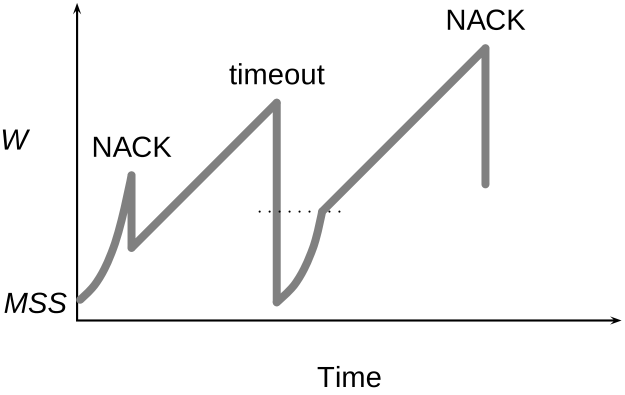
Sender Behavior



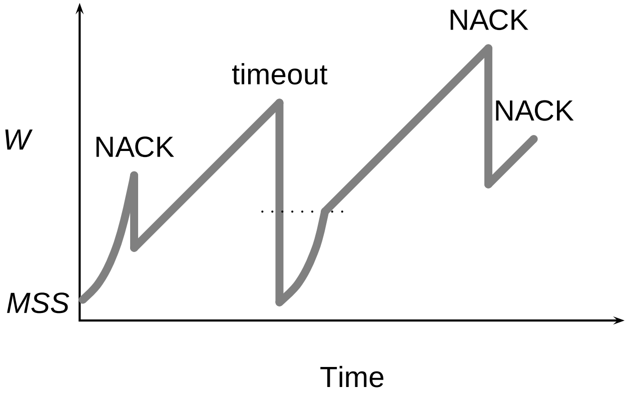
Sender Behavior



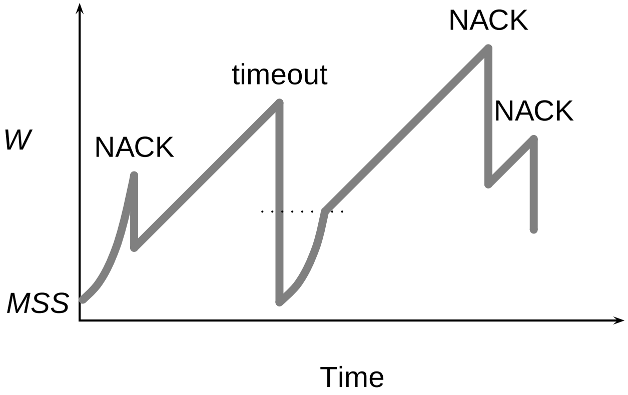
Sender Behavior



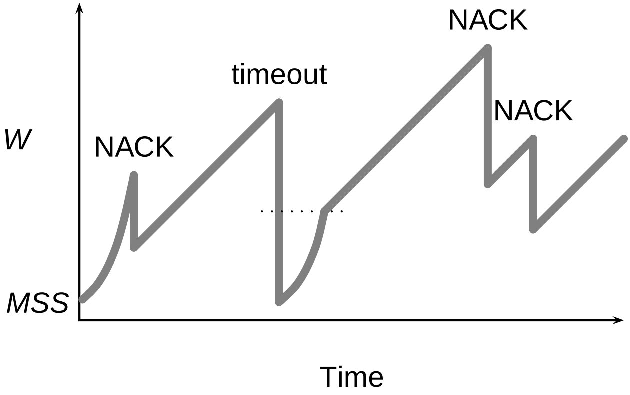
Sender Behavior



Sender Behavior



Sender Behavior



Sender Behavior

SS=*slow start* CA=*congestion avoidance*