Imperial College London

# Computer Networks and Distributed Systems

## Part 5 – Transport Layer

Course 527 – Spring Term 2015-2016

**Emil C Lupu & Daniele Sgandurra**

e.c.lupu@imperial.ac.uk, d.sgandurra@imperial.ac.uk

# Part 5 – Contents

Transport Layer
- End-to-end communications
- End-to-end addressing: ports

Transport protocols
- **UDP**
  - Header format
- **TCP**
  - Header format
  - Connection setup
  - Retransmission
  - Flow / congestion control
  - …



# Host-to-Host Communications

Datagrams transferred between hosts
- Network / Internet level in stack

Routed through networks based on IP addresses
- Source address of sending machine
- Destination address of recipient machine

But:
- No identification of which applications send or receive
- Only unreliable connection-less datagram service



# App-to-App Communications

Identify application instances (processes)?
- Need well-understood identification
- Need to handle process restart
- Need to handle pools of processes serving clients
  - e.g. multi-threaded web server

Destination given by function or service
- One process may handle multiple services

# Protocol Ports

Use **ports** to define end points

- Abstract addressing
- 16 bit unsigned integer: 0 - 65535

Processes specify ports to send to or receive from

- Use operating system calls to **bind** to port
- Packets have source and destination ports



# Well Known Service Ports

Other systems know which port to address to

- Standard services usually run on well known ports

List of well known services (RFC 1060)

Ports 0 – 255: Internet Assigned Numbers Authority (IANA)

Ports 0 – 1023: Privileged UNIX standard services

Static file with service/port mappings

Unix: **`/etc/services`**

Windows: **`\windows\system32\drivers\etc\services`**



# /etc/services

```
tcpmux        1/tcp     # TCP port service multiplexer
echo          7/tcp
echo          7/udp
discard       9/tcp  sink null
discard       9/udp  sink null
systat        11/tcp users
daytime       13/tcp
daytime       13/udp
netstat       15/tcp
qotd          17/tcp quote
msp           18/tcp            # message send protocol
msp           18/udp
chargen       19/tcp ttytst source
chargen       19/udp ttytst source
ftp-data      20/tcp
ftp           21/tcp
fsp           21/udp fspd
ssh           22/tcp        # SSH Remote Login Protocol
ssh           22/udp
telnet        23/tcp
smtp          25/tcp mail
```



# Less Well Known Addresses

Ports need not all be "well known"

Source port can be generated by application

- Useful when just needed for interaction

Destination port need not be well known

- Pass information to sender for novel application

Portmapper service (e.g., for Remote Procedure Calls)

- Process register (name, port) binding
- Client can query portmapper service by name.
- Portmapper (port 111)
- See remote procedure calls later on in the course

# UDP/TCP Services

Two main transport layer protocols: **UDP** and **TCP**

Some services use UDP:

- **`bootp`** (on Windows), **`tftp`**, **`snmp`**, …

Some services use TCP:

- **`smtp`**, **`ftp`**, **`telnet`**, **`finger`**, **`http`**, …

Some services use either:

- **`echo`**, **`dns`**, **`ssh`**, **`bootp`** (on Linux), **`irc`**, …



# Complete Addresses

The complete addressing for a datagram (UDP/TCP) describes the sender and receiver in such a way that the services which are communicating are fully defined:

Source IP address and source port

- Source port = return addr (may be omitted in UDP)
- e.g. **`columbia.doc.ic.ac.uk:57992`**

Destination IP address and destination port

- e.g. **`www.amazon.co.uk:80`**



# Sockets

**Socket:** Abstract communications endpoint

- **Stream socket** (SOCK_STREAM) which uses TCP
- **Datagram socket** (SOCK_DGRAM) which uses UDP

**Active socket**

- Connected to remote active socket via open connection
- Closing connection closes sockets at both ends

**Passive socket**

- Unconnected socket awaiting incoming connection
- Active socket spawned to handle connection
  - Passive socket goes back to waiting for new connections

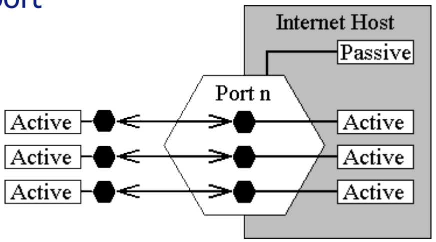


# Sockets and Ports

A socket is not a port (but they are related)

1+ sockets associated with 1 port

- At most one passive socket
  - Awaiting new connections
- Multiple active sockets
  - Handling open connections

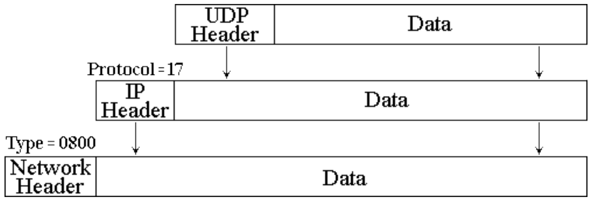



Hence 4-part addressing

i.e. (src IP, src port, dst IP, dst port)

- Many connections use same port
- Need to know where connection is coming from

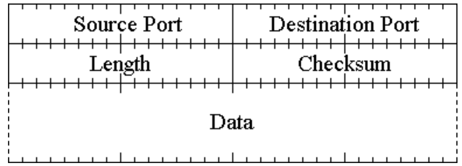
# User Datagram Protocol (UDP)




UDP provides plain, IP-like service
- Connection-less datagrams, unreliable delivery, no sequence control, possible duplication
- Good for fast transfer with resilience to packet loss



# UDP Header Format




Source port
- Optional, 0 if not used, reply-to port if used

Destination port
- Host addressing still provided by IP headers

Length (in bytes)
- Includes header and data (min. 8 for no data)

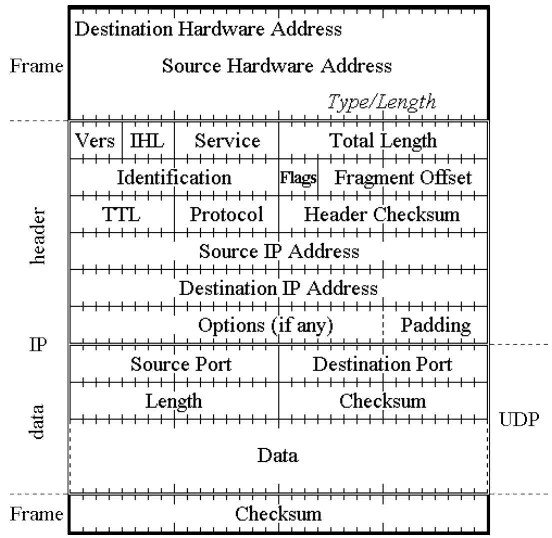


# UDP Checksum

Checksum (16 bit using 1s complement sum)
- Calculated over pseudo header + UDP header + data
- **Pseudo header** mimics IP header
  - Source IP address
  - Destination IP address
  - Protocol (17)
  - UDP length
    - Zeros to pad to multiple of two octets
- Allows detection of changed IP headers by gateways without actually duplicating data



# UDP/IP Packet in Ethernet Frame



# Reliable Service?

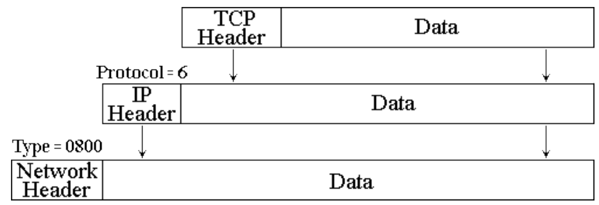
UDP is <u>unreliable</u>

Features of reliable a service
- Unstructured stream abstraction
- Virtual circuit connection
- Full-duplex connection

Could build reliable service over UDP
- Needs to keep track of successfully transmitted packets
- Add error-correcting & retransmission mechanisms



# Transmission Control Protocol (TCP)




TCP adds a lot to IP:
- Streams with reliable delivery
- Full-duplex operation
- Flow control
- Network adaptation for congestion control
- Complexity & overheads



# Connections

TCP uses **connection** as its basic abstraction
- Rather than just protocol port (receiving datagrams)

Connection-oriented service on top of connectionless IP
- Connection identified by pair of endpoints

**Endpoint** is (host, port) tuple: (IP addr:port)

e.g. src: **`146.169.7.41:1069`**
dst: **`140.247.60.24:25`**
Mail application on **`146.169.7.41`** connects to SMTP port on **`140.247.60.24`**



# TCP Features

**Streams**
- TCP data is stream of bytes
- Underlying datagrams concealed

**Sequence numbers** for reliable delivery
- Used to maintain byte order in stream
- TCP detects lost data and arranges retransmission
- Stream delayed during retransmission to maintain byte sequence

**Flow control**

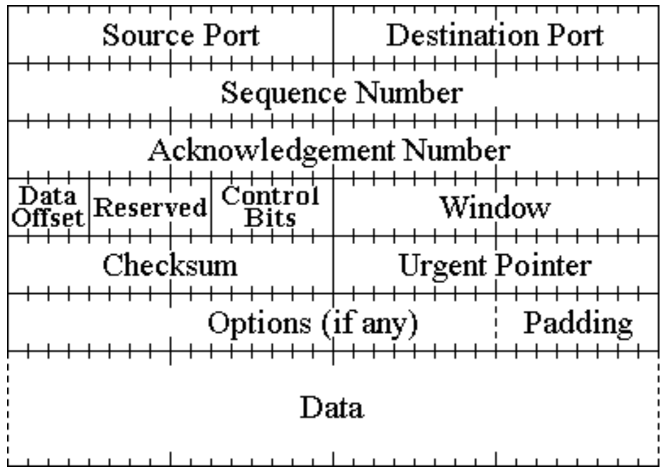
- Manages data buffers and coordinates traffic to prevent overflows
- Fast senders have to pause for slow receivers to keep up

**Congestion control**

- Monitors and learns delay characteristics of network
- Adjusts operation to maximise throughput without overloading network



# TCP Segment Format






# TCP Fields: Sequence Numbers

Sequence number (seq num)

- Indicates position in stream of 1$^{st}$ data byte in segment

Acknowledgement number (ack num)

- Exploit full-duplex connection and use segments to piggyback acknowledgments
- Ack num = next seq num sender expects to receive



# TCP Fields: Data Sizes

Data offset: Number of 32-bit words in TCP header

- Needed because of variable length options

Window size (used for flow control)

- Num of data bytes which may be sent, starting with byte indicated by ack num
- Recipient should not send more than (window size – bytes sent) without acks (in transit)
- 0 means “no more data now, please”

# TCP Fields: Control + Checksum

Control bits

**URG**: urgent pointer valid

**ACK**: ack num valid

**PSH**: "push" this segment (transmit promptly)

**RST**: reset connection

**SYN**: synchronise sequence numbers

**FIN**: sender has reached end of byte stream

Urgent pointer

- Pointer to high priority data in stream (e.g. error conditions)

Options

- Negotiate max segment size, scaling factor for window size, …

Checksum (16 bit using 1s complement sum)

- Same as for UDP with pseudo header



# Passive and Active Opens

Both endpoints cooperate to open TCP connection

**Passive open**

- One end waits for incoming requests (server)

**Active open**

- Other end initiates communication (client)



# Connection Control

Two hosts must synchronise seq nums

- Controls packet order and detects loss + duplicates

Use **SYN** segments to establish connection

- Establish initial sequence num (**ISN**)
- Stream positions are offsets from ISN
  - 1st data byte in segment = ISN + 1

ISN chosen randomly

- Need to be unique over life-time of connection
- Starting at 0 bad idea because of old packets…

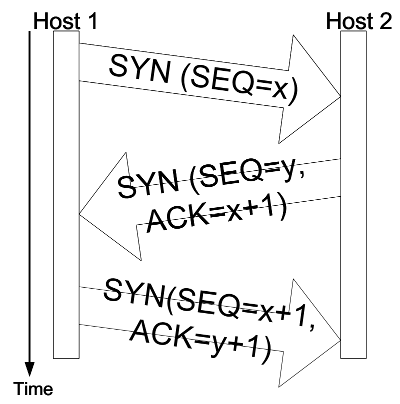


# Connection Establishment

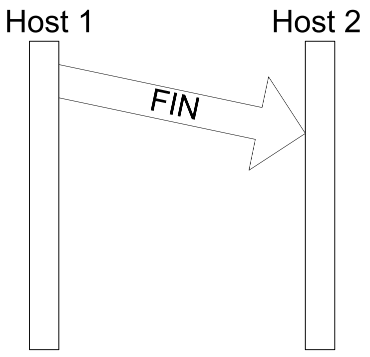
**Three way handshake**

- Establishes connection
- Sender and receiver agree on seq nums
- Works when two hosts establish connection concurrently

What happens when an old duplicate of the first message arrives?



# Asymmetric Connection Release




Unilateral close of connection

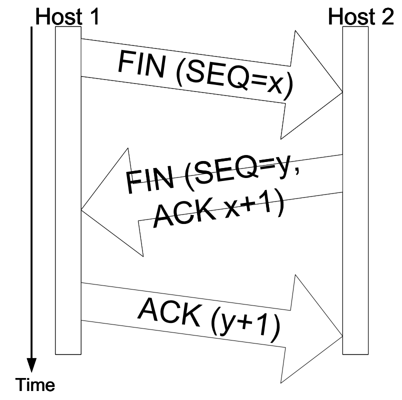
May loose data being transmitted



# Symmetric Connection Release

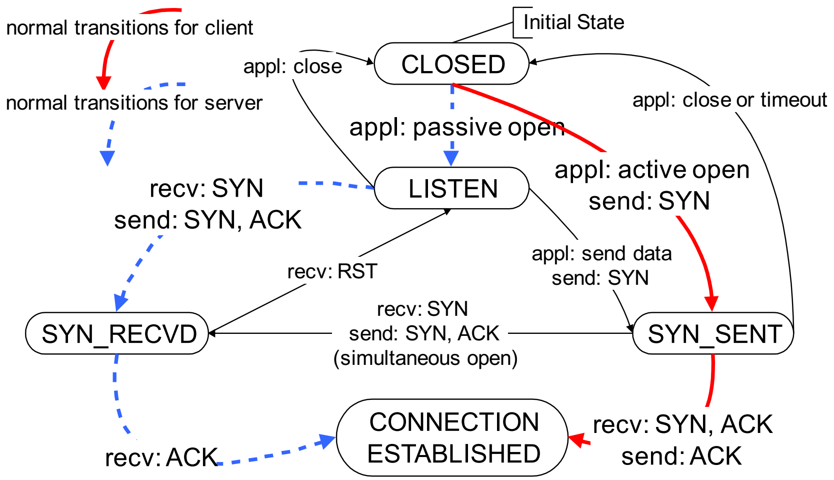
Treat connection as two unidirectional connections to be released

- Hosts agree on end seq nums
- Timeouts to handle lost messages





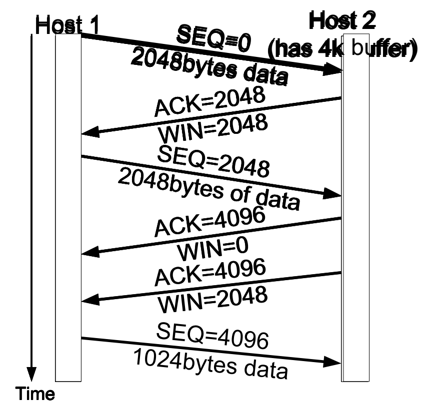

# TCP State Chart






# TCP Window Management

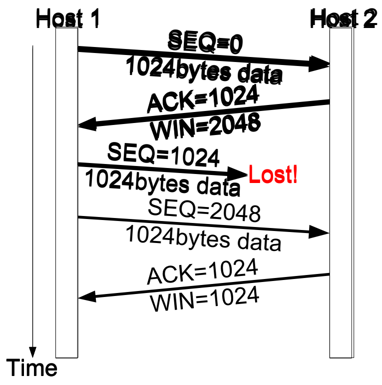
- Host 2 has 4k buffer
- Sent data controlled by WINdow field
- Sender does not have to fill receiver's buffer with each segment
- SEQuence and ACKnowledgement indicate what has been sent/received



# Loss & Duplicate ACKs

**Duplicate ACK**

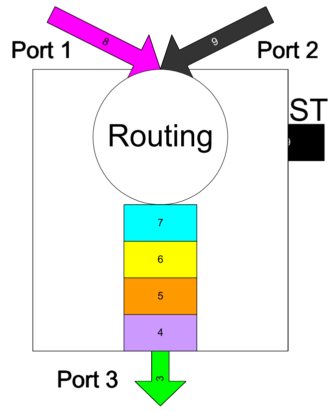
- Generated when out-of-order segment received
- Notifies sender of duplicate and expected seq num
- Sender resends data if ACK takes longer than timeout or several duplicates received






# Congestion

Congestion occurs in <u>routers</u> and in <u>medium access</u>




- Multiple incoming links can saturate single outgoing link
- Slower outgoing link can be saturated by one incoming link

Routers use store-forward

- Process each packet before sending
- If buffer becomes full, drops packets



# TCP Congestion Control

Packet loss mostly due to congestion and not error

- Detect congestion by considering packet loss
- Change transmission rate to adapt to congestion

TCP sender maintains <u>2 windows</u>

- **Receiver window** (flow control) and **congestion window**
- Uses whichever currently smaller



# TCP Congestion Window

**Congestion window** based on network conditions

- Windows <u>grows</u> and <u>shrinks</u> based on packet loss
- Different algorithms for finding optimal size
  e.g. additive increase, multiplicative decrease

Requires efficient timeouts to detect loss

- TCP measures RTT and adjusts timeouts

Lots of complexity to ensure throughput, fairness, …

# Rough Behaviour

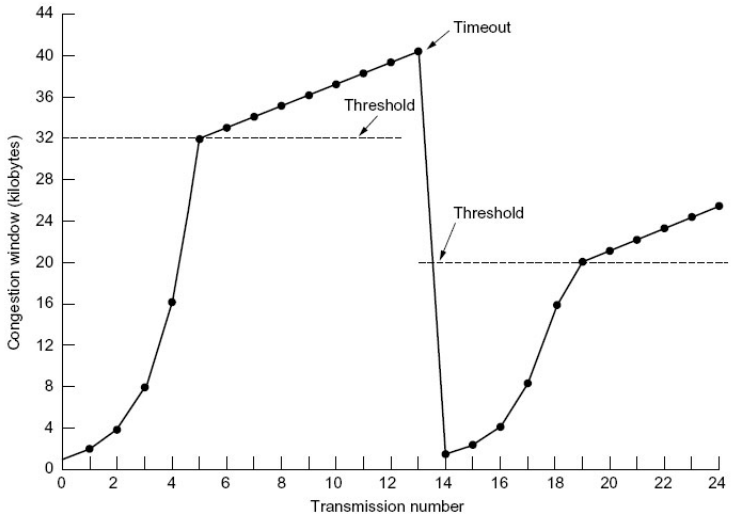
– Start with small window size.

– For each data segment acknowledged before time out increase window size by another segment until threshold.

– Increase linearly afterwards.

– For each packet lost reduce threshold by half.








# Summary: UDP or TCP?

**UDP**

No need for reliability and error detection

- Message exchanges without transactional behaviour, e.g. DNS, DHCP
- Real-time apps, e.g. sensor monitoring, video streaming

Good for short communications

Efficient for fast networks

**TCP**

Need for reliability, error correction, flow and congestion control, or security

- Terminal sessions, e.g. SSH, Telnet
- Large data transfer, e.g. web, FTP, email

Efficient for long-lived connections

Requires more CPU time and bandwidth than UDP