

Reliable Data Transfer

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

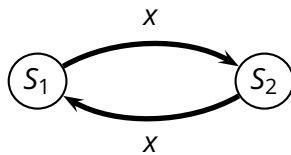
November 3, 2017

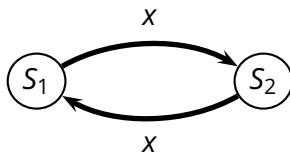
- Finite-state machines
- Using FSMs to specify protocols
- Principles of reliable data transfer
- Reliability over noisy channels
- ACKs/NACKs

- 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)

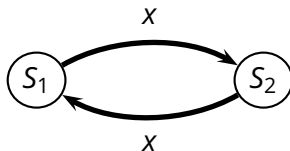
- A *finite-state machine (FSM)* is a mathematical abstraction
 - ▶ a.k.a., finite-state automaton (FSA), deterministic finite-state automaton (DFA), non-deterministic finite-state automaton (NFA)
- FSMs are a very useful formalism to specify and implement network protocols

- 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)
- FSMs are a very useful formalism to specify and implement network protocols
- Ubiquitous in computer science
 - ▶ theory of formal languages
 - ▶ compiler design
 - ▶ theory of computation
 - ▶ text processing
 - ▶ behavior specification
 - ▶ ...

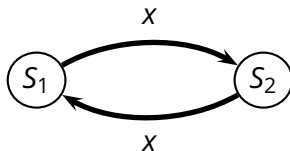




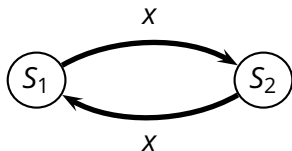
- **States** are represented as *nodes in a graph*



- **States** are represented as *nodes in a graph*
- **Transitions** are represented as *directed edges in the graph*



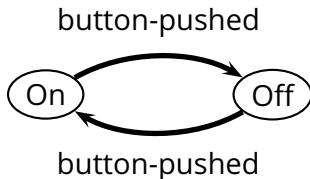
- **States** are represented as *nodes in a graph*
- **Transitions** are represented as *directed edges in the graph*
 - ▶ an edge labeled x going from state S_1 to state S_2 says that when the machine is in state S_1 and event x occurs, the machine switches to state S_2



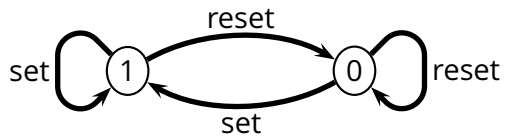
■ **States** are represented as *nodes in a graph*

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

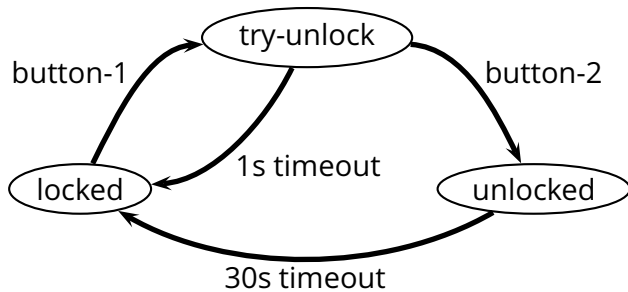
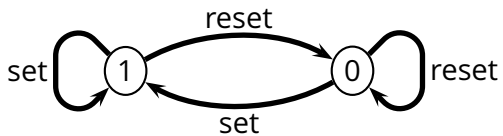
- ▶ an edge labeled x going from state S_1 to state S_2 says that when the machine is in state S_1 and event x occurs, the machine switches to state S_2



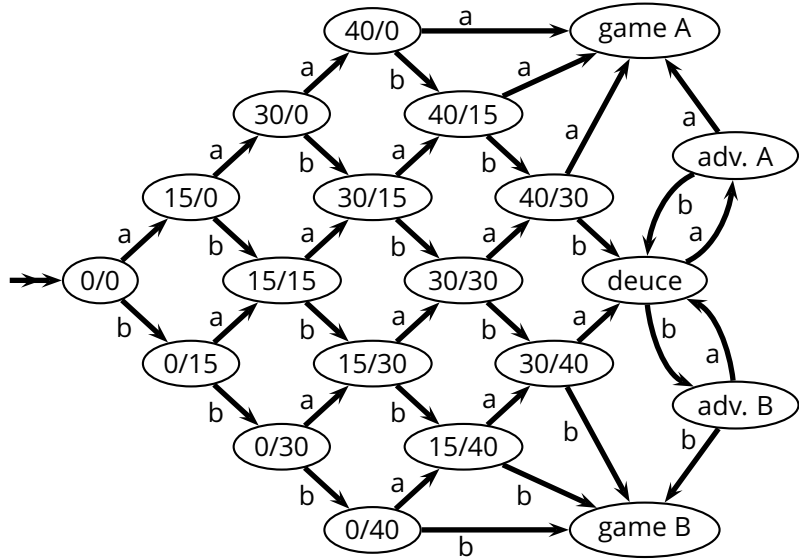
Finite-State Machines



Finite-State Machines



Finite-State Machines

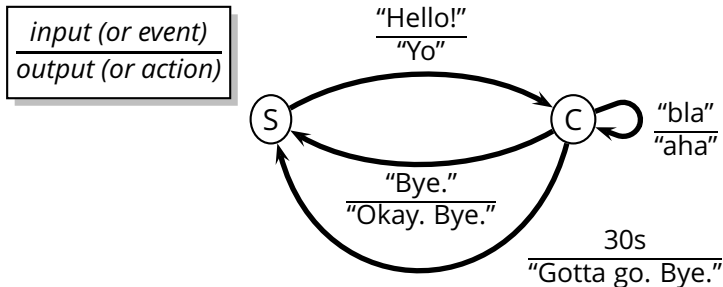


FSMs to Specify Protocols

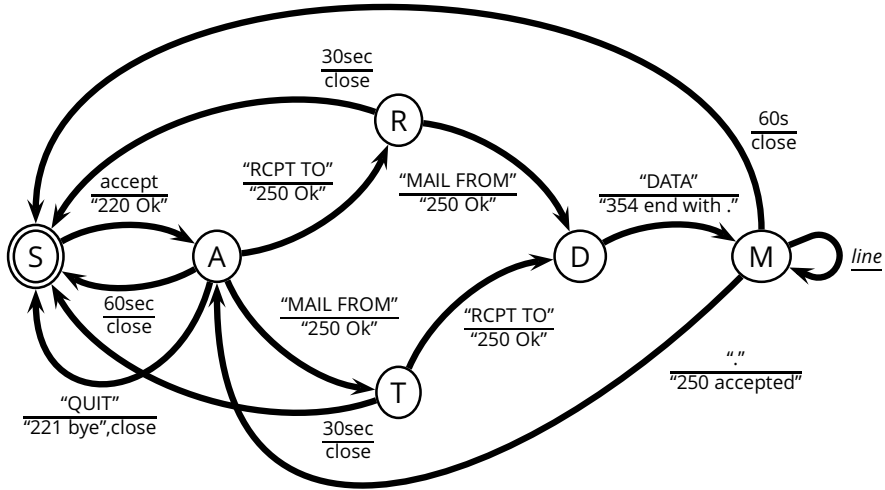
- States represent the *state of a protocol*

- *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***

- 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"



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



Back to Reliable Data Transfer

application

Web
browser

Web
server

Back to Reliable Data Transfer

application

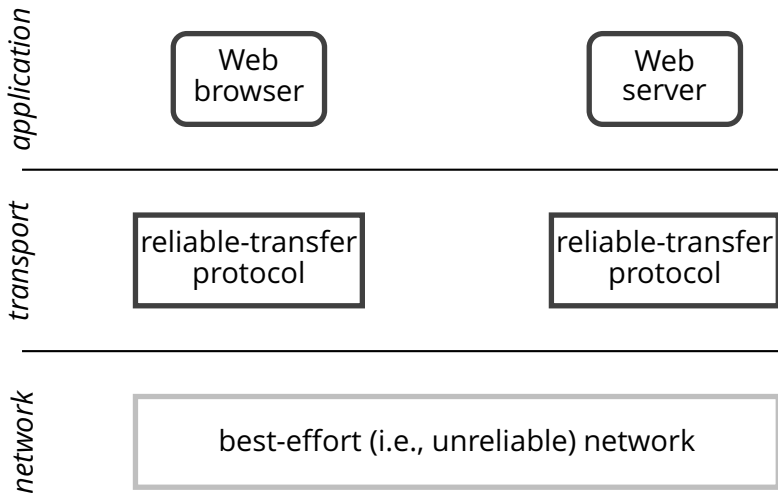
Web
browser

Web
server

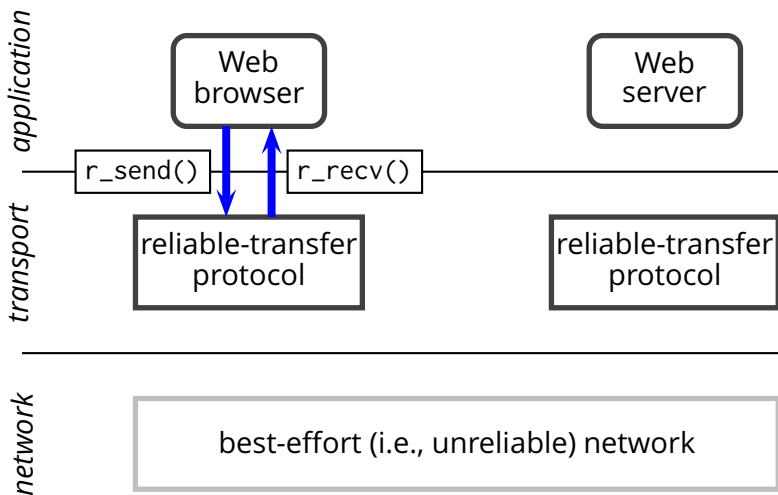
network

best-effort (i.e., unreliable) network

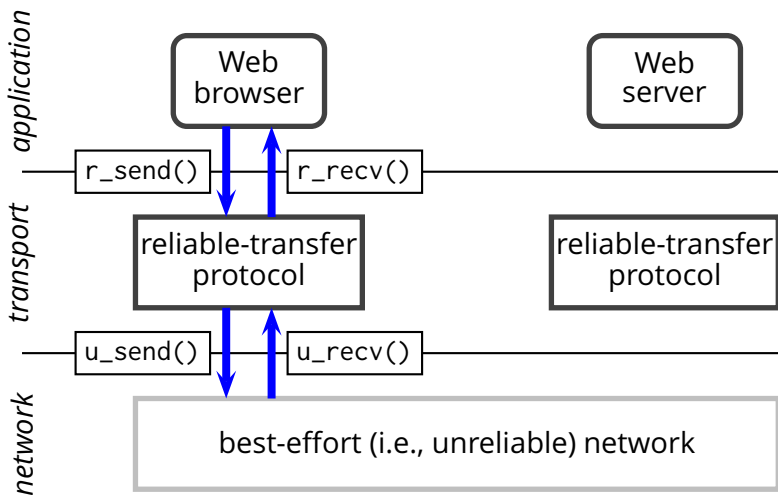
Back to Reliable Data Transfer



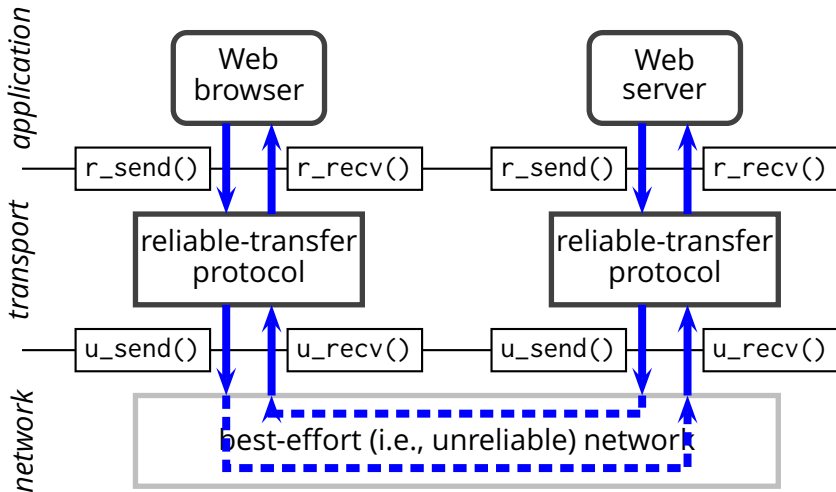
Back to Reliable Data Transfer



Back to Reliable Data Transfer



Back to Reliable Data Transfer



Reliable Data Transfer Model

sender

receiver

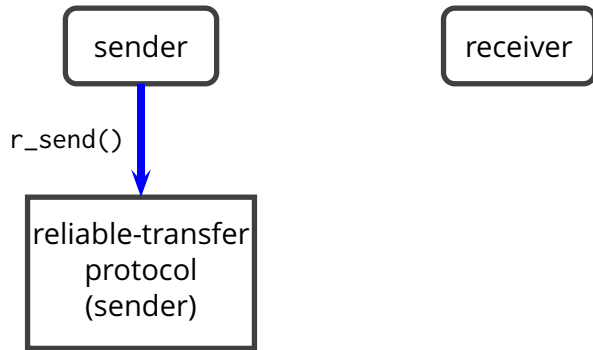
Reliable Data Transfer Model

sender

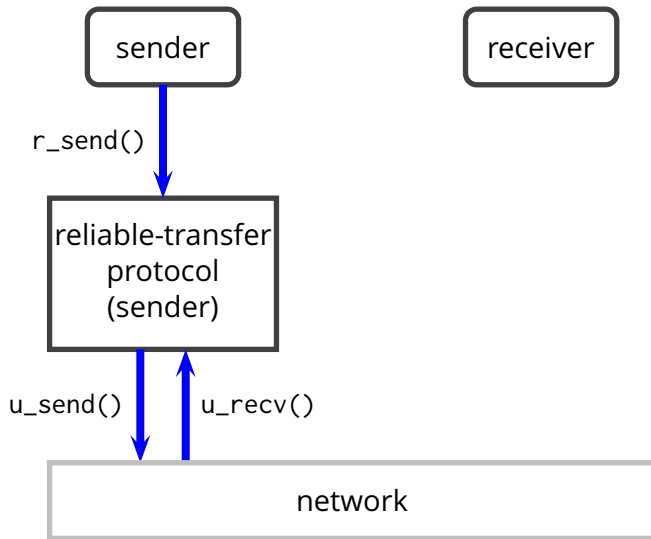
receiver

reliable-transfer
protocol
(sender)

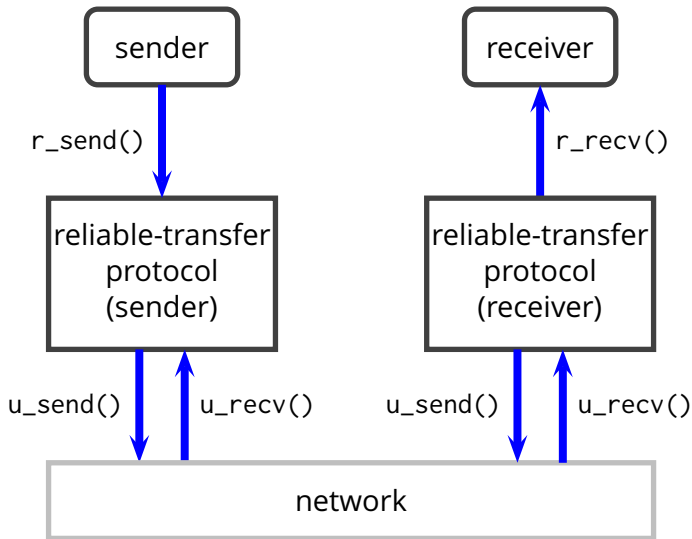
Reliable Data Transfer Model



Reliable Data Transfer Model

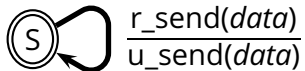


Reliable Data Transfer Model



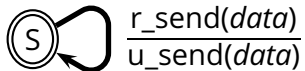
- Reliable transport protocol that uses a reliable network (obviously a contrived example)

sender

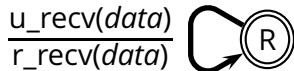


- Reliable transport protocol that uses a reliable network (obviously a contrived example)

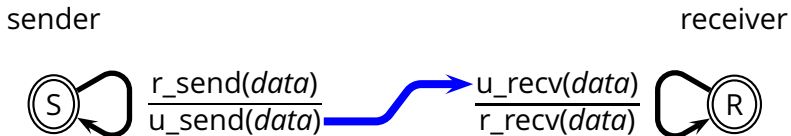
sender



receiver



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



- 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

- 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)

- 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)

- 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

- 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

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

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

- Key idea: *sending redundant information*
 - ▶ e.g., the sender could repeat the message twice
 - ▶ error when the receiver hears two different messages
 - ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

- Key idea: *sending redundant information*
 - ▶ e.g., the sender could repeat the message twice
 - ▶ error when the receiver hears two different messages
 - ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

- *Error-detection codes*

- Key idea: *sending redundant information*
 - ▶ e.g., the sender could repeat the message twice
 - ▶ error when the receiver hears two different messages
 - ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

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

- Key idea: *sending redundant information*
 - ▶ e.g., the sender could repeat the message twice
 - ▶ error when the receiver hears two different messages
 - ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

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

■ Key idea: *sending redundant information*

- ▶ e.g., the sender could repeat the message twice
- ▶ error when the receiver hears two different messages
- ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

■ *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)

■ Key idea: *sending redundant information*

- ▶ e.g., the sender could repeat the message twice
- ▶ error when the receiver hears two different messages
- ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

■ *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

■ Key idea: *sending redundant information*

- ▶ e.g., the sender could repeat the message twice
- ▶ error when the receiver hears two different messages
- ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

■ *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 10010110111010000

■ Key idea: *sending redundant information*

- ▶ e.g., the sender could repeat the message twice
- ▶ error when the receiver hears two different messages
- ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

■ *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 10010110111010000

Receiver:

receives 1001011010101000

■ Key idea: *sending redundant information*

- ▶ e.g., the sender could repeat the message twice
- ▶ error when the receiver hears two different messages
- ▶ not very efficient (uses twice the number of bits) but there are better error-detection codes

■ *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 10010110111010000

Receiver:

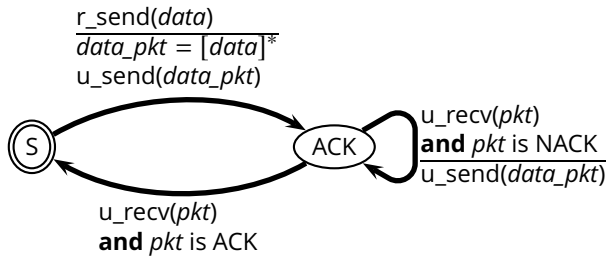
receives 1001011010101000 \Rightarrow error!

- Sender

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

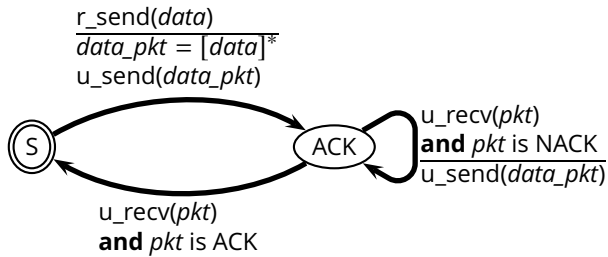
■ Sender

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

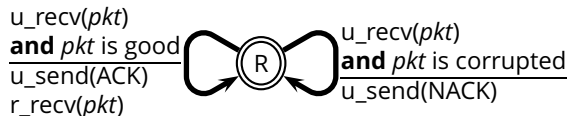


■ Sender

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



■ Receiver



- 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

- 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?

- 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 lose 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 lose 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 lose 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: Sender

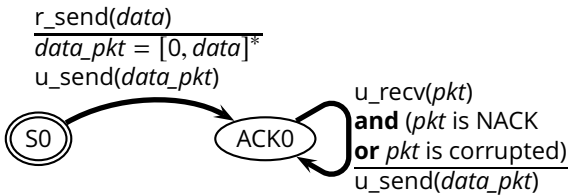


Using Sequence Numbers: Sender

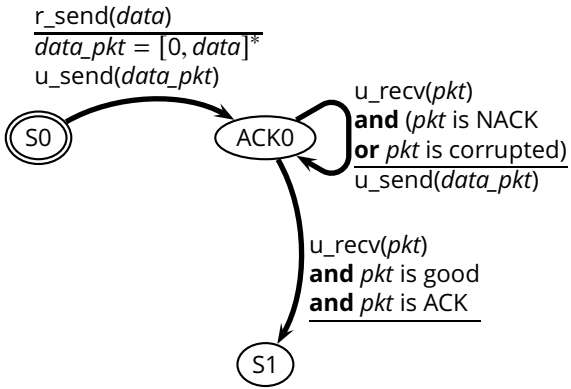
$r_send(data)$
 $\hline data_pkt = [0, data]^*$
 $u_send(data_pkt)$



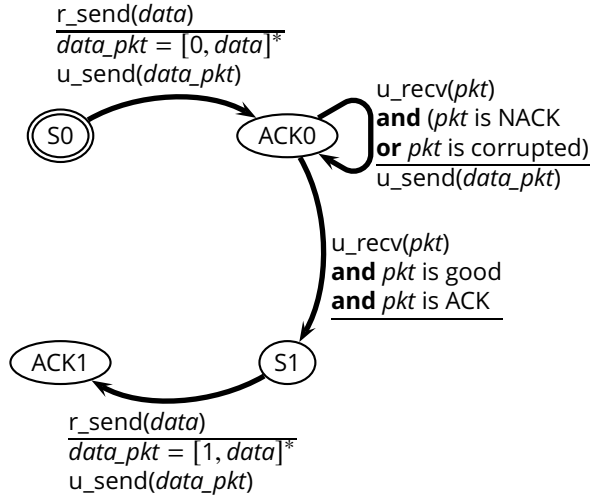
Using Sequence Numbers: Sender



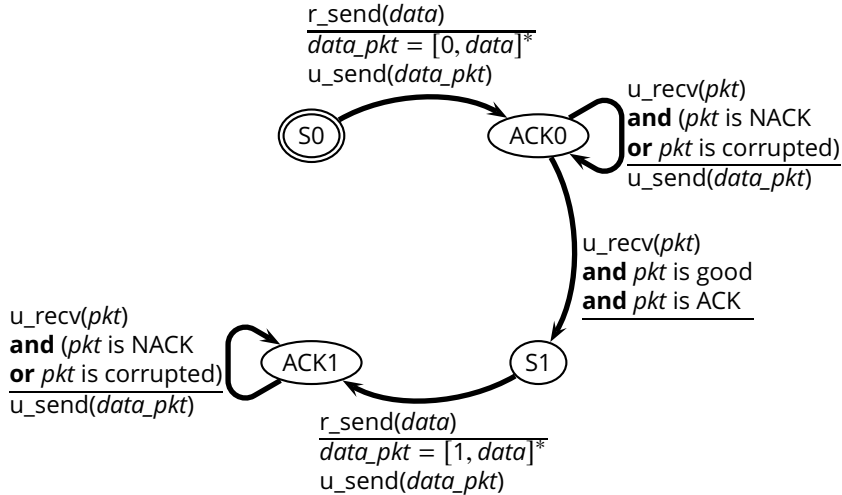
Using Sequence Numbers: Sender



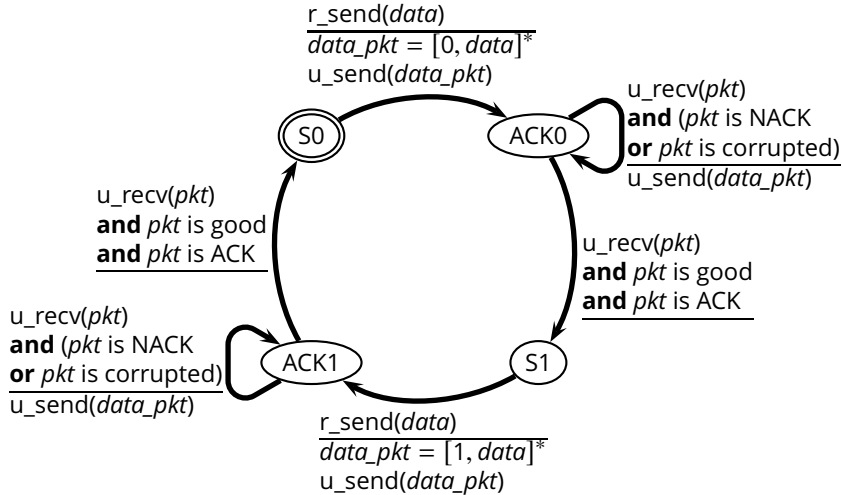
Using Sequence Numbers: Sender



Using Sequence Numbers: Sender



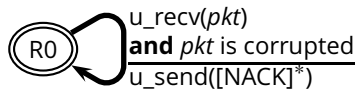
Using Sequence Numbers: Sender



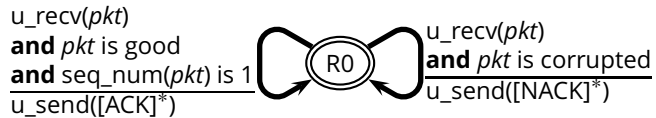
Using Sequence Numbers: Receiver



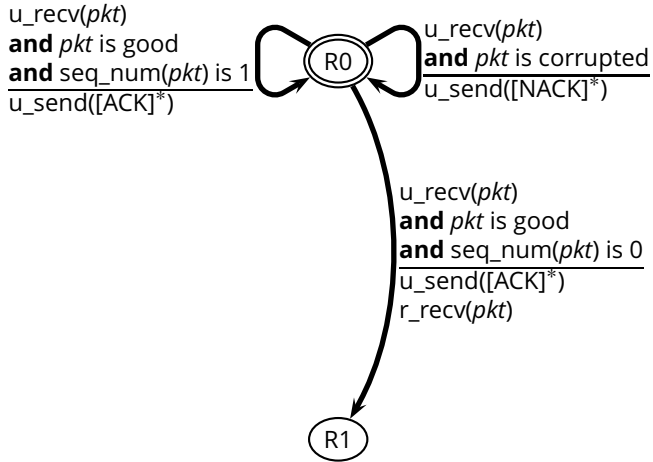
Using Sequence Numbers: Receiver



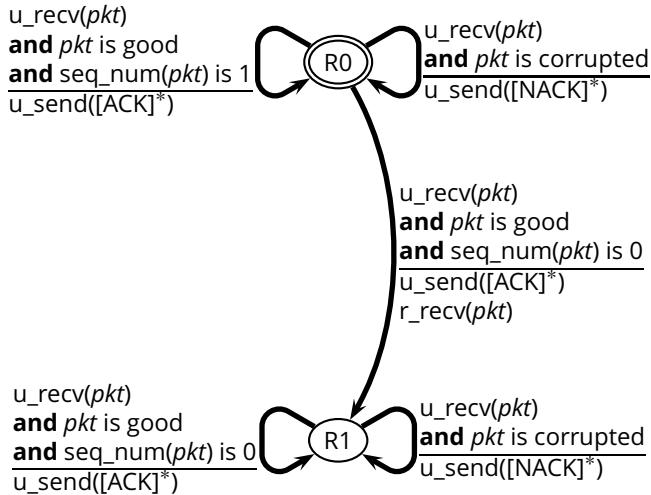
Using Sequence Numbers: Receiver



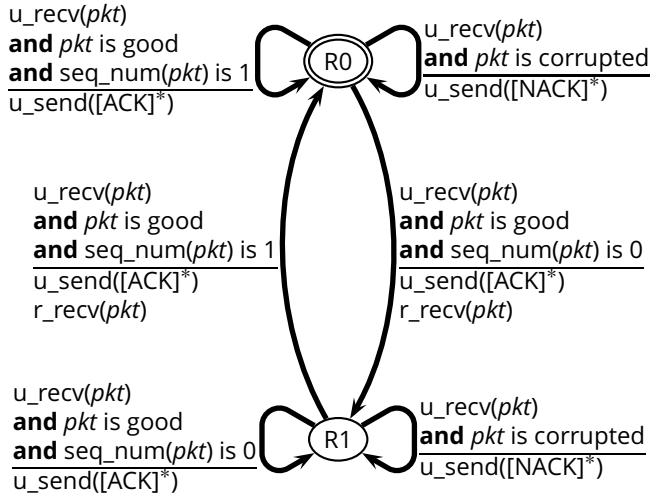
Using Sequence Numbers: Receiver



Using Sequence Numbers: Receiver



Using Sequence Numbers: Receiver



- Do we really need both ACKs and NACKs?

- 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

- 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 ..."

- 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"

- 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

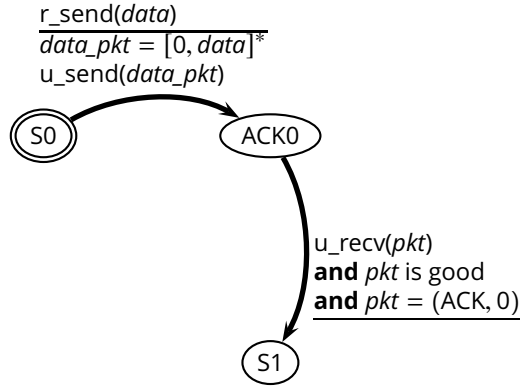
S0

ACK-Only Protocol: Sender

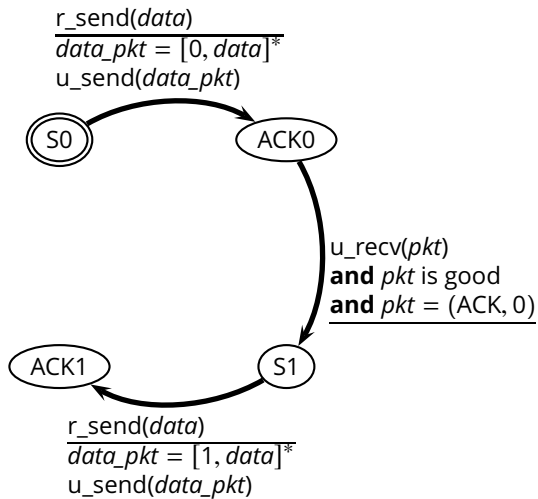
$r_send(data)$
 $\hline data_pkt = [0, data]^*$
 $u_send(data_pkt)$



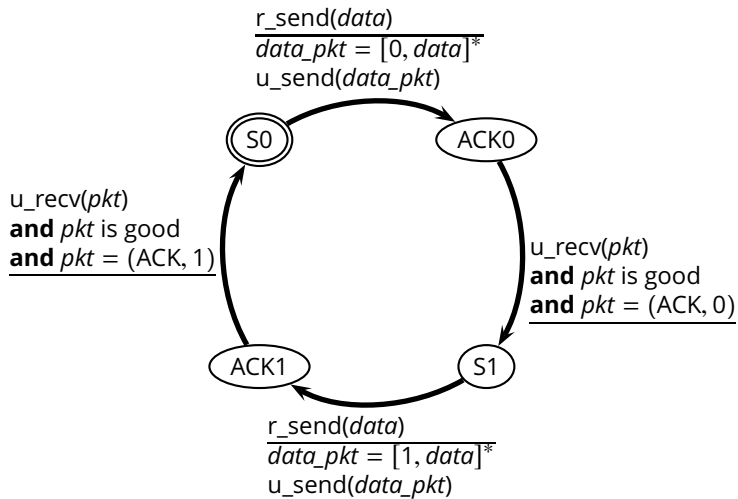
ACK-Only Protocol: Sender



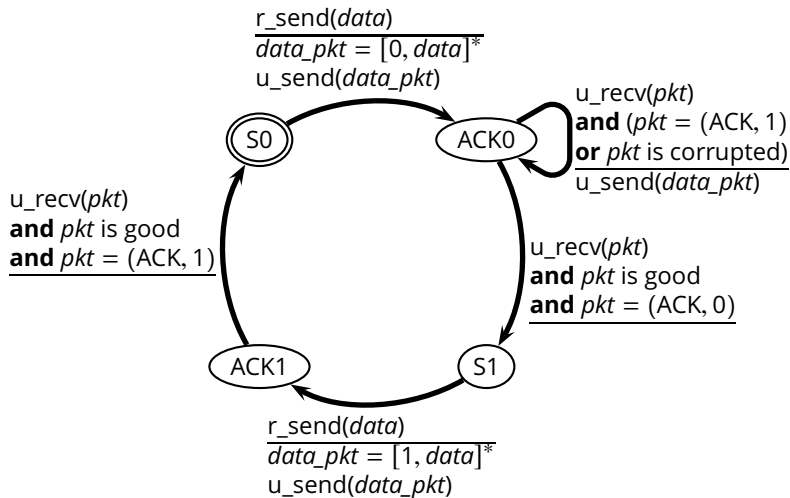
ACK-Only Protocol: Sender



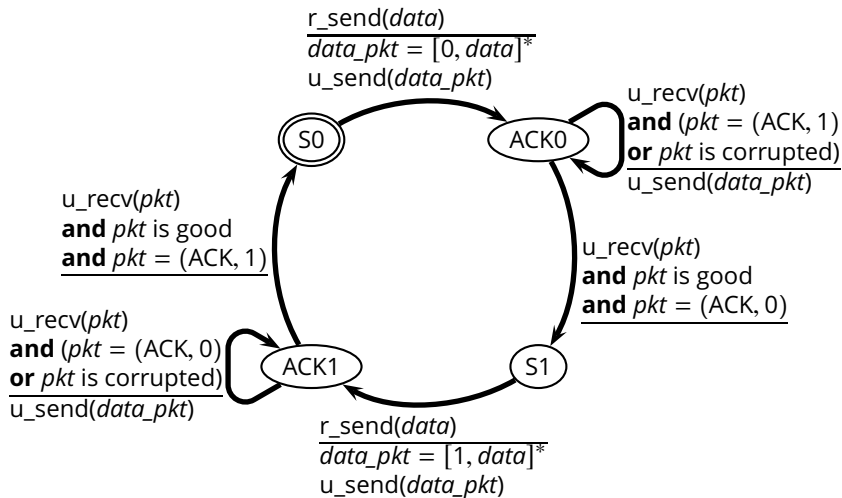
ACK-Only Protocol: Sender



ACK-Only Protocol: Sender



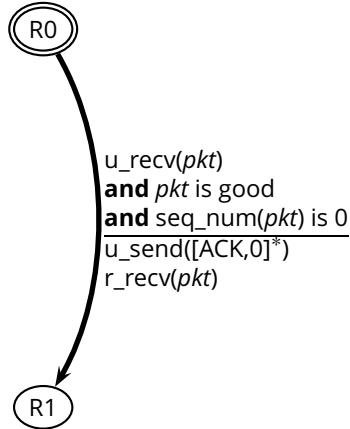
ACK-Only Protocol: Sender



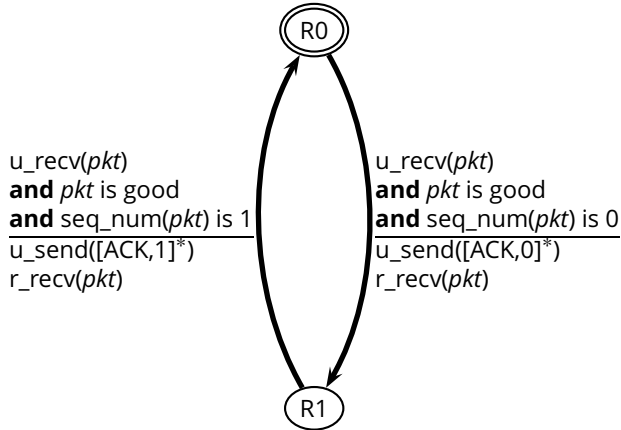
ACK-Only Protocol: Receiver

R0

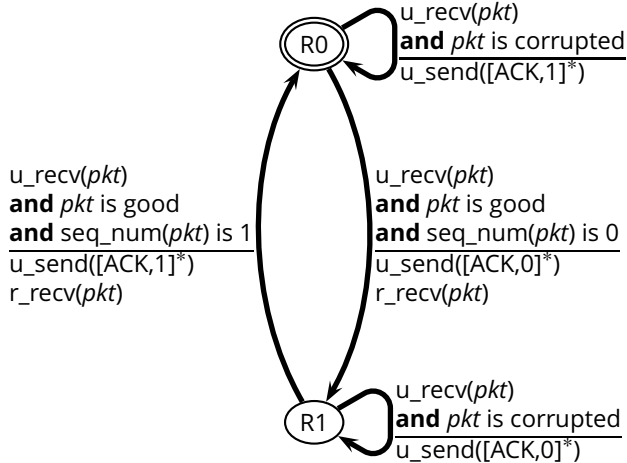
ACK-Only Protocol: Receiver



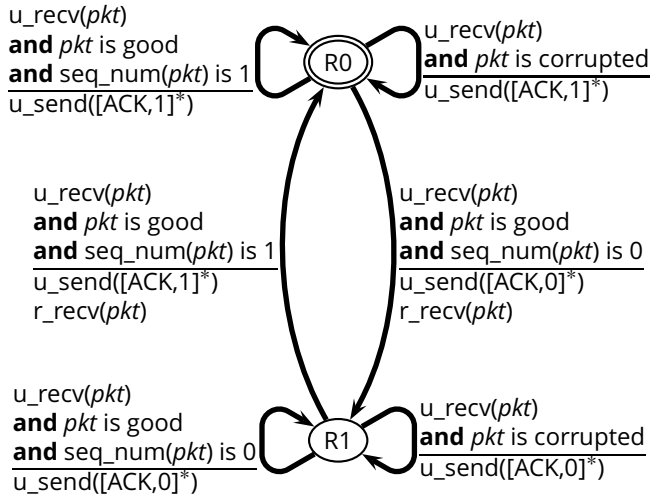
ACK-Only Protocol: Receiver



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*** allow us to recover from transmission errors

Summary of Principles and Techniques

- **Error detection codes** (checksums) can be used to detect transmission errors
- **Retransmission** allow 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** allow 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 interpreted as NACKs, possibly generating duplicate segments

Summary of Principles and Techniques

- **Error detection codes** (checksums) can be used to detect transmission errors
- **Retransmission** allow 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 interpreted as NACKs, possibly generating duplicate segments
- **Sequence numbers** allow the receiver to ignore duplicate data segments

Lossy And Noisy Channel

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

- 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)

- 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?)

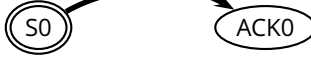
- Reliable transport protocol over a network that may
 - ▶ introduce *bit errors*
 - ▶ *loose packets*
- How do people deal with such situations?
(Think of radio transmissions over a noisy and shared medium. Also, think about what we just did for noisy channels)
- *Detection*: the receiver and/or the 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

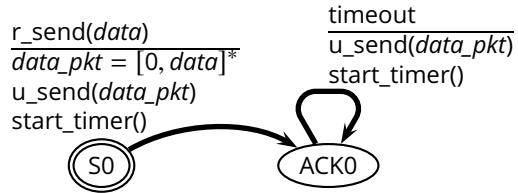


Sender Using Timeouts

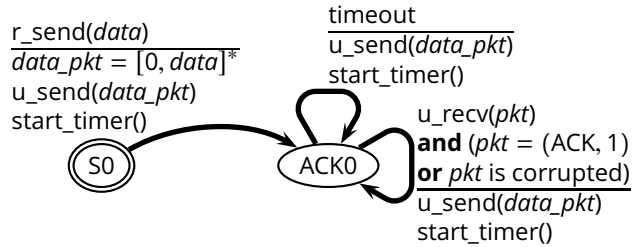
$r_send(data)$
 $\frac{}{data_pkt = [0, data]^*}$
 $u_send(data_pkt)$
 $start_timer()$



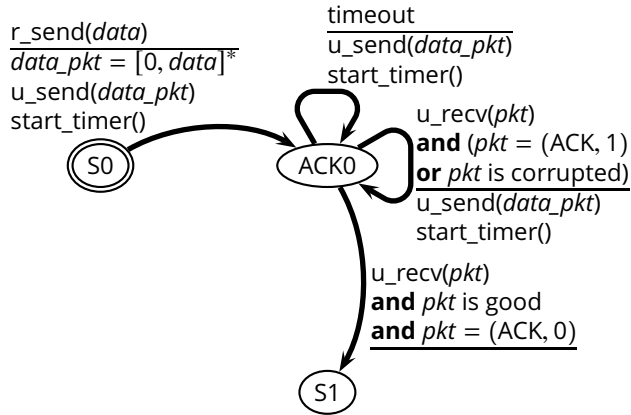
Sender Using Timeouts



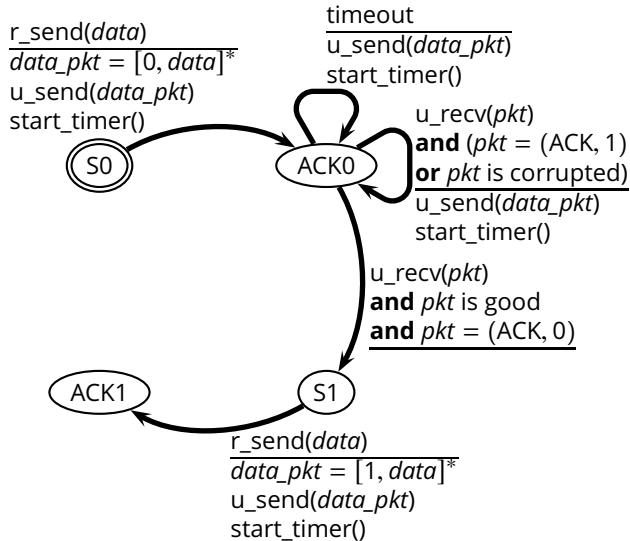
Sender Using Timeouts



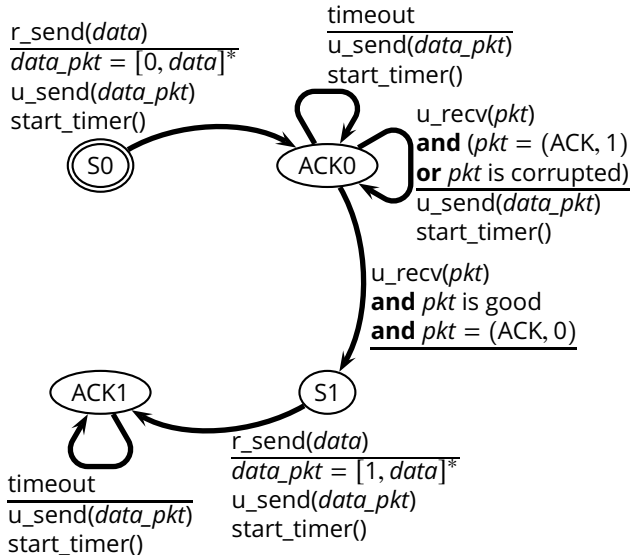
Sender Using Timeouts



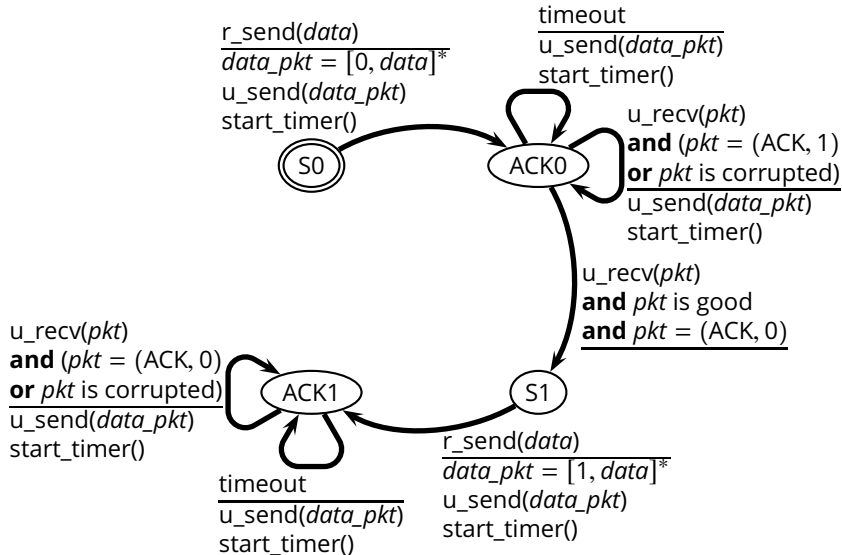
Sender Using Timeouts



Sender Using Timeouts



Sender Using Timeouts



Sender Using Timeouts

