Networks and Communications
"The Transport Layer"

Konstantinos Gkoutzis
Imperial College

Outline

Imperial College London

- Transport Layer Protocols

- TCP

- UDP

- Transport Issues

- ...and solutions

Introduction

- The Transport Layer provides:
  - reliable connection-oriented services
  - unreliable connection-less services
  - parameters for specifying Quality of Service

- The Transport Layer protocols provide for logical communication between application processes

- It runs on *end hosts* only (*not on routers/switches*)
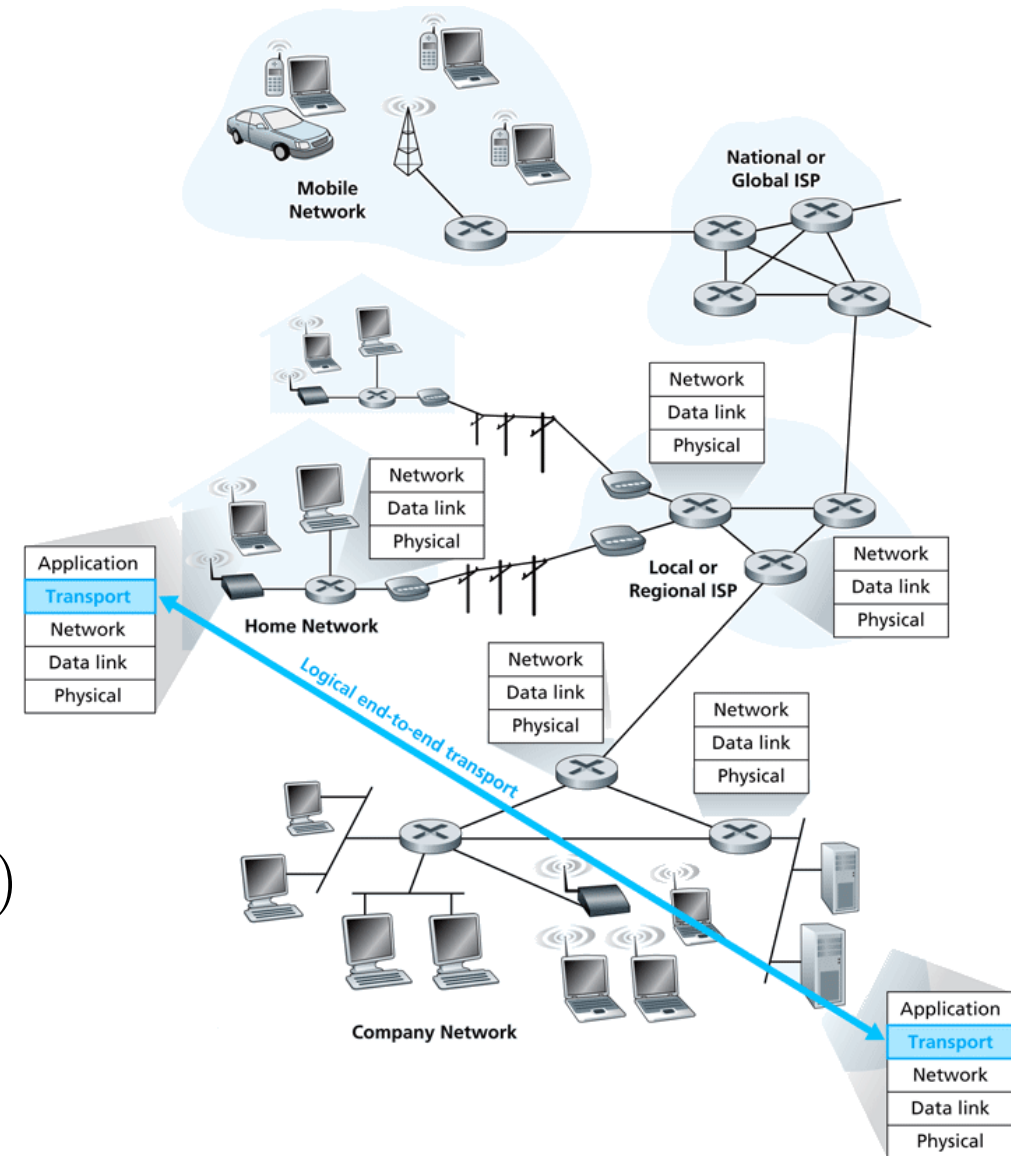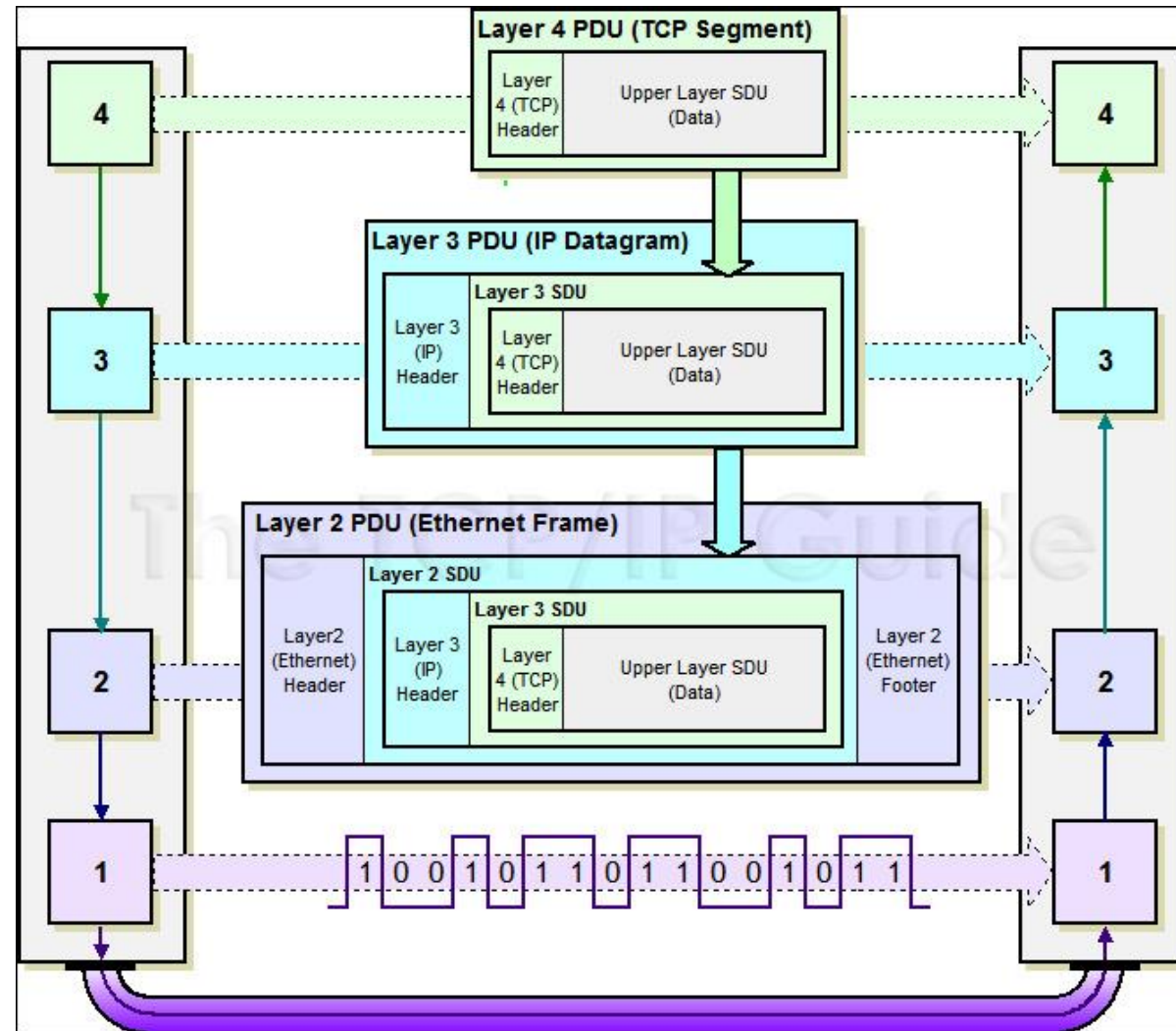
Transport Layer on (top of) the Internet

Imperial College London

- **Transmission** Control Protocol (TCP)
  - connection-oriented

- User **Datagram** Protocol (UDP)
  - connection-less

- Transport Layer's **TCP** data are called **segments**
- Transport Layer's **UDP** data are called **datagrams** (*like telegrams*)

- Basic assumptions for the underlying (Network) layer:
  - every host has one unique IP address
  - IP: "best-effort" delivery service
    - no guarantees on the integrity of data (=*packet*) transmission
    - no guarantees on the order in which packets (*or segments*) are delivered

- *Other Layer 4 protocols: UDP-Lite, DCCP, SCTP(PR), RSVP*

Data Encapsulation

Imperial College London

- Terminology – *An Agreement*

	Application — **Data**
Segmentation (*TCP only*)	**Transport** — **TCP Segments** or **UDP Datagrams**
Fragmentation	**Network/Internet** — **IP Datagrams** (*a.k.a. Packets*)
	Data Link — **Frames**
	Physical — **Bits**

Data Encapsulation (cont'd)

The big picture

Multiplexing/Demultiplexing

Imperial College London

GET / HTTP/1.1

HELO ...

GET /index.html HTTP/1.1

Client host

Server host

- How do we distinguish all these simultaneous connections?

Ports

- Each application running on a host is identified (*within that host*) by a unique port number
  - port numbers are simply **cross-platform process identifiers**

- How do we identify a (*socket*) "connection"?
  - two pairs of (Host + Application) identifiers + Transport Layer protocol
  - i.e. two pairs of **IP_address + Port_number + TCP/UDP**
  - e.g. **146.179.40.24**:**80 TCP** ⇔ **192.168.1.1**:**7155 TCP**

- How do we find out which application (*host and port number*) to connect to?
  - this is outside the scope of the definition of the Transport Layer
  - but we do have known and "well-known" port numbers
  - e.g. 80 for HTTP, 25 for SMTP, 22 for SSH, and more
  - the first 1024 ports (0 - 1023) are "well-known" (*i.e. reserved*)

Transport Layer services/features

- Transport-layer multiplexing/**de**multiplexing
  - connecting applications (*as opposed to hosts*)

- **Reliable** data transfer
  - integrity and (*possibly*) ordered delivery

- Connections
  - streams
  - *can be seen to be equivalent to "ordered delivery"*

- Congestion control
  - end-to-end traffic (admission) control
  - to avoid destructive congestion within the network

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 initially "shake hands" to establish a connection
  - *not a circuit-switched connection, nor a virtual circuit*

- Full-duplex service
  - both endpoints can **send and receive at the same time**

Transport Layer Interface

Imperial College London

- Consider the Berkeley **socket interface**, which has been adopted by all UNIX systems (*as well as Windows*)

SOCKET Create a new communication endpoint
BIND Attach a local address to a socket

- The client and server each bind a transport-level address and a name to the locally created socket

LISTEN Announce willingness to accept N connections

- The server starts listening on this socket, thus telling the kernel that it will now wait for connections from clients

ACCEPT Block until some remote client wants to establish a connection

- After this, the server can accept or select connections from clients

CONNECT Attempt to establish a connection

- A client connects to the socket; it needs to provide the full transport-level address to successfully locate the socket
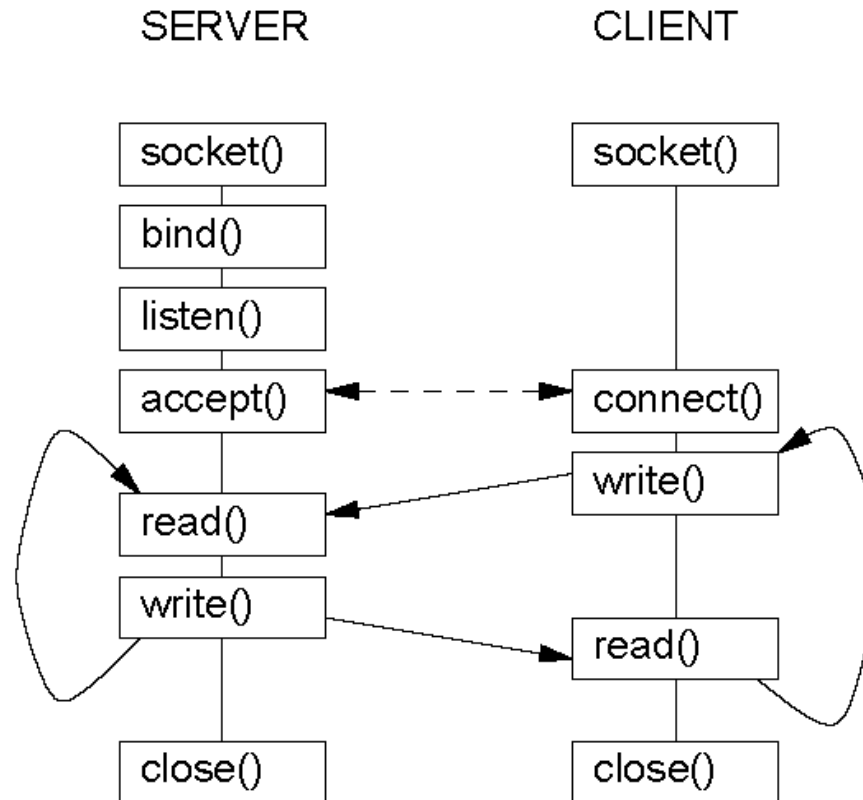
SEND Send data over a connection
RECEIVE Receive data over a connection

- Now the client and server communicate through send/receive operations on their respective sockets

CLOSE Release the connection

- Communication ends when a connection/socket is closed

Connection-Oriented Socket Communication

Question:
What about connection-less communication ?

Answer:
There is no connection, i.e. no need for **listen**, **accept**, and **connect**

Transmission Control Protocol (TCP) Client

```
public class Client {
        public static void main (String[] args) throws UnknownHostException, IOException {
                Socket socket = new Socket ("127.0.0.1", 2259);
                BufferedReader in = new BufferedReader (new InputStreamReader
                                                    (socket.getInputStream()));
                PrintWriter out = new PrintWriter (new OutputStreamWriter
                                                    (socket.getOutputStream()), true);
                System.out.println (System.console().readLine());
                System.out.println (in.readLine());
        }
}
```

- The **Socket** constructor also implicitly *connects* the TCP socket to the **IP address** and **port** specified
- For UDP, you would need to use the **DatagramSocket** class instead
- You can read/write Sockets using the standard operations to access *files*

More at: Introduction to [JavaSE Networking Tutorial](#)

Transmission Control Protocol (TCP) Server
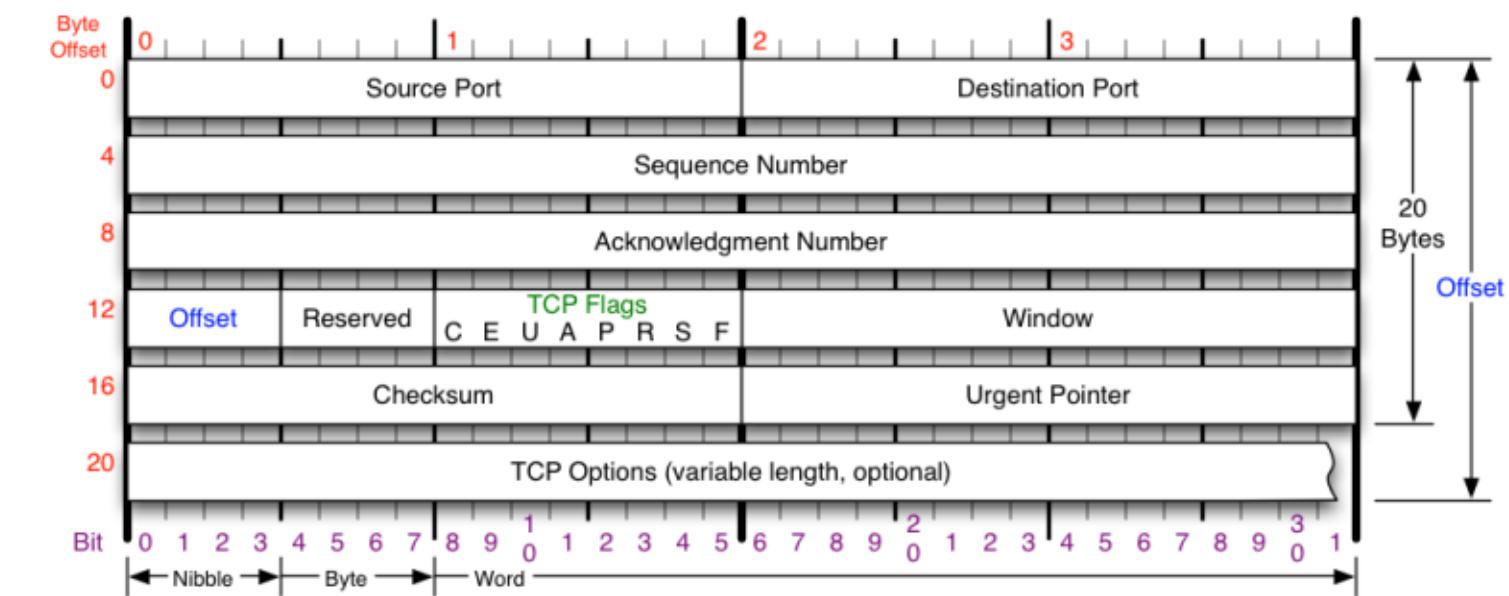
```java
public class Server {
        public static void main (String[] args) throws IOException {
                ServerSocket serverSocket = new ServerSocket (2259);
                System.out.println ("Listening on port 2259...");
                while (true) {
                        Socket socket = serverSocket.accept();
                        BufferedReader in = new BufferedReader (new InputStreamReader
                                        (socket.getInputStream()));
                        PrintWriter out = new PrintWriter (new OutputStreamWriter
                                        (socket.getOutputStream()), true);
                        System.out.println (in.readLine());
                        System.out.println (System.console().readLine());
                        socket.close();
                }
        }
}
```

- The **ServerSocket** constructor also implicitly binds the TCP socket to the port
- The equivalent Java class for UDP is **DatagramSocket**
- You can read/write Sockets using the standard operations to access files
- To handle multiple clients at the same time, **a new thread must be created** for each new client connection

Preliminary Definitions

Imperial College London

- **TCP Segment**: "envelope" for TCP data
  - TCP data are transmitted within TCP segments
  - TCP segments are transmitted within a Network Layer protocol (*e.g. IPv4*)

- **Maximum Segment Size** (**MSS**): maximum amount of application data transmitted in a single segment (*headers not included*)
  - typically related to the MTU of the connection, to avoid network-level fragmentation

- **Maximum Transmission Unit** (**MTU**): largest link-layer frame available to the sender host
  - *Path MTU Discovery (PMTUD): determine the largest link-layer frame that can be sent on all links from the sender host to the receiver host*

Looking inside Layer 4: TCP

Image from the [Nmap book](#)

TCP Header Fields

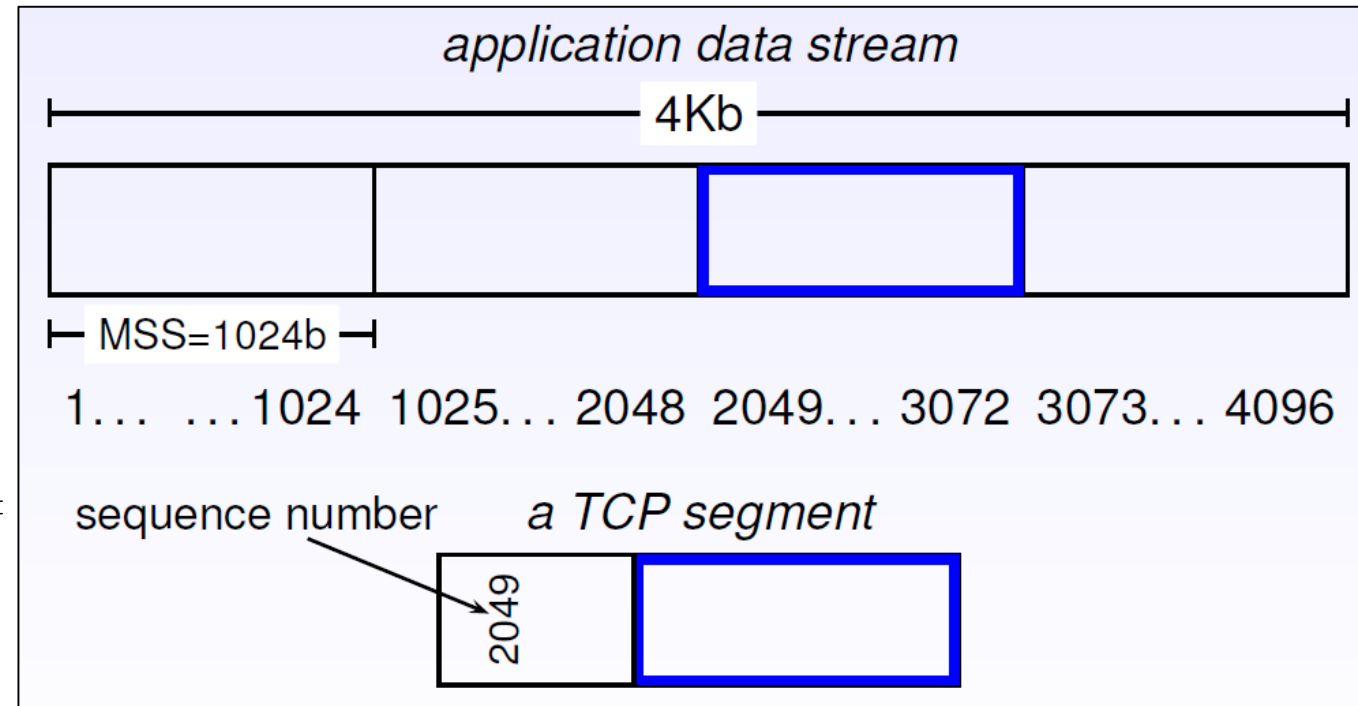**Imperial College London**

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

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

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

- *Receive window (16-bit)*: size of the "window" on the receiver end

- *Header length / offset (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 (cont'd)

- *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 are marked by the urgent data pointer field

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

- *PSH flag (1-bit)*: "push" flag, used to solicit the receiver to pass the data to the application immediately

- *RST flag (1-bit)*: used during connection setup and shutdown

- *SYN flag (1-bit)*: used during connection setup and shutdown

- *FIN flag (1-bit)*: used during connection shutdown

- *Checksum (16-bit)*: used to detect transmission errors

Sequence Numbers

Imperial College London

- Sequence numbers are associated with bytes in the data stream
  - *not* with segments, i.e. it is not a packet numbering system per se

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

- When the TCP connection is set up, a random **I**nitial **S**equence **N**umber (ISN) is decided upon, in order to avoid receiving any leftover segments by mistake

- The ISN is used to initialise the SEQ#

application data stream

4Kb

MSS=1024b

1... ...1024 1025... 2048 2049... 3072 3073... 4096

sequence number   *a TCP segment*

2049

Acknowledgement Numbers

Imperial College London

- An *acknowledgement number* represents the *first sequence number not yet seen by the receiver*
    - TCP acknowledgements can be "cumulative"
    - Typically, TCP implementations *ack* every *other* packet
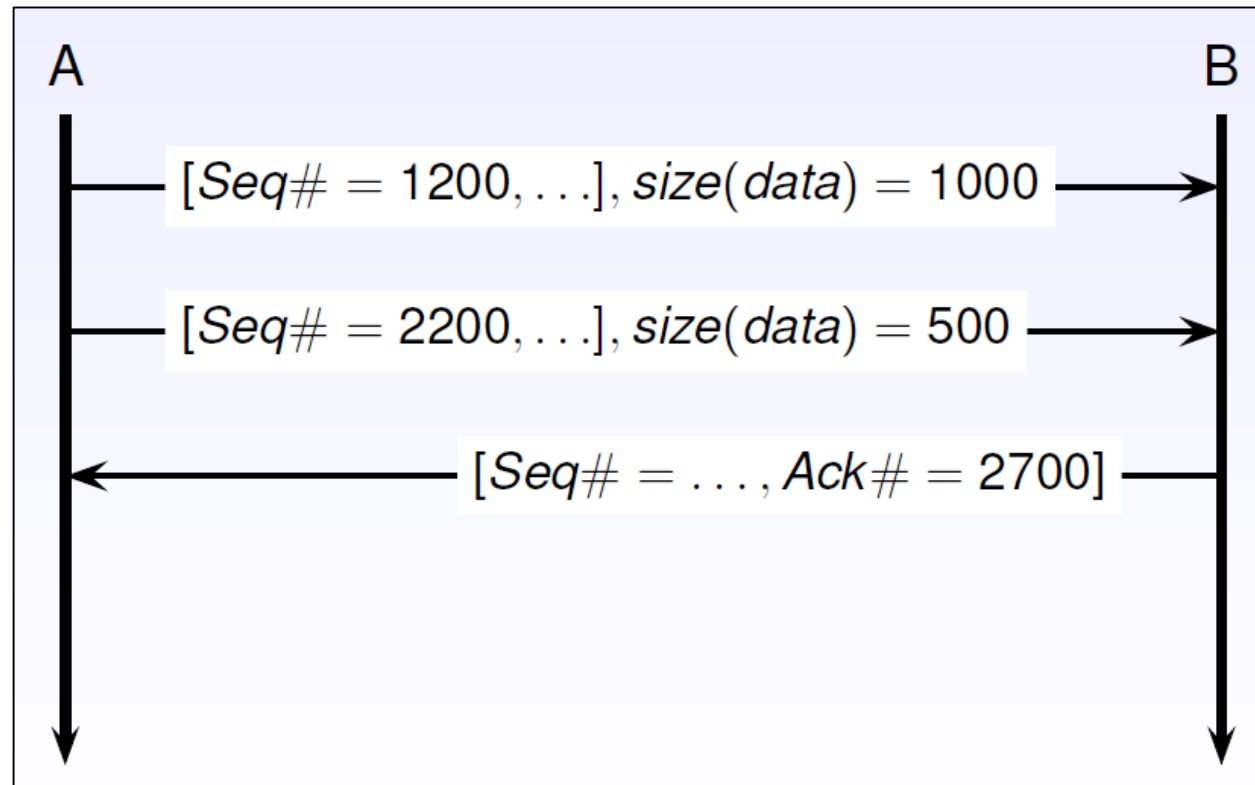
Sequence Numbers and ACK Numbers

Imperial College London

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

- For example, consider a simple "Echo" application:
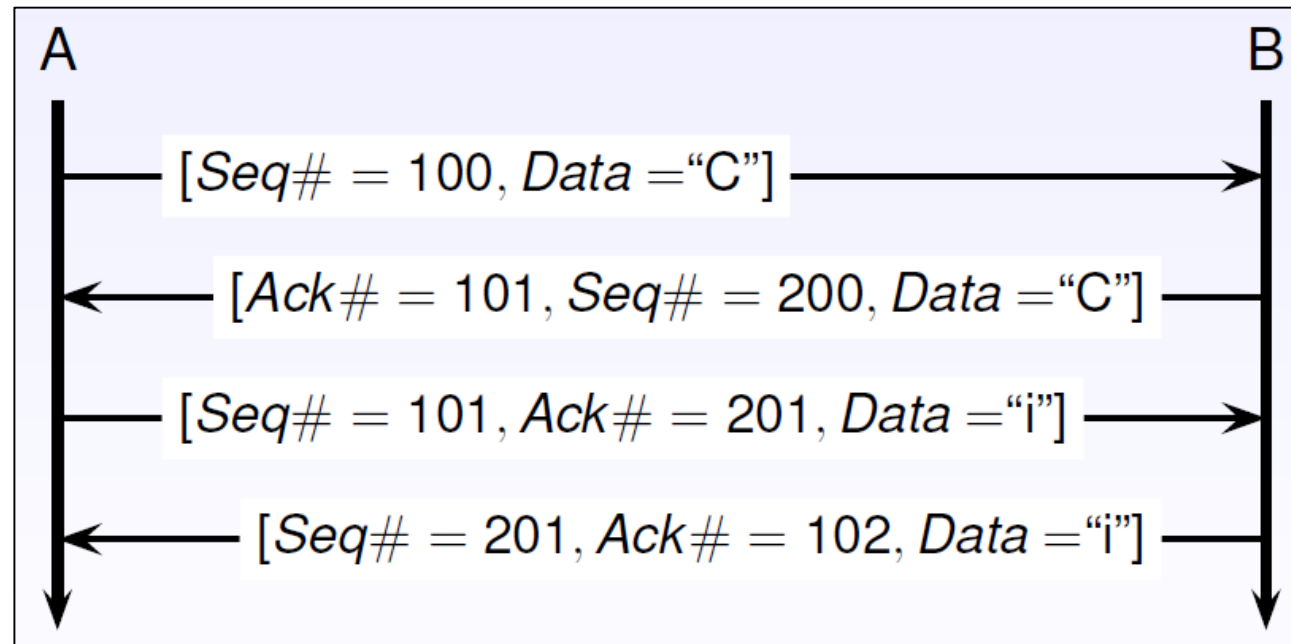
- Acknowledgments are "*piggybacked*" on data segments

Three-way Handshake

- The client sends a TCP segment with the SYN flag set to true
  - and also its initial sequence number

- The server responds with another SYN TCP segment
  - which also has the ACK flag set to true, and the first unseen client SEQ#
  - as well as an initial sequence number for the server

- Finally, the client responds with an ACK
  - including the first unseen server SEQ#
  - and the client's new SEQ#

- A *similar* process is used to disconnect
  - instead of SYN, we use FIN
  - *not necessarily three exchanges*

Three-way Handshake (cont'd)

Imperial College London
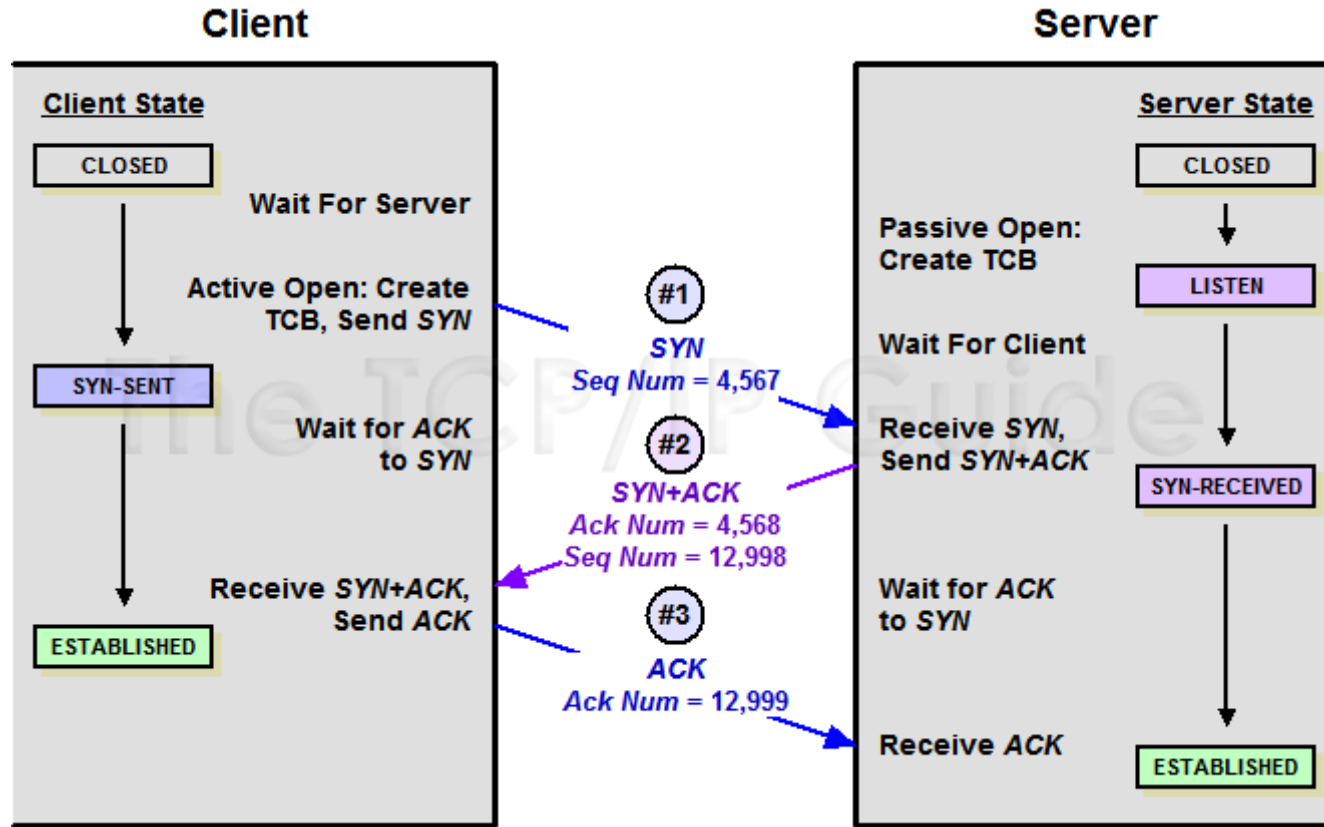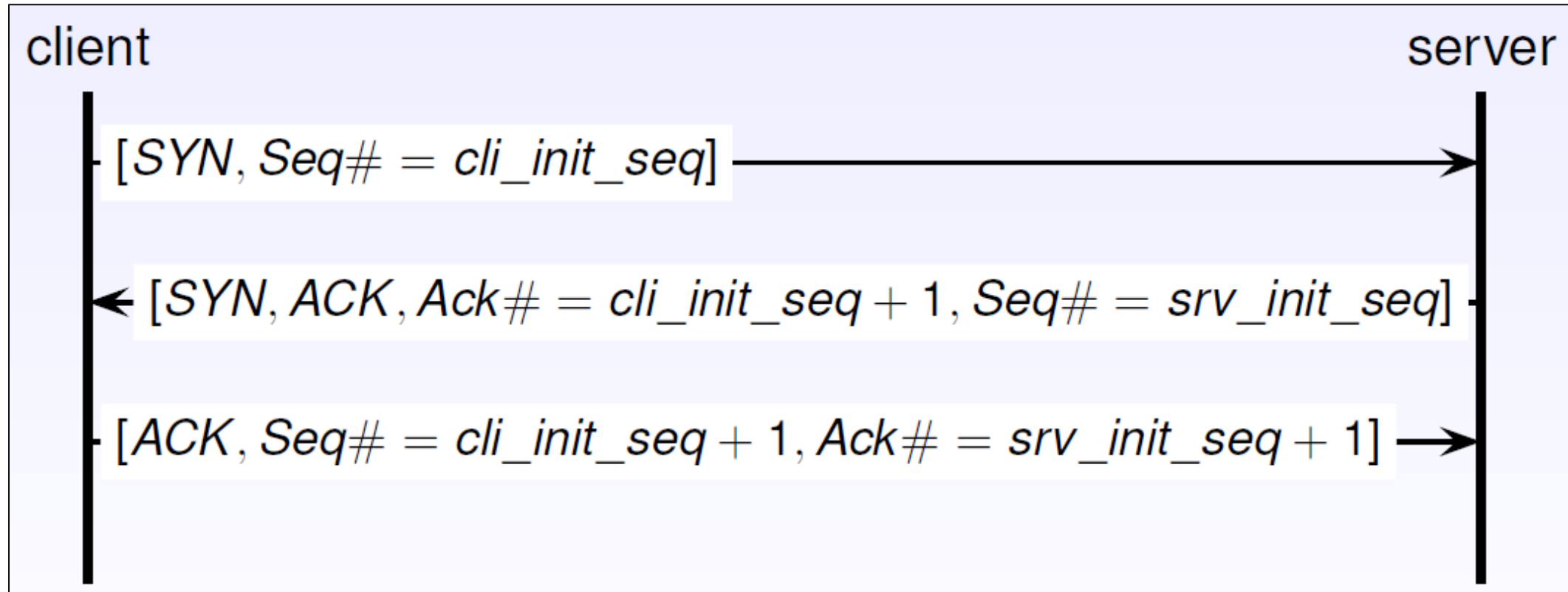

Connection Establishment
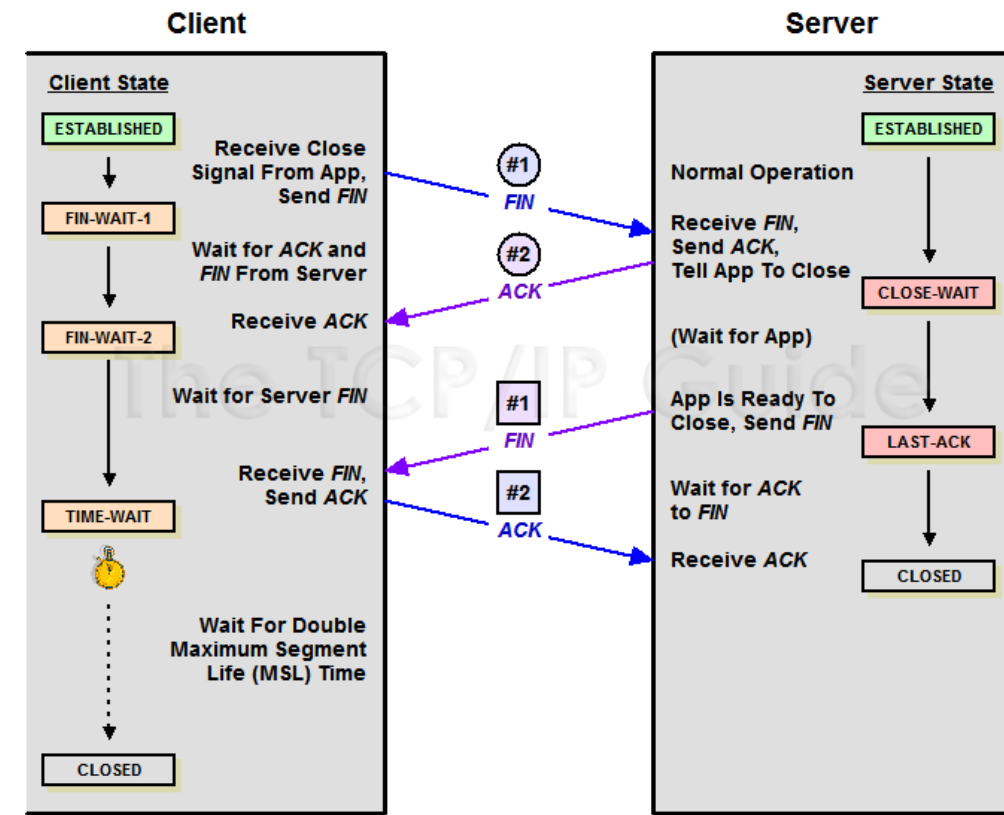
Connection Termination

Image source: [The TCP/IP Guide](#)

Three-way Handshake (cont'd)

- So, to generalise:

UDP Features

Imperial College London

- UDP provides only the two most basic functions of a transport protocol
  - application identification (multiplexing/demultiplexing)
  - integrity check by means of a CRC-type checksum

- UDP is simple:
  - no flow control
  - no error control
  - no retransmissions

Question:
So why not just use IP instead?
Answer:
Because we still need the **port number** fields to deliver the datagram to the correct application

- UDP datagrams cannot be larger than 65 K
  - *20B IP header + 8B UDP header + 65,507B data* = 65,535 Bytes
    - that is the maximum IP packet/datagram size
  - Although, in practice, 500 to 1,000 Byte datagrams are used
    - the smaller they are, the more likely they are to make it intact!

User Datagram Protocol (UDP) Client

Imperial College London

```java
public class UDPClient {
        public static void main (String[] args) throws IOException {
                byte buf[] = System.console().readLine().getBytes();
                DatagramPacket packet = new DatagramPacket (buf, buf.length,
                                        InetAddress.getByName ("127.0.0.1"), 2259);
                DatagramSocket socket = new DatagramSocket();
                socket.send (packet); // no connection needed
                buf = new byte[256];
                packet = new DatagramPacket(buf, buf.length);
                socket.receive (packet); // fingers crossed
                System.out.println (new String(packet.getData()));
        }
}
```

- UDP: connection-less protocol
  - no need to connect; you just send the message
  - each datagram packet must carry the full *address:port* of the recipient

User Datagram Protocol (UDP) Server

Imperial College London

```java
public class UDPServer {
        public static void main (String[] args) throws IOException {
                DatagramSocket socket = new DatagramSocket (2259);
                while (true) {
                        byte buf[] = new byte[256];
                        DatagramPacket packet = new DatagramPacket (buf, buf.length);
                        socket.receive (packet); // receive a message from a client
                        String s = new String (packet.getData(), 0, packet.getLength());
                        System.out.println (s);
                        buf = System.console().readLine().getBytes();
                        InetAddress clientAddress = packet.getAddress();
                        int clientPort = packet.getPort();
                        packet = new DatagramPacket (buf, buf.length, clientAddress, clientPort);
                        socket.send (packet); // respond to the client
                }
        }
}
```

- UDP: connection-less protocol
    - no need to acknowledge or stay connected
    - each datagram packet must carry the full *address:port* of the recipient
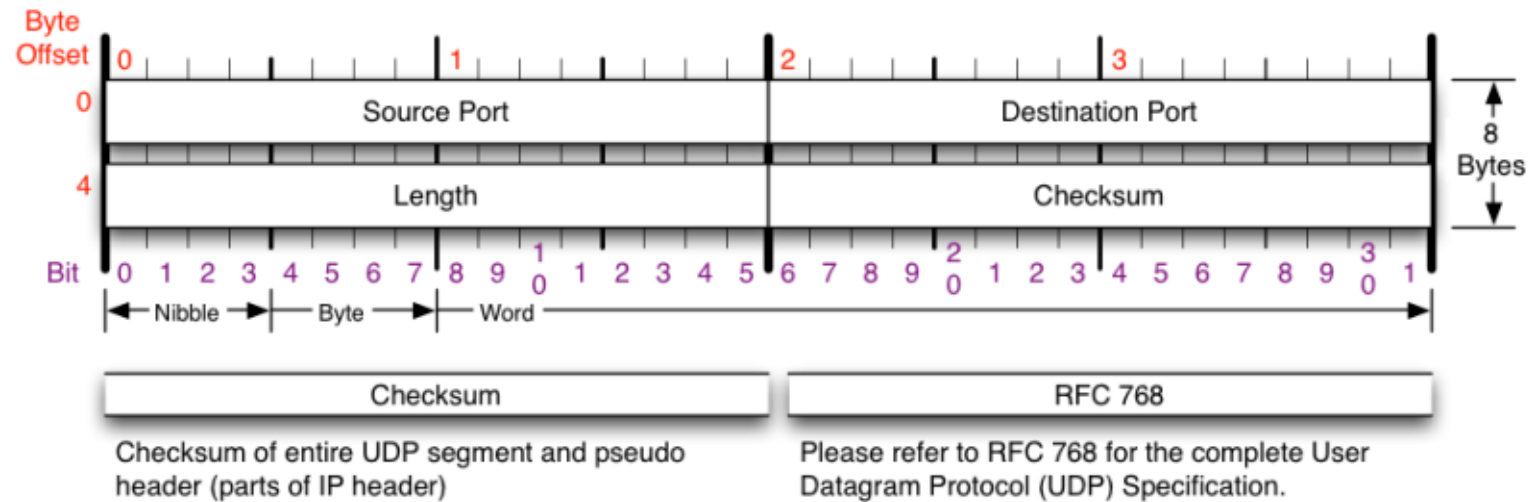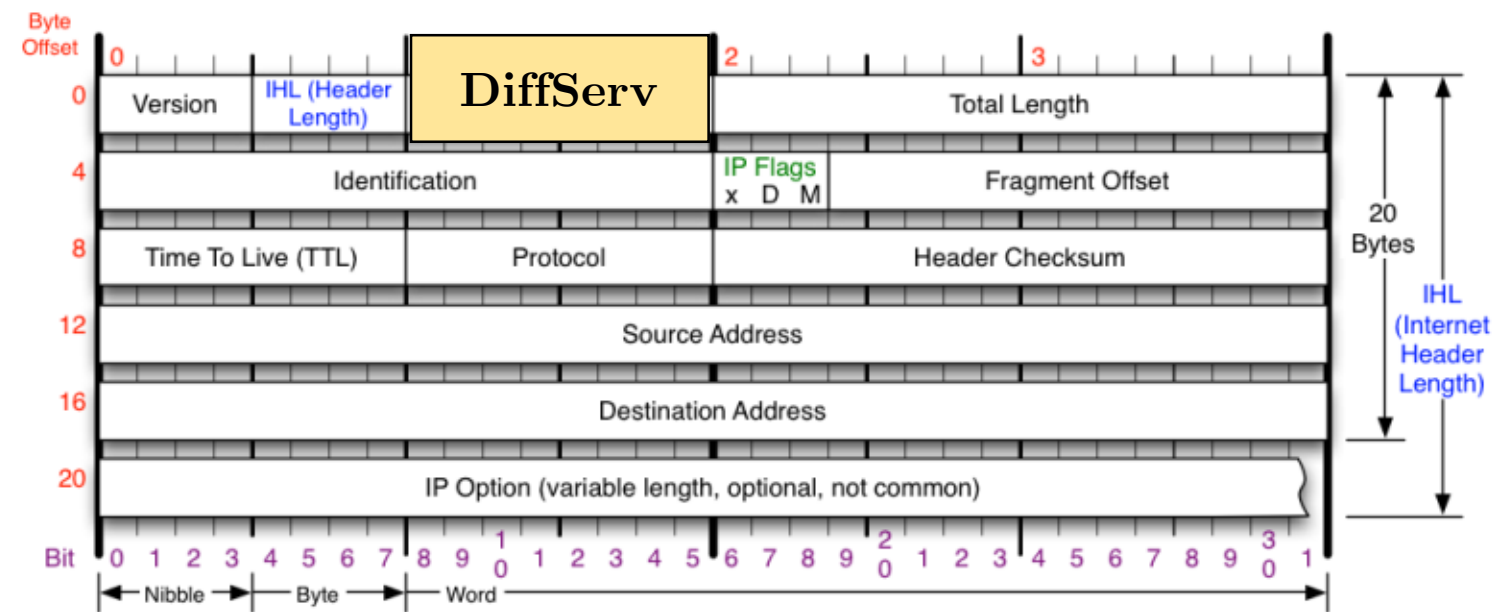
Looking inside Layer 4: UDP

Image from the Nmap book

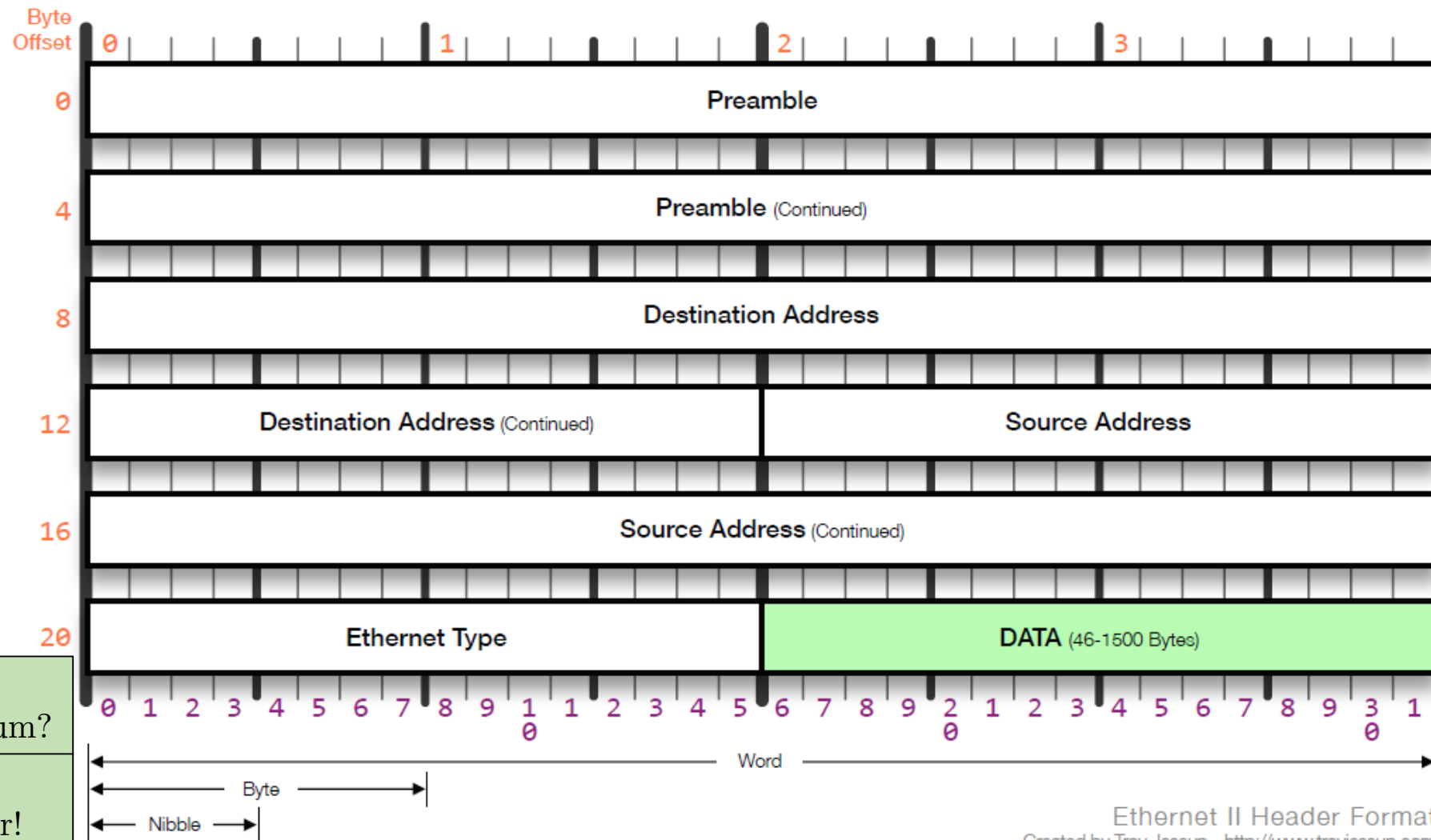Networks and Communications

UDP: When shall we use it?

- A typical question is: "Why should people ever use UDP (*instead of TCP*), since it does not provide any reliability or flow control?"

- Here are some reasons
  1. finer Application Level control over what data are sent and when (*e.g. real-time, Skype, etc.*)
  2. no connection establishment (*faster than TCP*)
  3. no connection state
  4. small packet header overhead

- Beside real-time apps, one area where UDP is really useful is in *client-server* situations
  - often, the client sends a short request to the server and expects a short reply back
  - if either the request or reply is lost, the client can just time-out and try again
  - simpler code and fewer messages required
  - e.g. DNS query: symbolic address (*hostname*) => IP address translation

IPv4 Header (for reference)

Ethernet II Header (for reference)

Imperial College London

Question:
What; no checksum?
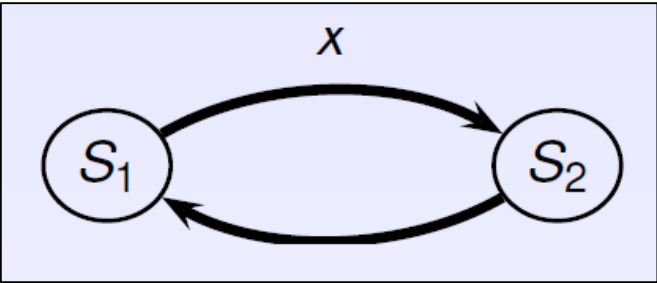
Answer:
It's (in) the footer!

Jokes Time!

Imperial College London
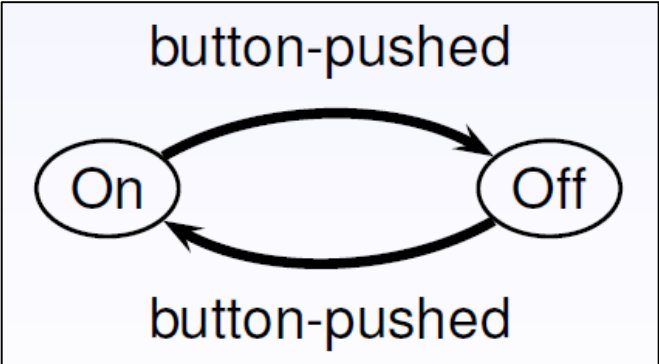
- A TCP packet walks into a bar and says: "I want a beer"
  - The barman says: "You want a beer?"
    - and the TCP packet says: "Yes, I want a beer"

- I'd tell you a UDP joke,
  - but you might not get it.

- "Knock knock."
  - "Who's there?"
    - "SYN flood."
      - "SYN flood who?"
        - "Knock knock."

- *And a Data Link one:*
  - Q: How do you catch an Ether Bunny?
    - A: With an Ethernet!

Sources + more jokes: [*1*] [*2*] [*3*]

Finite-State Machines

Imperial College London

- 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

- States are represented as nodes in a graph:

- Transitions are represented as directed edges in the graph an edge labeled x going from state S1 to state S2 says that when the machine is in state S1 and event x occurs, the machine switches to state S2

FSMs to Specify Protocols

- States represent the state of a protocol

- Transitions are characterised 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"
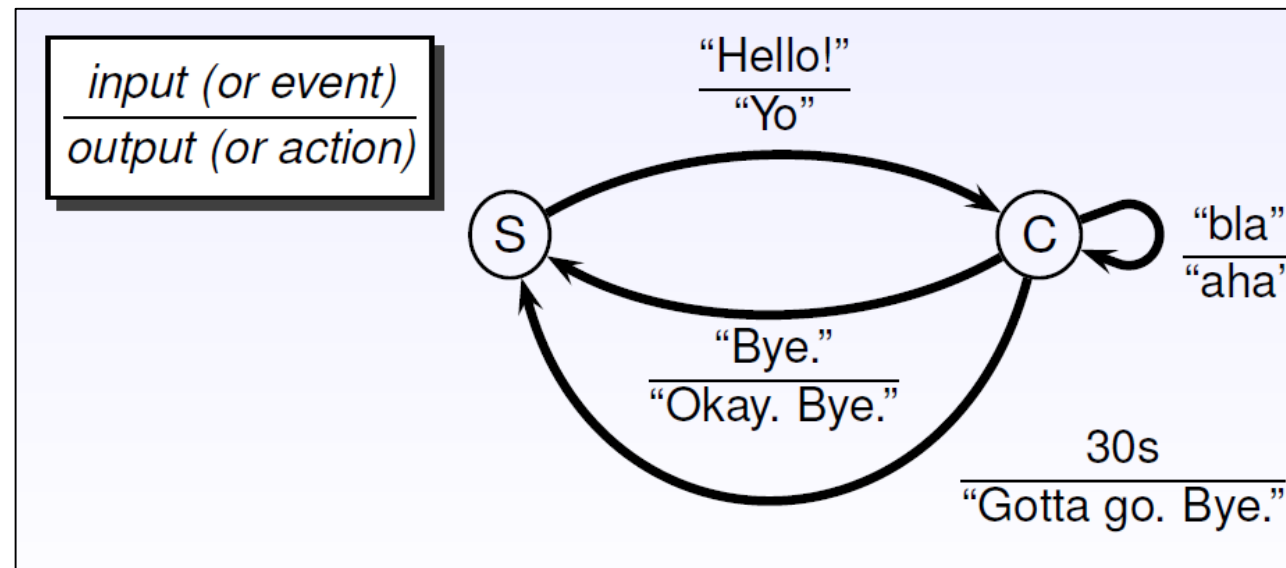
The TCP State Machine (Client)

Imperial College London
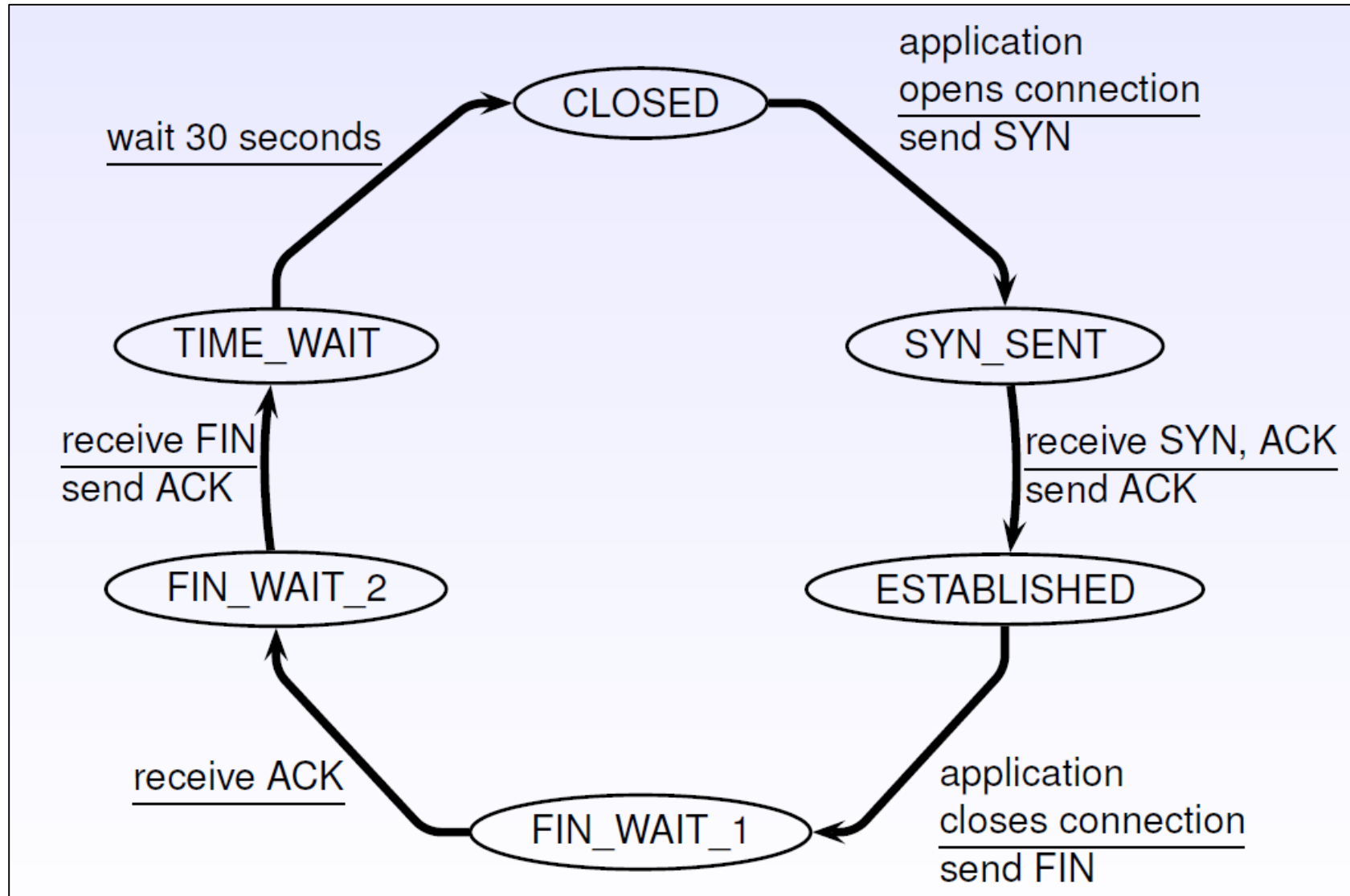
The TCP State Machine (Server)

Networks and Communications
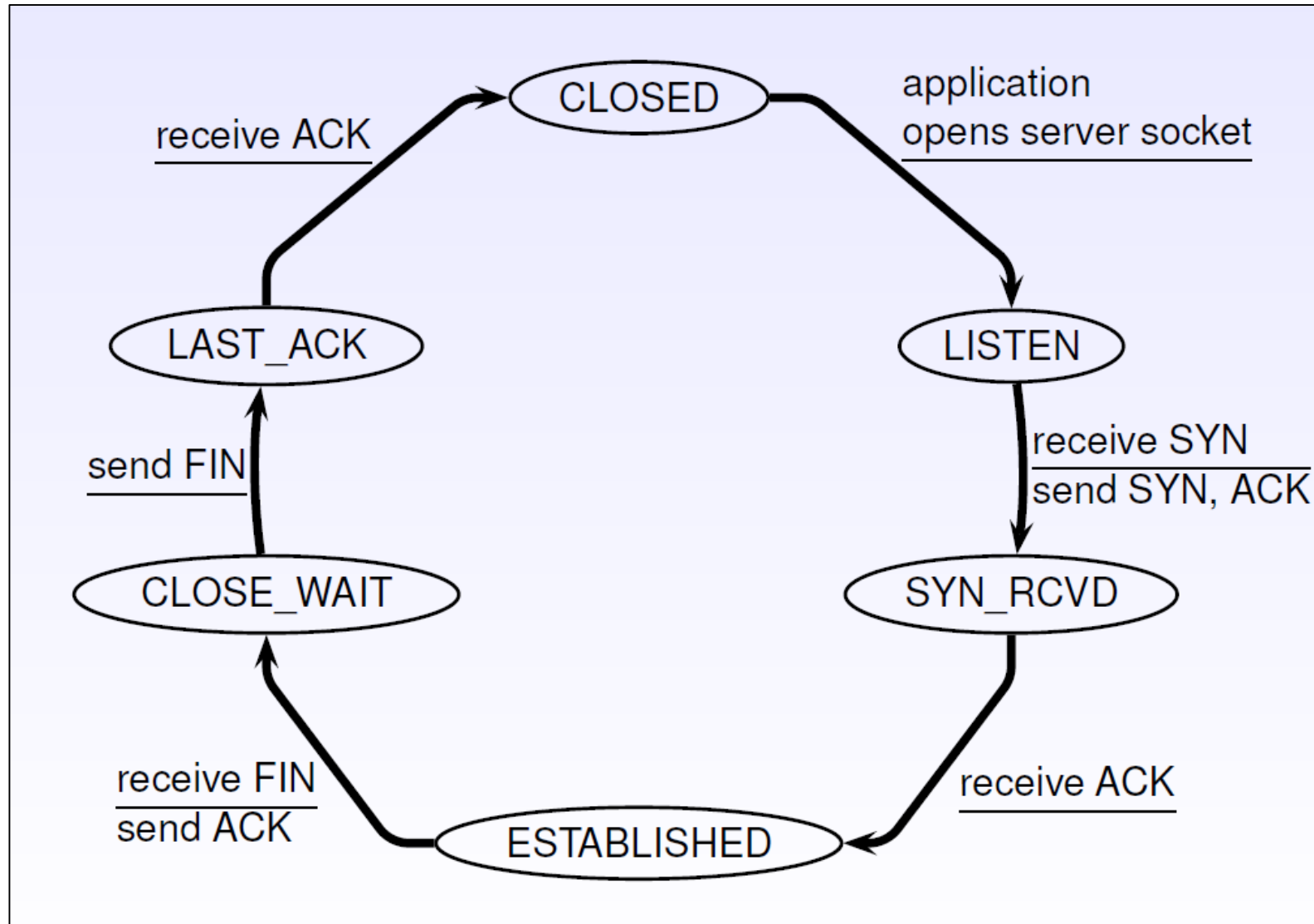
Reliable Data Transfer

- Service model and implementation
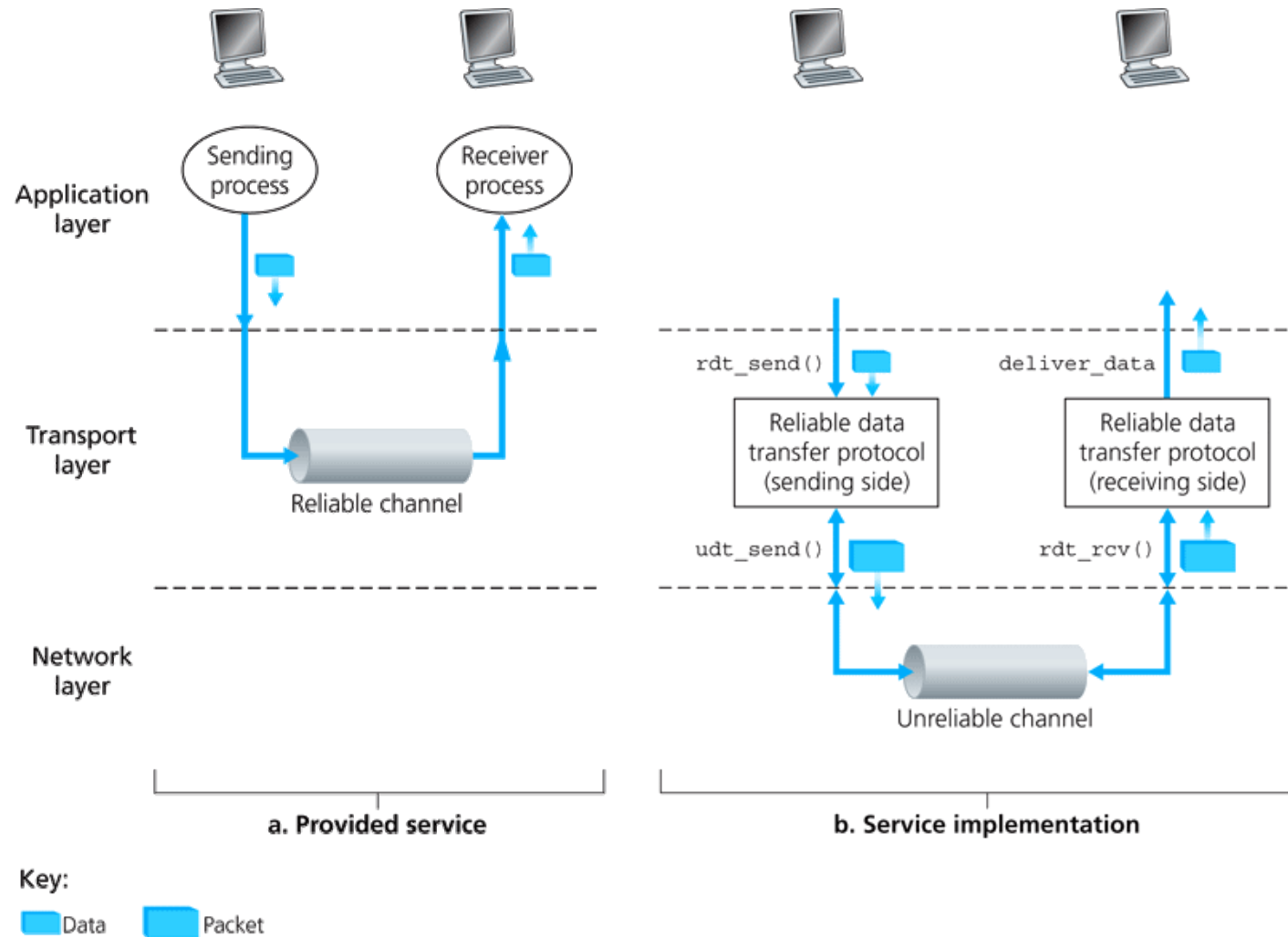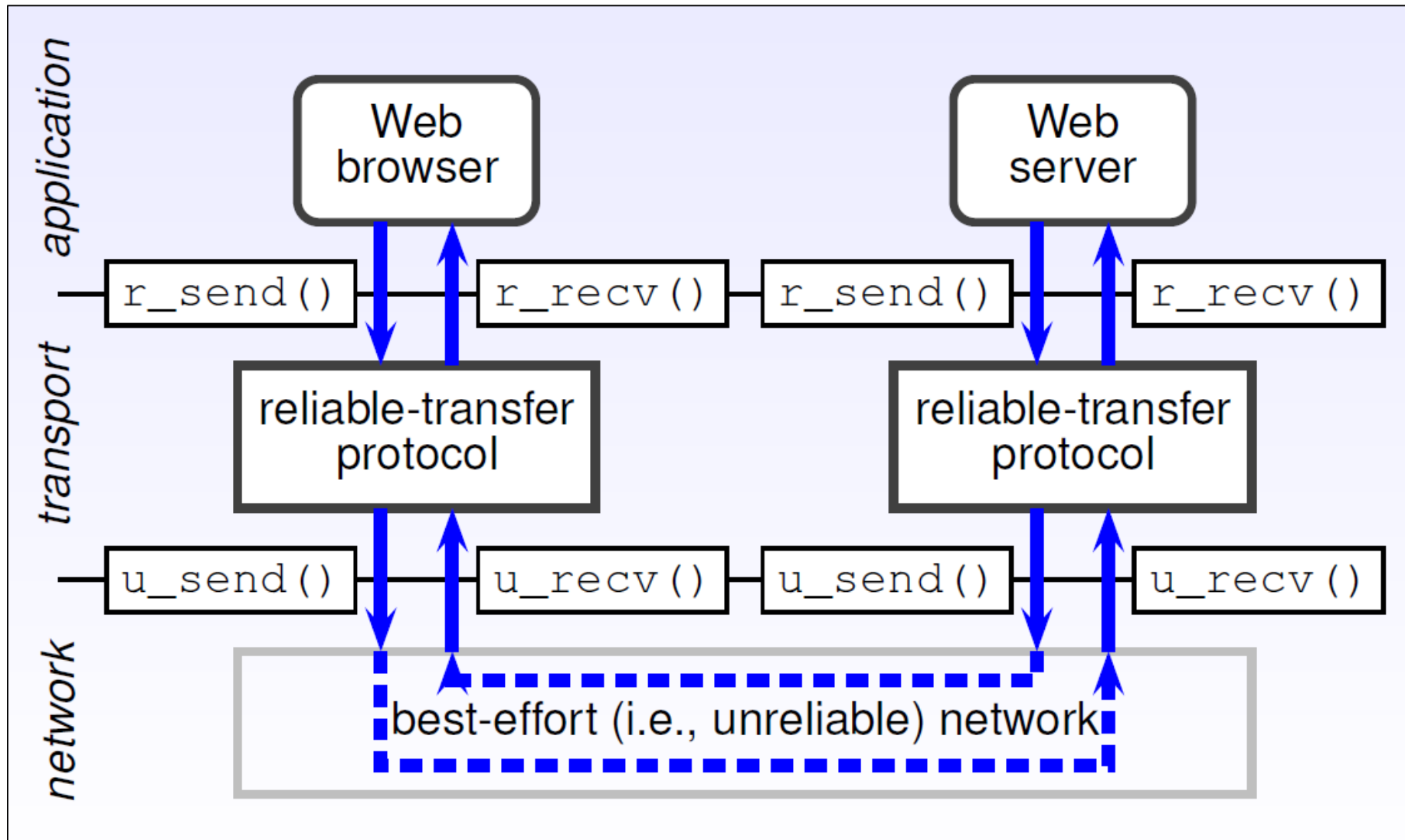
Reliable Data Transfer (cont'd)

Reliable Data Transfer (cont'd)

Imperial College London
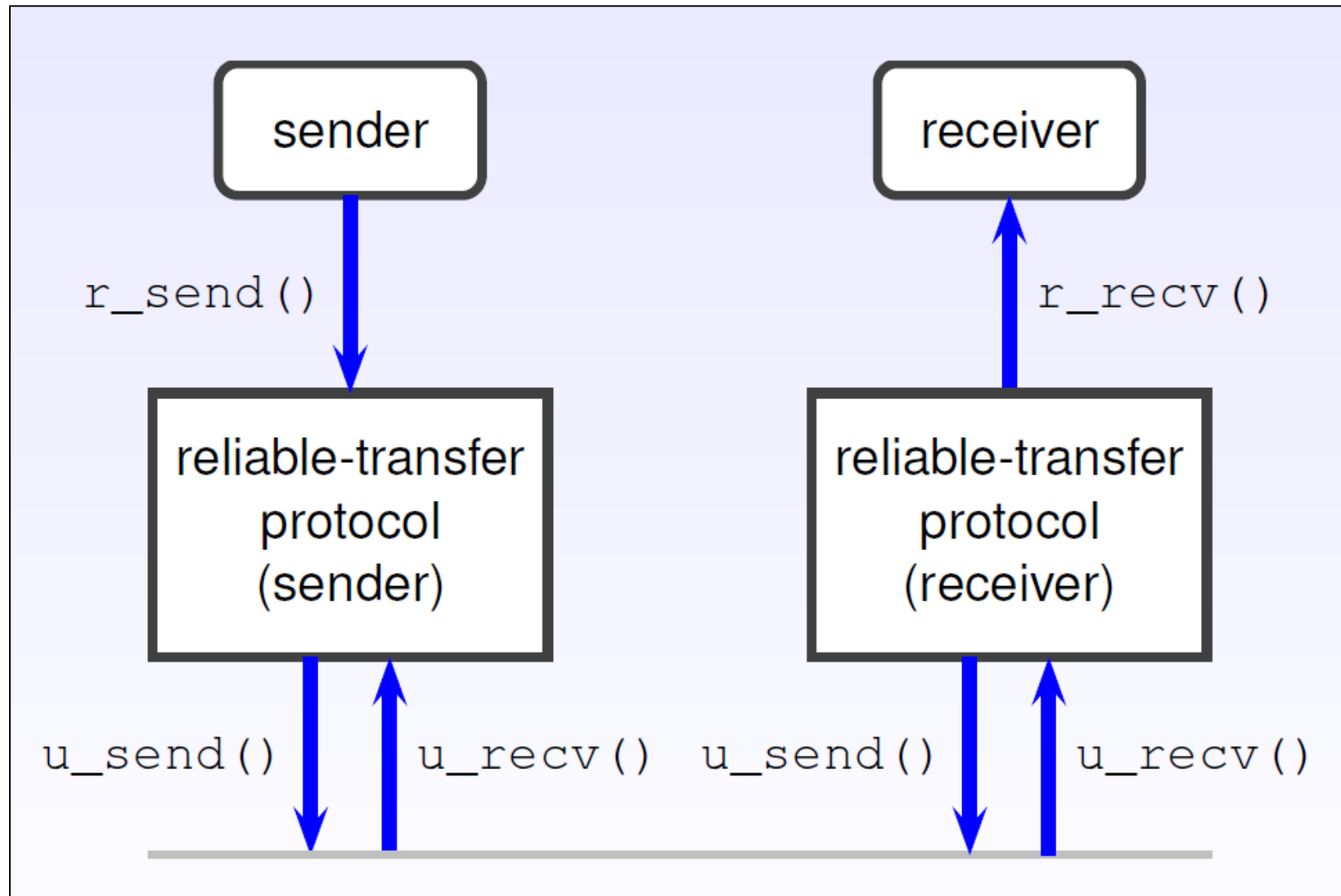
Reliable Data Transfer (cont'd)

- So, a baseline reliable transport protocol could be described using this diagram:

Noisy Channel

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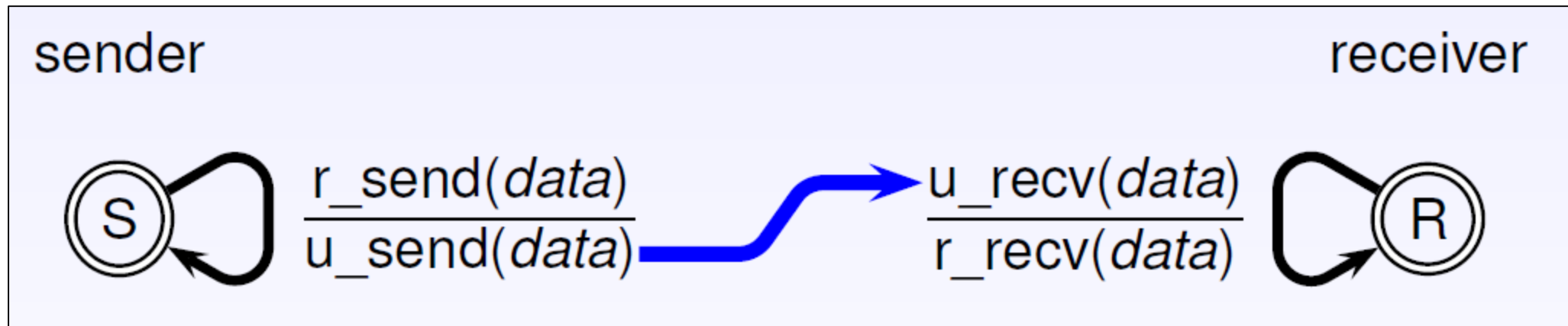
- Reliable transport protocol over a network with *bit errors*
  - every so often, a bit will be modified during transmission
    - that is, a bit will be "flipped"
  - however, no packets will be lost

- 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 must retransmit corrupted packets

Error Detection

Imperial College London

- Idea: how about 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. a *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 was not the expected

- Sender: message is   1001   => send 1001<u>0</u>
- Receiver: receives   1011<u>0</u> => error!

Noisy Channel (cont'd)

- Sender:

[data] indicates a packet containing data + an error-detection code (i.e. a **checksum**)*
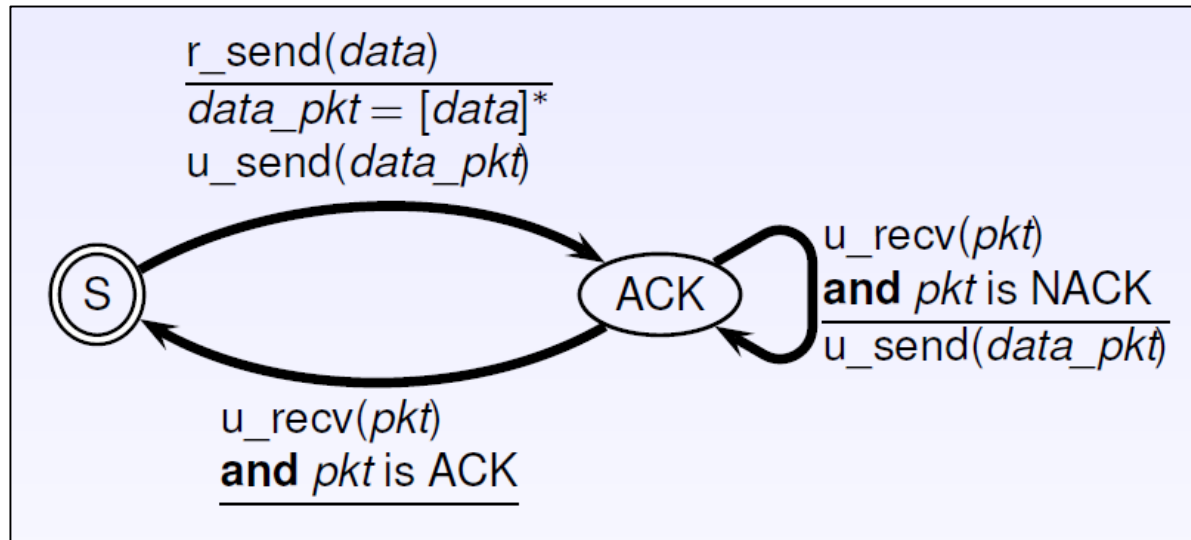
- Receiver:
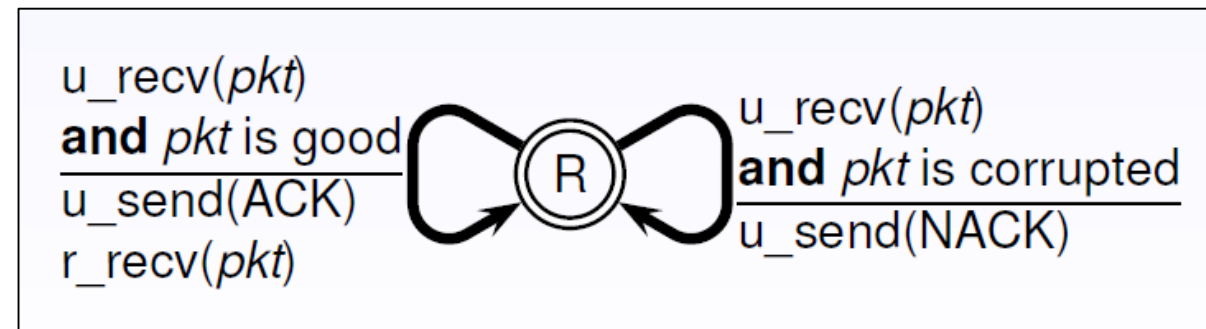
Noisy Channel (cont'd)

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

- Does the protocol *really* work though?

- What happens if an error occurs within an ACK/NACK segment?

Dealing With Bad ACKs/NACKs

Imperial College London

- Negative acknowledgments for ACKs and NACKs
    1. sender says: "Let's go watch a movie"
    2. receiver hears: "Let's . . . a . . . "
    3. receiver says: "Repeat message!"
    4. sender hears: ". . . (*noise*) . . . "
    5. sender says: "Repeat your ACK please!"
    6. . . .
    - Not Good: this protocol does not 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 only for reliable 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

- 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 watch a movie"
  2. receiver hears: "7: Let's go watch a movie"
  3. receiver passes "Let's go watch a movie" 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 watch a movie"
  7. receiver hears: "7: Let's go watch a movie"
  8. receiver ignores the packet

- How many bits do we need for the sequence number?
  - this is a "stop-and-wait" protocol for each segment, so the receiver needs to distinguish between: *(1)* the next segment and *(2)* the retransmission of the current segment
  - so, ***one bit*** is sufficient (*0 and 1*)

Using Sequence Numbers: Sender
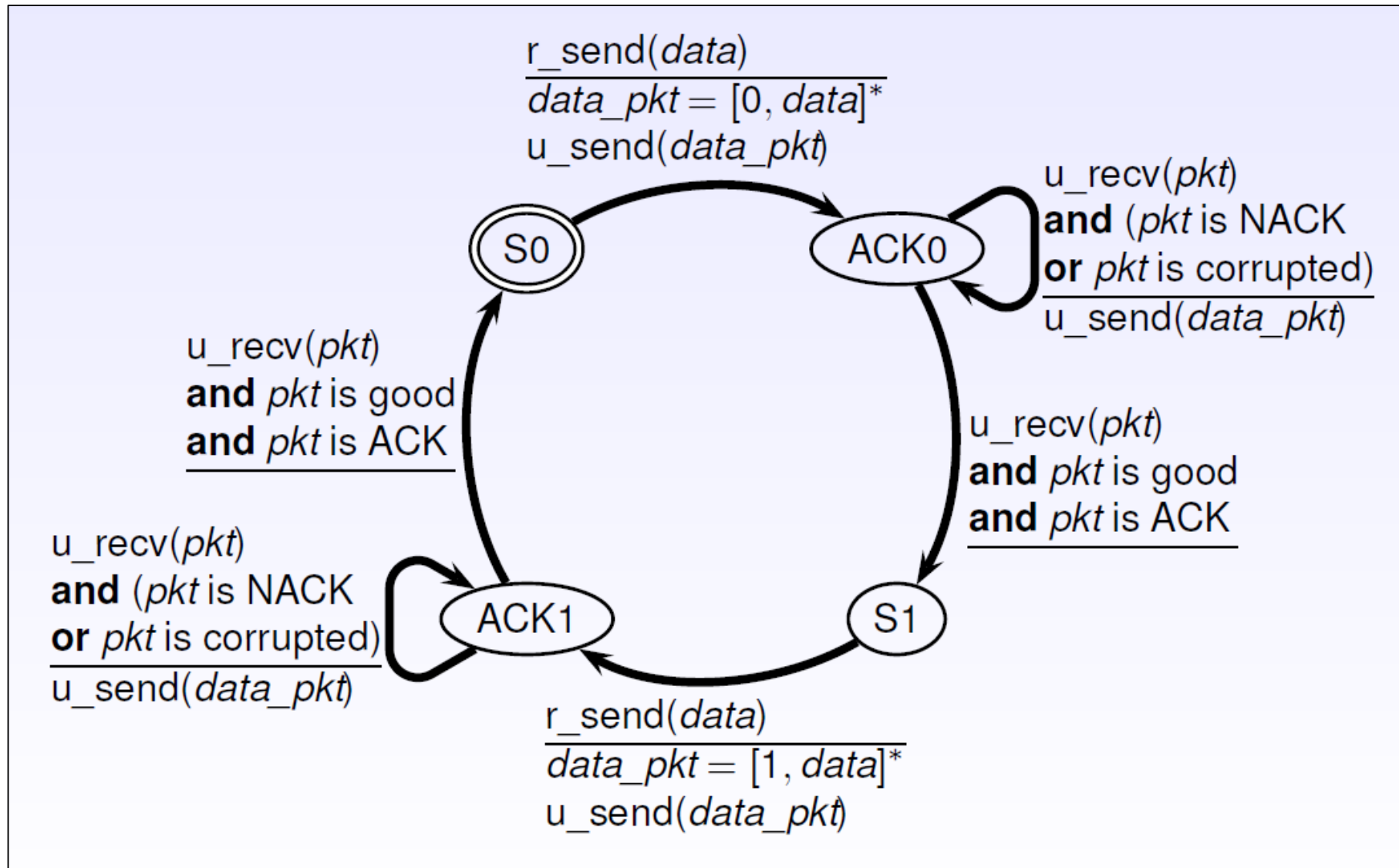
Using Sequence Numbers: Receiver
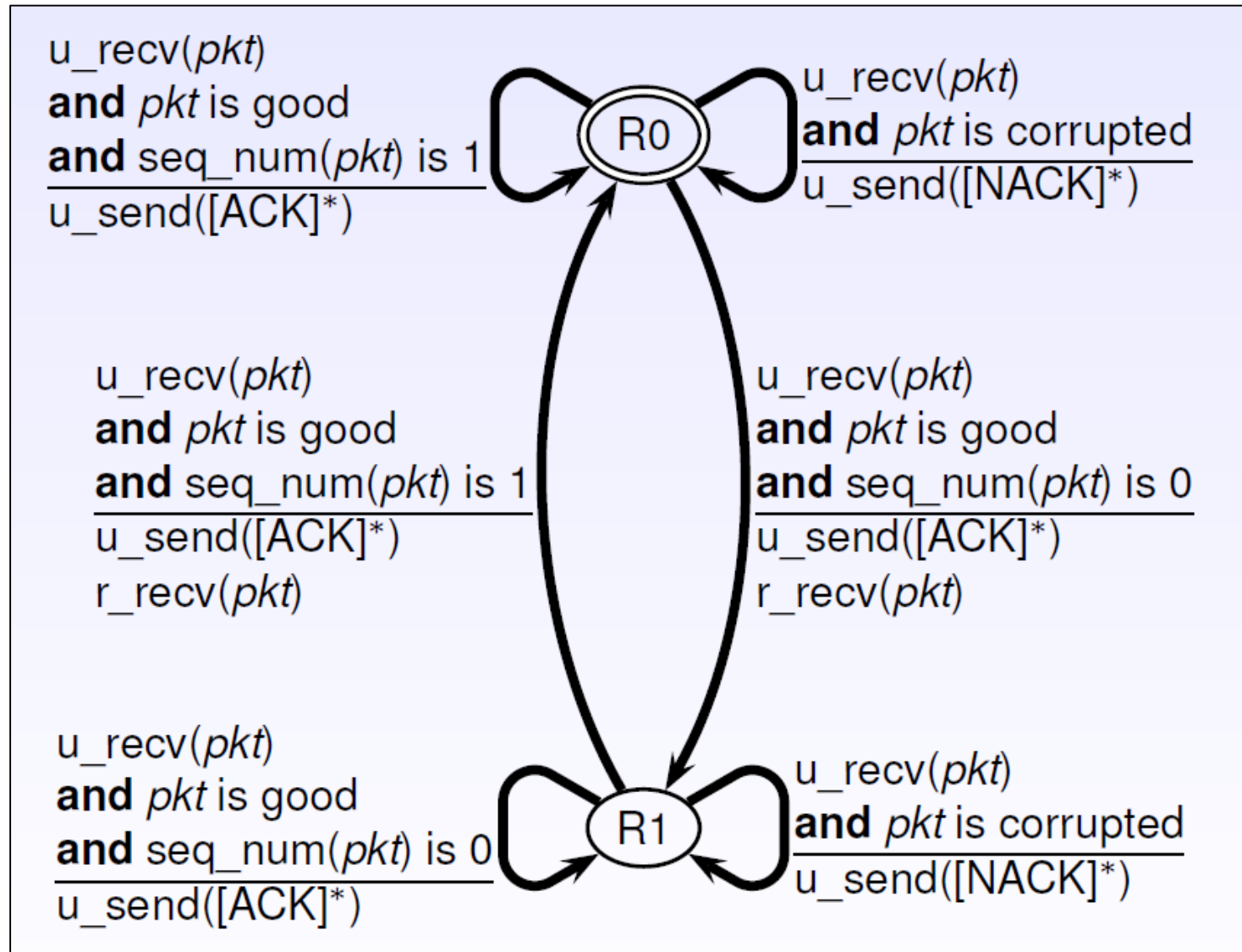
Better Use of ACKs

- But 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 watch a movie"
  2. receiver hears: "7: Let's go watch a movie"
  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
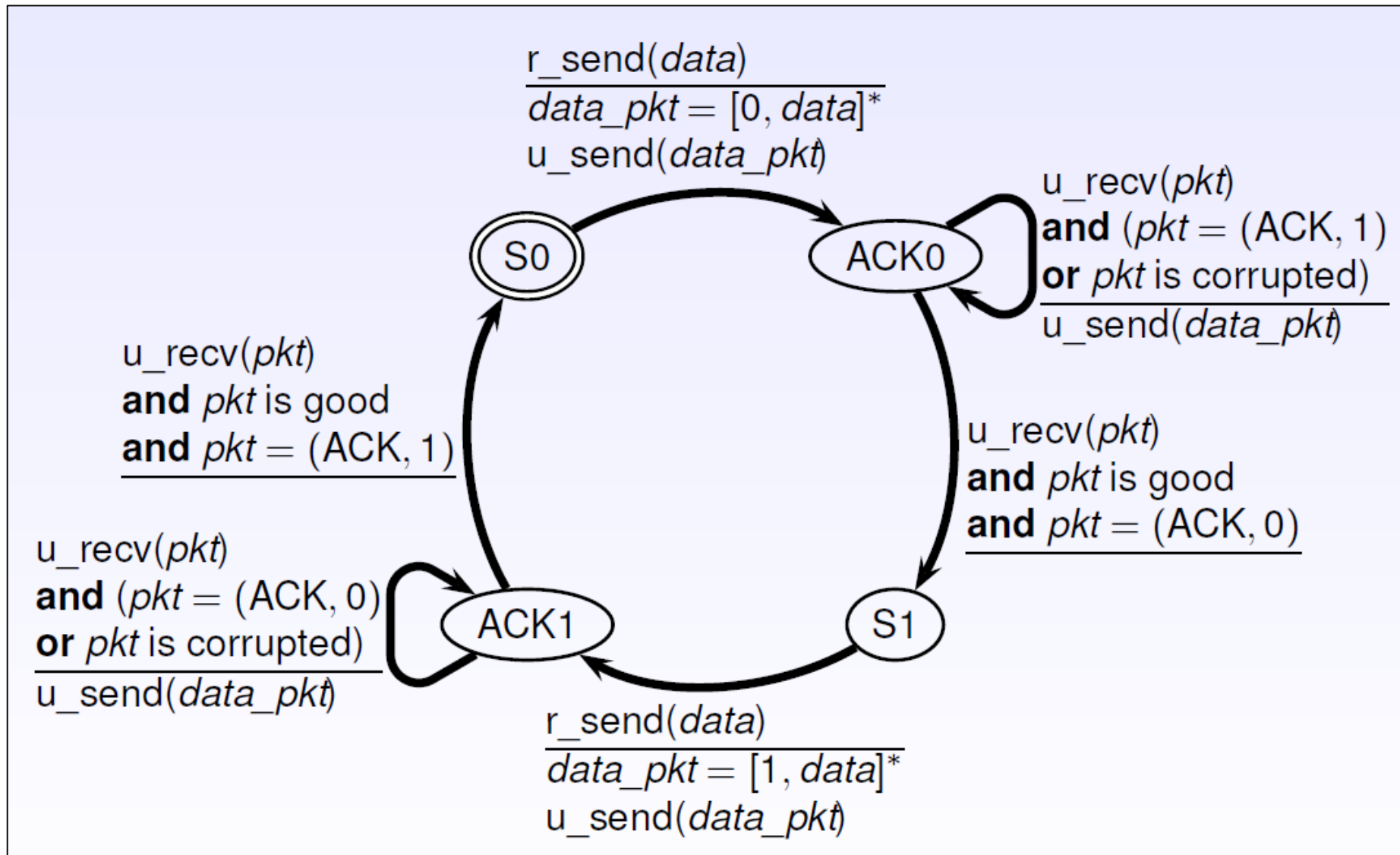
ACK-Only Protocol: Receiver
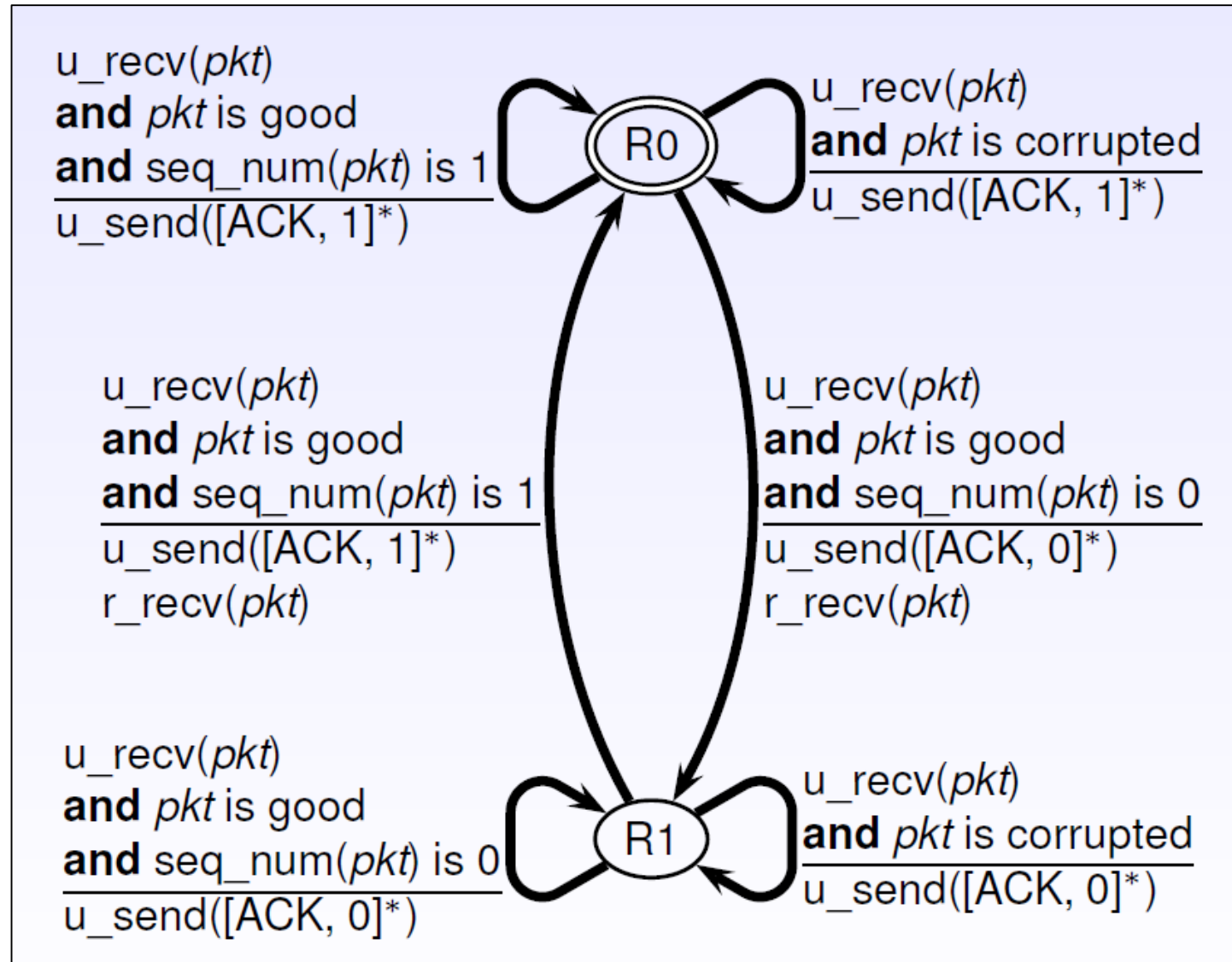
Acknowledgement Generation (Receiver)

- Arrival of in-order segment with expected sequence number;
all data up to expected sequence number already acknowledged
    - Delayed ACK: wait "*X* ms" for another in-order segment (*up to 500ms*);
    if that does not arrive, send ACK (*just for this one*)

- 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
(*i.e. 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 segment starts at the lower end of gap

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
  - However, 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

- 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

- In addition to the alternating bit, we can introduce timeouts

Sender Using Timeouts

$\dfrac{\text{r\_send}(data)}{data\_pkt = [0, data]^*}$
u_send(data_pkt)
start_timer()

$\dfrac{\text{timeout}}{\text{u\_send}(data\_pkt)}$
start_timer()

$\dfrac{\text{u\_recv}(pkt) \text{ and } (pkt = (ACK, 1) \text{ or } pkt \text{ is corrupted})}{\text{u\_send}(data\_pkt)}$
start_timer()

$\dfrac{\text{u\_recv}(pkt) \text{ and } pkt \text{ is good and } pkt = (ACK, 1)}{}$

$\dfrac{\text{u\_recv}(pkt) \text{ and } pkt \text{ is good and } pkt = (ACK, 0)}{}$

$\dfrac{\text{u\_recv}(pkt) \text{ and } (pkt = (ACK, 0) \text{ or } pkt \text{ is corrupted})}{\text{u\_send}(data\_pkt)}$
start_timer()

$\dfrac{\text{r\_send}(data)}{data\_pkt = [1, data]^*}$
u_send(data_pkt)
start_timer()

$\dfrac{\text{timeout}}{\text{u\_send}(data\_pkt)}$
start_timer()

S0   ACK0   ACK1   S1

The alternating bit protocol

a. Operation with no loss

b. Lost packet

Networks and Communications

The alternating bit protocol (cont'd)

c. Lost ACK

d. Premature timeout

Networks and Communications
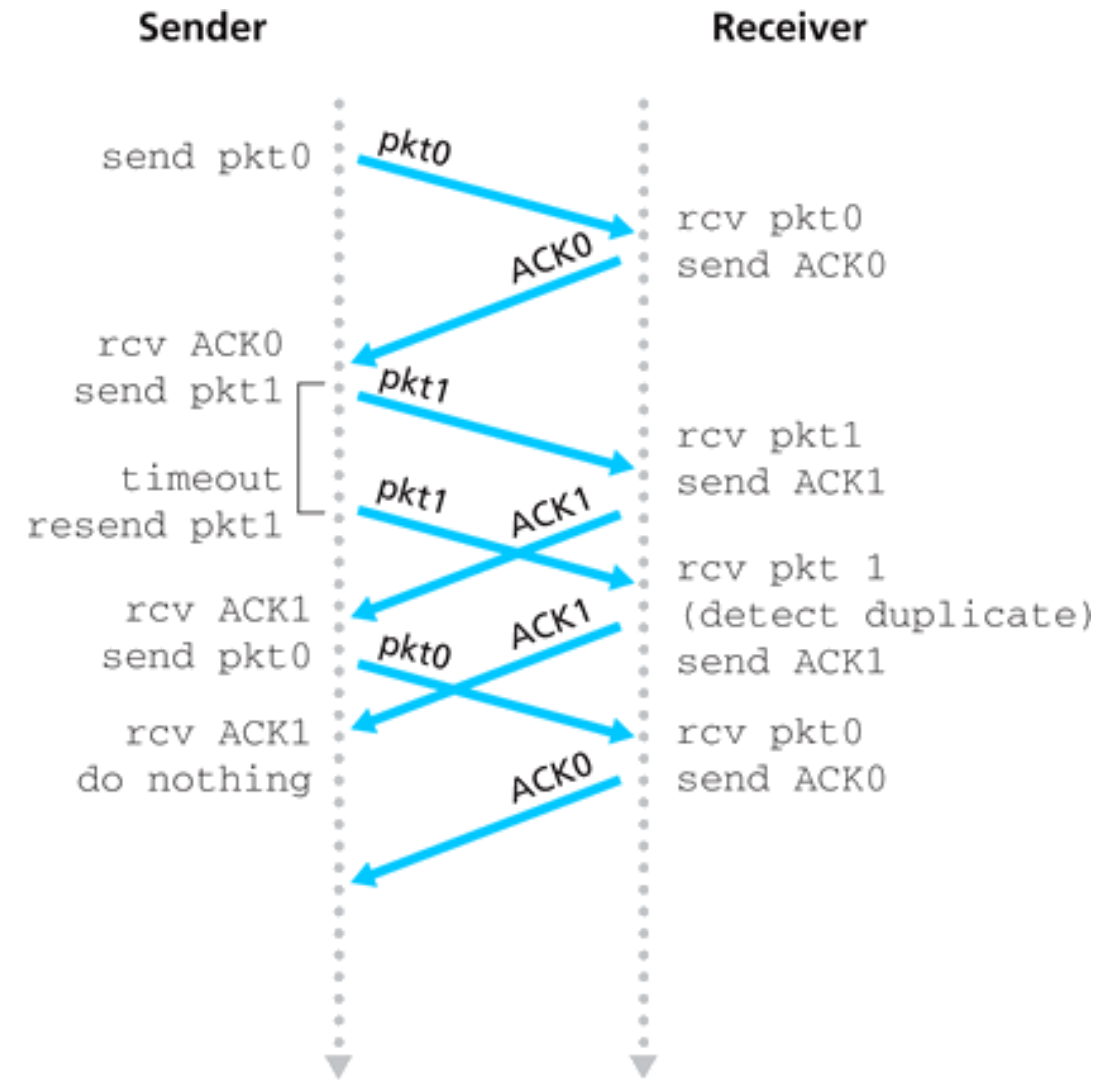
Detecting Congestion

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

- Congestion means that the queue of one or more routers between the sender and the receiver overflow
  - 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 (*equivalent to NACK*)

TCP Congestion Protocols

▪ What needs to change to support each congestion protocol?
- ▪ Depends on the algorithm:

▪ Which one am I using right now?
- ▪ Linux: `cat /proc/sys/net/ipv4/tcp_congestion_control`
  - ▪ You will probably get: cubic
- ▪ Windows: `netsh interface tcp>sh gl`
  - ▪ You will probably get: none (*uses Windows default*)
- ▪ You can force sockets to use a different variant each time

▪ These algorithms combine various characteristics:
- ▪ e.g. Slow Start, Congestion Avoidance,
- ▪ (*Fast Retransmit, Fast Recovery, and more*)
- ▪ Maybe one day you will create a new one!

Algorithm	Affects
TFRC	Receiver, Sender
RED	Router
CLAMP	Router, Receiver
XCP	Router, Receiver, Sender
VCP	Router, Receiver, Sender
MaxNet	Router, Receiver, Sender
JetMax	Router, Receiver, Sender
ECN	Router, Receiver, Sender
Vegas	Sender
High Speed	Sender
BIC	Sender
CUBIC	Sender
H-TCP	Sender
FAST	Sender
Compound TCP	Sender
Westwood	Sender
Jersey	Sender
BBR	Sender

Other TCP Variants/Algorithms

- The TCP protocol described thus far is often referred to as **TCP Reno**

- Another popular implementation is **TCP Vegas**
    - The goal is to detect congestion before losses occur
    - Imminent packet loss is predicted by observing the RTT
    - The longer the RTT of packets, the greater the congestion in the routers

- A problem with TCP Reno is that flows with small RTTs are advantaged, compared to the ones with large RTTs, as their windows can grow faster

- TCP CUBIC (*currently the standard choice in Linux*) solves this problem by making window increase a function of time rather than RTT

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

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

- where

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

- The resulting maximum output rate is roughly

$$\boldsymbol{\lambda} = \frac{\textbf{W}}{\textbf{RTT}}$$

Congestion Control

- How does TCP "modulate" its output rate? (*RFC 2581* & *RFC2001*)

  - Slow Start

  - Congestion Avoidance

  - Additive-Increase and Multiplicative-Decrease

  - Reaction to timeout events

Slow Start

- What is the initial value of W?
    - Initial value of W is MSS => quite low for modern (high-speed) networks

- To quickly get to a good throughput, TCP increases its sending rate exponentially for its first phase (*specifically, W is doubled every RTT*)
    - Increase W by 1 MSS on every (good) acknowledgment
      until W > Slow Start Threshold (*ssthresh*) or until a congestion occurs
    - If W > *ssthresh*, use Congestion Avoidance
    - If W = *ssthresh*, use either

- Initial value of *ssthresh* may be set to an arbitrarily high value, or to the size of the advertised window

- This process is called Slow Start, because of the small initial value of W

Congestion Avoidance

Imperial College London

- TCP increases W linearly in this phase (*W is increased by ~1 MSS every RTT*)

$$\textbf{W} = \textbf{W} + \textbf{MSS} * \frac{\textbf{MSS}}{\textbf{W}}$$

- i.e.

$$\textbf{increment} = \textbf{MSS} * \frac{\textbf{MSS}}{\textbf{W}}$$

- Congestion Avoidance continues until congestion is detected

Sender Behavior
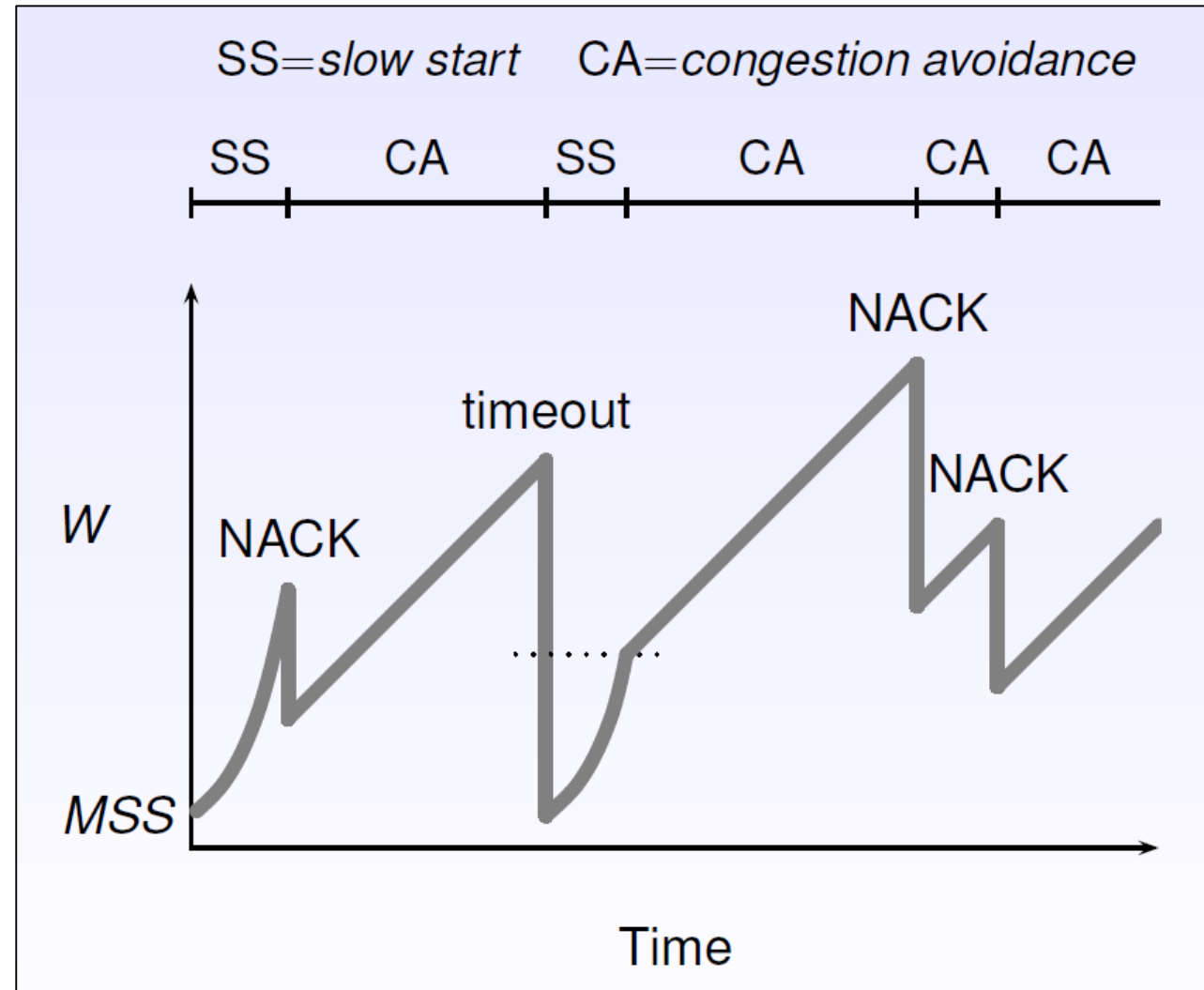
Additive-Increase / Multiplicative-Decrease

- AIMD

- How W is *increased*: at every (good) acknowledgment, increase W in accordance with Congestion Avoidance (MSS^2/W)

    - e.g. suppose W = 14,600 and MSS = 1,460
        - then the sender increases W to 16,060 after 10 acknowledgments

- How W is *reduced*: at every packet 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

Reliability and Timeout

- TCP provides reliable data transfer using a timer to detect lost segments
  - timeout without an ACK => lost packet => retransmission!

- How long to wait for acknowledgments?
  - The timeout interval T must be *larger than* the RTT
    - so as to avoid unnecessary retransmissions

  - 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 $\overline{DevRTT}$, like so:
    $$T = \overline{RTT} + 4 * \overline{DevRTT}$$

- TCP controls its timeout by continuously *estimating* the current RTT

Fast Retransmit

- Agreement: three duplicate ACKs are interpreted as a NACK (**Fast Retransmit**)

- 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

- TCP reacts differently to a *timeout* and to *triple duplicate* ACKs

- *But why **three** duplicate ACKs?*
- Waiting for two packets (rather than only one) is a good trade-off between triggering a Fast Retransmission, when needed, and not retransmitting prematurely in the face of packet reordering

Fast Recovery

- Assuming the current window size is $W = \overline{\overline{W}}$
  - Timeout
    - go back to $W = MSS$
    - run slow start until W reaches $\overline{\overline{W}}/2$ (=ssthresh)
    - then proceed with *Congestion Avoidance*

  - NACK
    - cut W in half: $W = \overline{\overline{W}}/2$ (=ssthresh)
    - run congestion avoidance, ramping up W linearly
   - this is called **Fast Recovery**

Wireless TCP

- *Problem*: TCP assumes that IP is running across wires
  - when packets are lost, TCP assumes this is caused by congestion and slows down
  - in wireless environments, packets usually get lost due channel reliability issues
  - in those cases, TCP should do the opposite: try harder!!

- *Solution #1*: Split TCP connections to distinguish between wired/wireless IP:
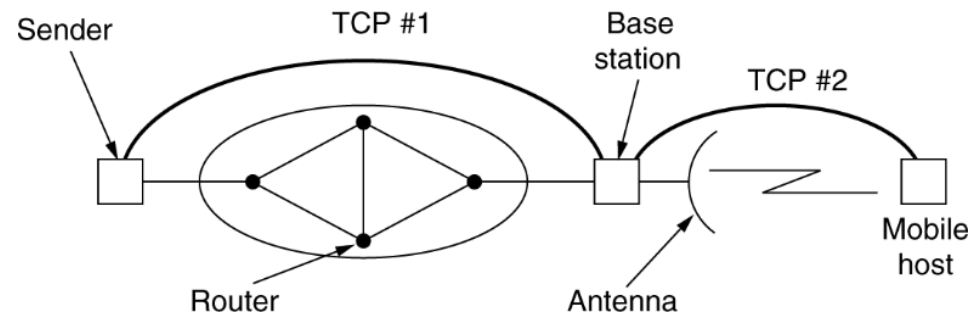
- *Solution #2*: Let the base station do at least some retransmissions, but without informing the source
  - effectively, the base station makes an attempt to improve the reliability of IP over wireless, by using TCP

Flow Control vs. Congestion Control

Imperial College London

- **Congestion Control** aims not to overflow the network
- **Flow Control** aims not to overflow the receiver

- The receiver along with the acknowledgment sends the maximum number of bytes that may be sent (*ReceiverWindow*)

- A window size of **0** is legal
  - it means that no more messages can be sent...
  - ...apart from a 1-byte segment to *ping* the receivers (*no deadlocks*)
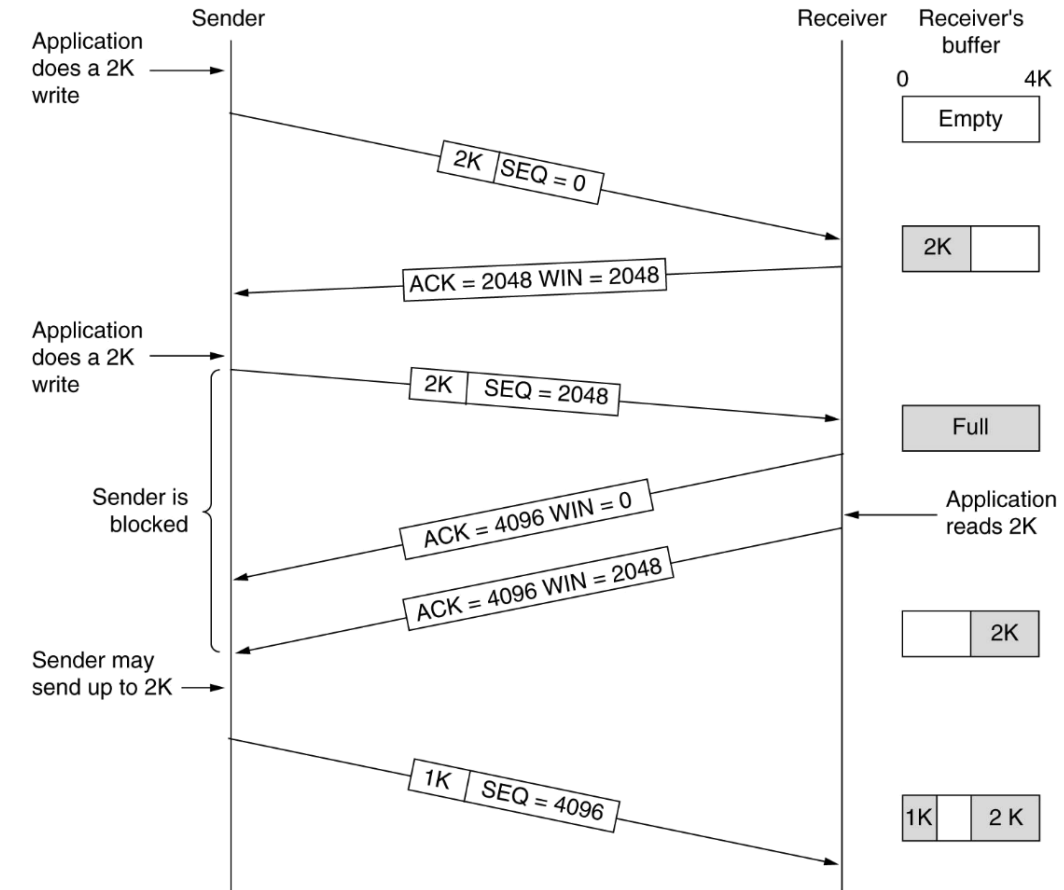
Go-Back-N

- 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
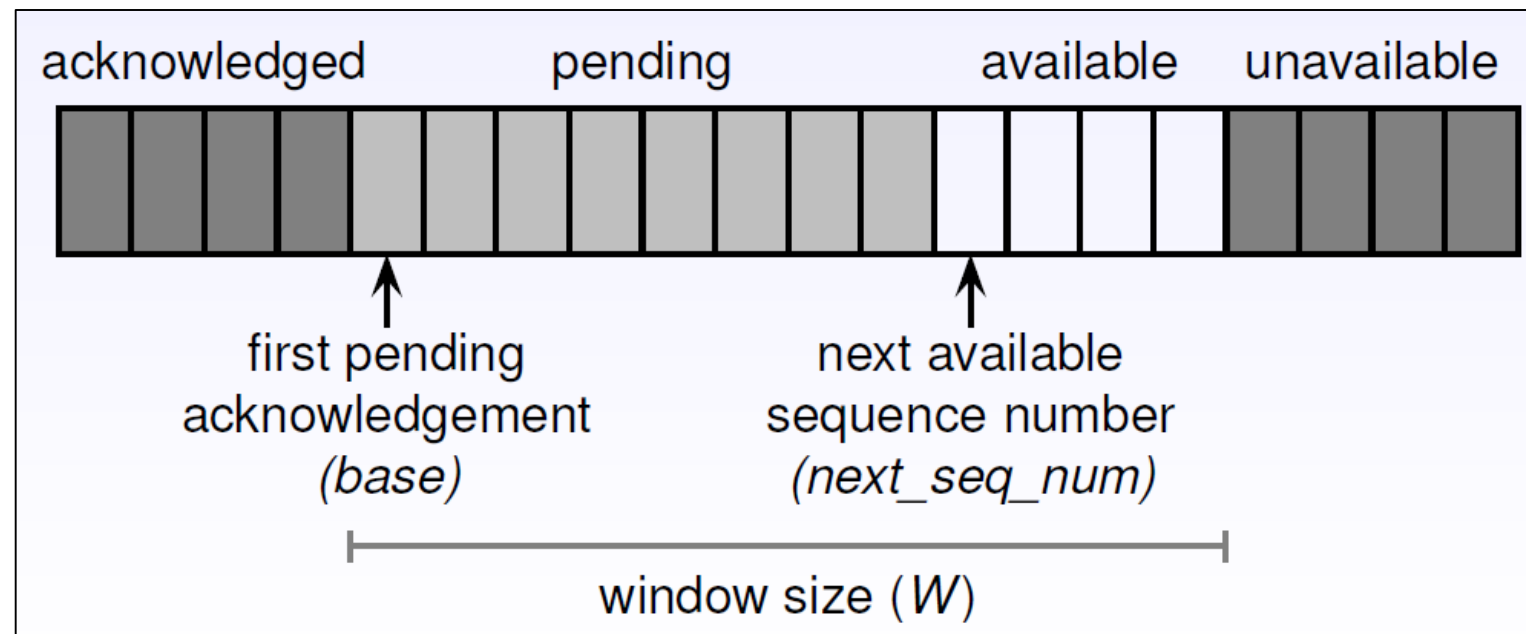
Selective Repeat

- Have the sender retransmit only those packets that it *suspects* were lost or corrupted

- Example:
  - sender maintains a vector of acknowledgement flags
  - receiver maintains a vector of acknowledged flags
  - in fact, receiver maintains a buffer of out-of-order packets
  - sender maintains a timer for each pending packet
  - sender resends a packet when its timer expires
  - sender slides the window when the lowest pending sequence number is acknowledged

Network Usage
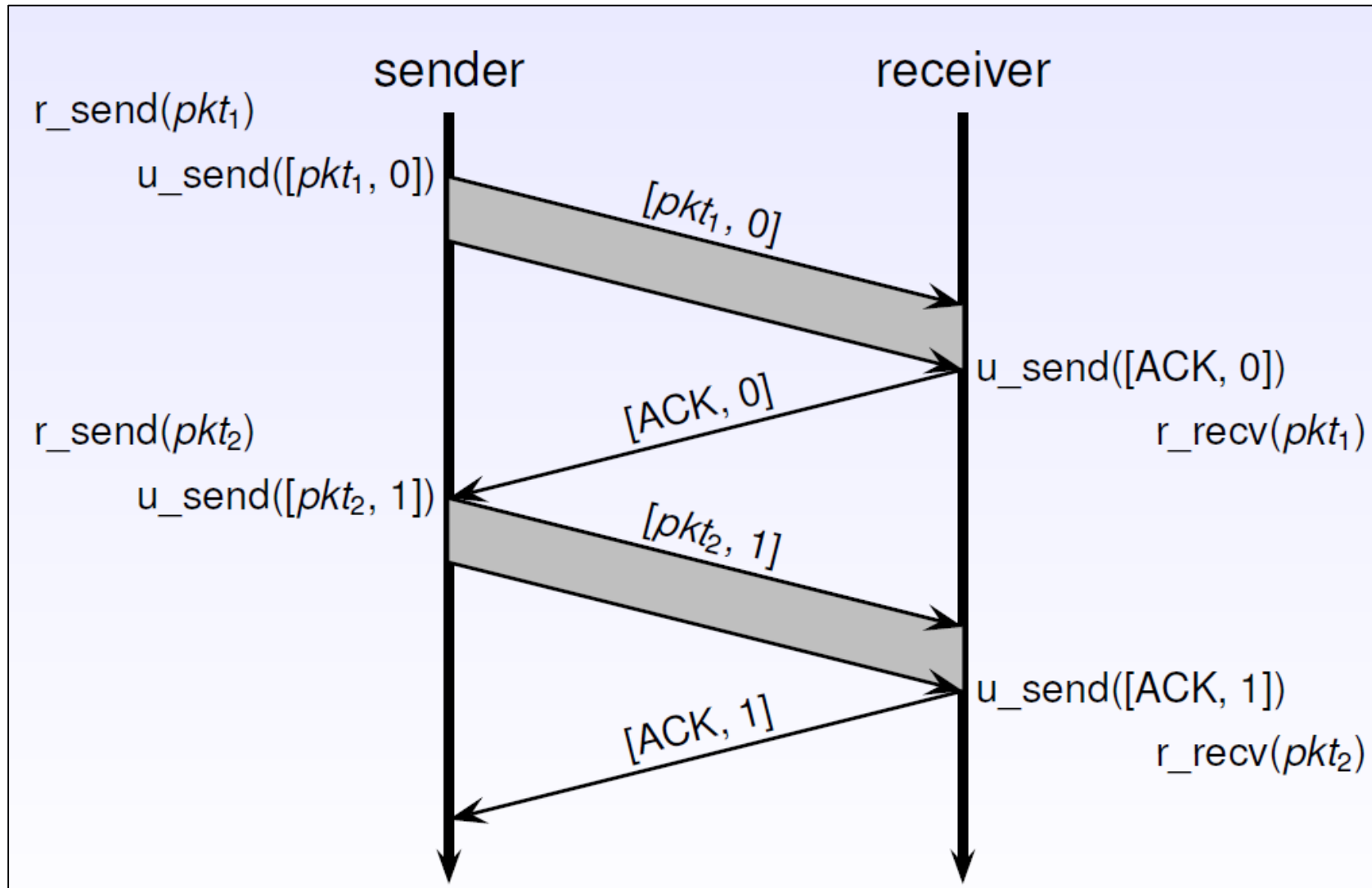
Network Usage (cont'd)

- Utilisation Factor:

$$\frac{how\ much\ we\ actually\ used\ the\ network}{how\ much\ we\ could\ have\ used\ it}$$

- e.g. we are renting a 24Mbps line, but we are currently only downloading at 2MBps

- 2MBps * 8 = 16 Mbps

- $\frac{16Mbps}{24Mbps} \approx 0.67 = 67\%$ utilisation

Network Usage: Example

- Consider an idealised case of two hosts, one located on the West Coast of the US and one on the East Coast

- The Round Trip Time (RTT) is approximately 30ms

- Suppose the channel has transmission rate R = 1 Gbps

- With a packet size L = 1,000 Bytes (8,000 bits), the time to transmit a packet is:
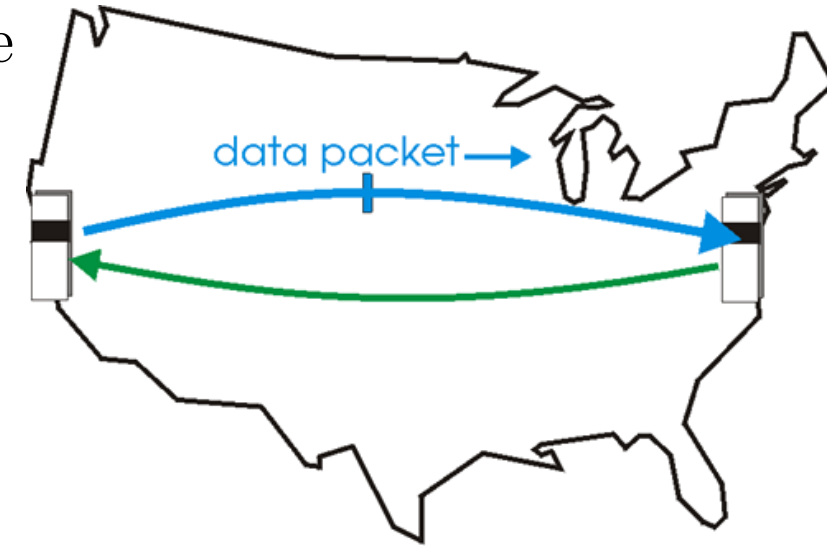
$$\mathbf{d_{trans} = \frac{L}{R} = \frac{8000 \ bits/packet}{10^9 \ bits/sec} = 8 \ \mu s \ (microseconds)}$$

- The *utilisation* of the network is:

$$\mathbf{U = \frac{L/R}{RTT + L/R} = \frac{0.008}{30,000 + 0.008} \approx 0.00027 \approx 0.027\%}$$

Network Simulators

- Cisco Packet Tracer
  - (*Lightweight*) Network Simulator
  - Netacad website
    - Sign Up (Linux/Windows/Mobile)
    - Register for the free 1-hour course
    - Download Packet Tracer
    - Execute and login
  - Used by the CCNA programme

- Freeware alternative: GNS3
  - Very active community

- Paid alternative: OPnet (*now Riverbed*)
  - Multinational clientele
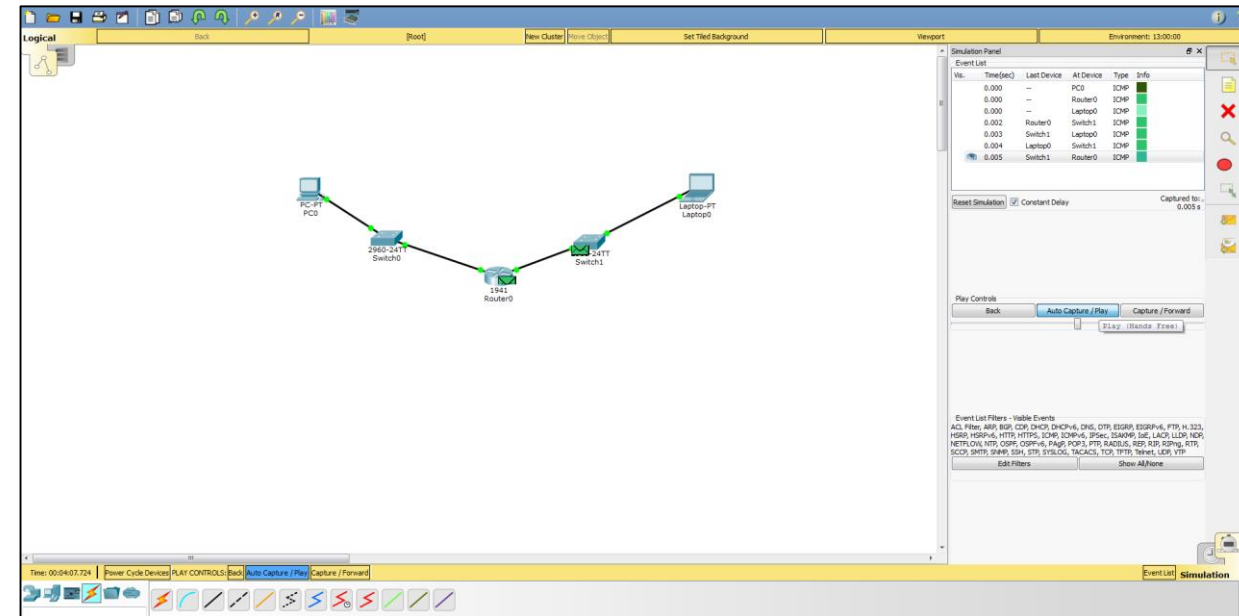
Other useful tools

- netcat (nc)
  - utility which reads and writes data across network connections using TCP or UDP

- Wireshark
  - easy-to-use packet analyser/sniffer
    - *you will see this in action soon*

- Nmap
  - Network mapper and security scanner

- Windows Sysinternals
  - Networking tools for Windows users

- Countless more networking tools/utilities exist
  - always test them on your private network

Q&A

- You will find the **assessed coursework** on the course website(s) on Monday (06/02)
  - as well as the third **exercise worksheet**
    - *worksheet solutions will be uploaded on the DoC website at the end of next week*

- **Suggested reading**: Tanenbaum#6; Peterson#5; Kurose#3.

- Please provide *anonymous* feedback on www.menti.com using the code **40 02 50**
  - *always active throughout the term*

- You can also provide *eponymous* feedback or ask questions via email (*username: **kgk***)

- Thank you for your attention

- **Movie of the week**: Johnny Mnemonic

- **Next time**: Are you safe? Are you really?

Acknowledgements

- *Many thanks to:*
    - *Anandha Gopalan*
    - *Paolo Costa*
    - *Peter McBrien*

- *Special thanks to:*
    - *Maarten van Steen*
    - *Antonio Carzaniga*