UNIVERSITY OF Southampton
School of Electronics and Computer Science

Transport Layer 1

ELEC3227/ELEC6255

Alex Weddell
asw@ecs.soton.ac.uk

Photo by Alessandro Venturi on Unsplash

Overview

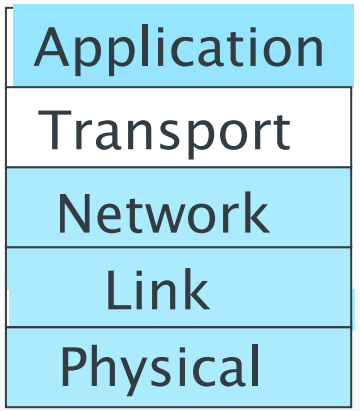
- Services provided by transport layer
- Transport service primitives
- Sockets
- Connection establishment and release
- Error/flow control

UNIVERSITY OF Southampton
School of Electronics and Computer Science

The 5-layer Model

- Transport is **below** the application layer and **above** the network layer
- **Provides services** to the Application layer
  - e.g. TCP/IP, UDP
- **Relies on services** provided by the Network layer
  - Transferring packets across network

- Responsible for delivering data across networks with the desired **reliability** or **quality**

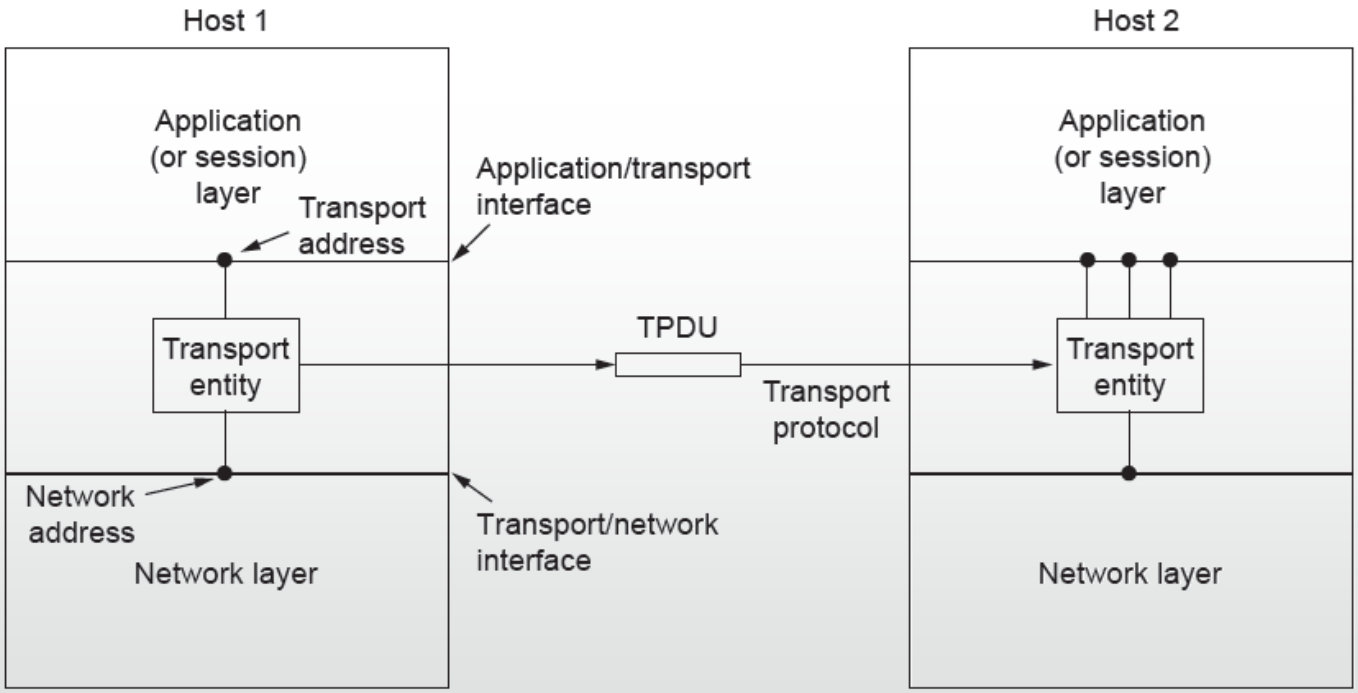


Services Provided by Transport Layer

- Enable data to be transmitted across the network/internet
  - **TCP/IP: Transmission Control Protocol/Internet Protocol**
    Reliable, guaranteed transmission, but significant overheads. Used for file transfer.
  - **UDP: User Datagram Protocol**
    Quick but unreliable protocol, transmissions may not arrive, application must be able to cope with this. Typically used where low latency is more important than accuracy (e.g. video chat, streaming).

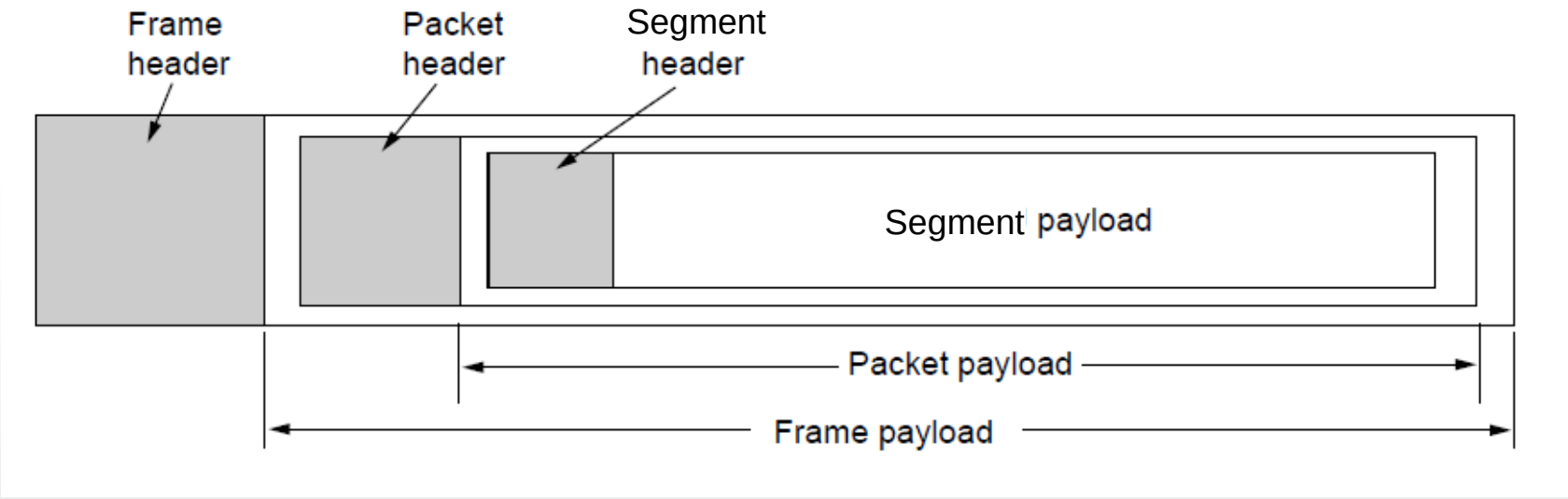
Services Provided to Application Layer

- Transport layer adds reliability to the network layer
  - Offers connectionless (e.g., UDP) and connection-oriented (e.g, TCP) service to applications



Services Provided to Application Layer

- Transport layer sends **segments** in packets (in frames)



Transport Service Primitives

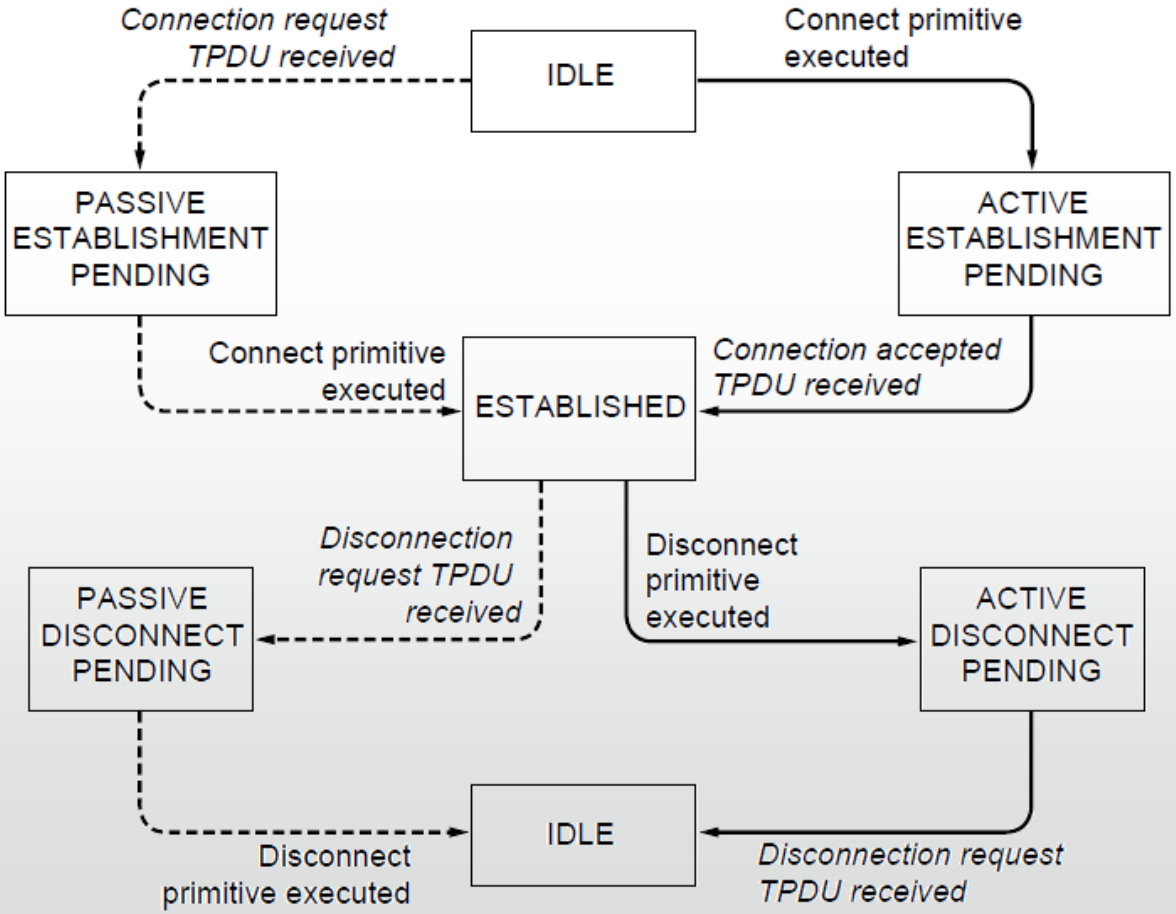
- Primitives that applications might call to transport data for a simple connection-oriented service:
  - Client calls CONNECT, SEND, RECEIVE, DISCONNECT
  - Server calls LISTEN, RECEIVE, SEND, DISCONNECT

Primitive	Segment sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Transport Service Primitives

- State diagram for a simple connection-oriented service



Solid lines (right) show client state sequence

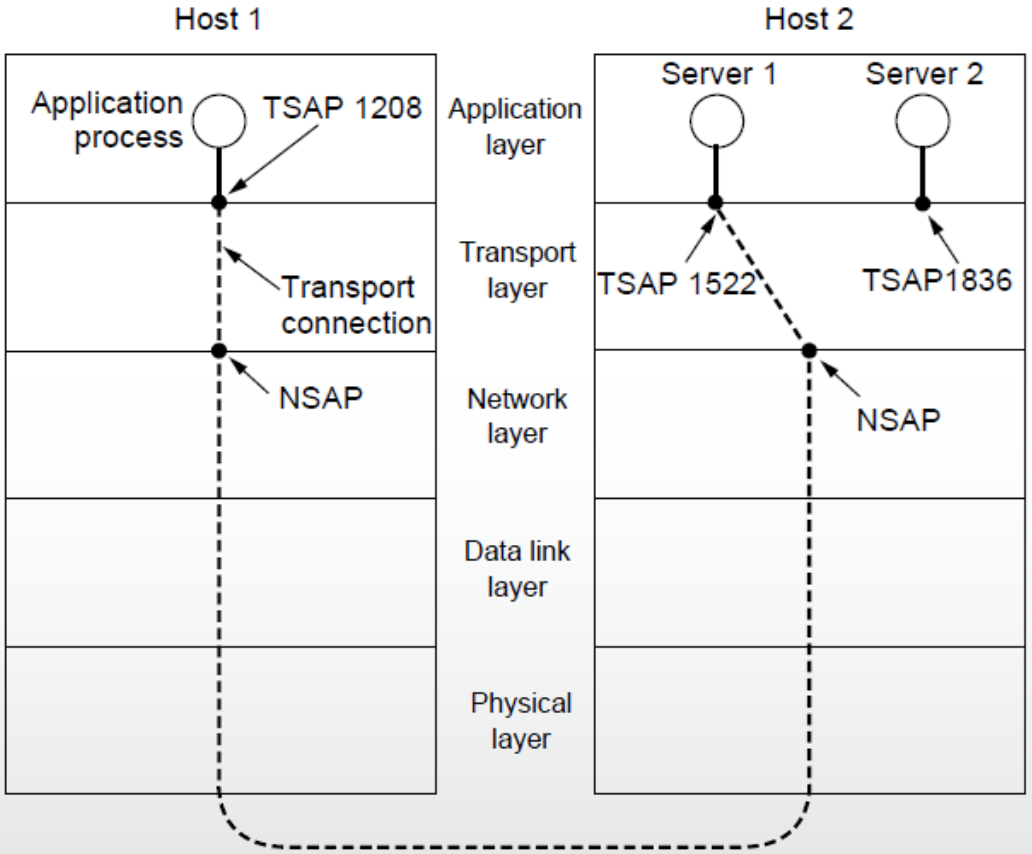
Dashed lines (left) show server state sequence

Transitions in italics are due to segment arrivals.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Addressing

- Transport layer adds TSAPs (Transport Service Access Points)
- Multiple clients and servers can run on a host with a single network (IP) address
- TSAPs are **ports** for TCP/UDP



Sockets

"A **socket** is one endpoint of a two-way communication link between two programs running on the **network**. A **socket** is bound to a port number so that the [**Transport**] layer can identify the application that data is destined to be sent to. **An endpoint is a combination of an IP address and a port number**."

https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

Sockets

- Sockets are also known as 'virtual ports'
- Very widely used primitives started with TCP on UNIX
  - Notion of "sockets" as transport endpoints
  - Like simple set plus SOCKET, BIND, and ACCEPT

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Socket Example – Internet File Server

- Client code 1:

```
. . .

if (argc != 3) fatal("Usage: client server-name file-name");
h = gethostbyname(argv[1]);
if (!h) fatal("gethostbyname failed");

s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s <0) fatal("socket");
memset(&channel, 0, sizeof(channel));
channel.sin_family= AF_INET;
memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
channel.sin_port= htons(SERVER_PORT);

c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
if (c < 0) fatal("connect failed");

. . .
```

Get server's IP address

Make a socket

Try to connect

Socket Example – Internet File Server

- Client code 2:

```
. . .
write(s, argv[2], strlen(argv[2])+1);

while (1) {
    bytes = read(s, buf, BUF_SIZE);
    if (bytes <= 0) exit(0);
    write(1, buf, bytes);
}
}
```

Write data (equivalent to send)

Loop reading (equivalent to receive) until no more data; exit implicitly calls close

Socket Example – Internet File Server

- Server code 1:

```
. . .

memset(&channel, 0, sizeof(channel));
channel.sin_family = AF_INET;
channel.sin_addr.s_addr = htonl(INADDR_ANY);
channel.sin_port = htons(SERVER_PORT);

/* Passive open. Wait for connection. */
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s < 0) fatal("socket failed");
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
if (b < 0) fatal("bind failed");

l = listen(s, QUEUE_SIZE);
if (l < 0) fatal("listen failed");

. . .
```

Make a socket

Assign address

Prepare for incoming connections

Socket Example – Internet File Server

- Server code 2:

```
. . .
while (1) {
    sa = accept(s, 0, 0);
    if (sa < 0) fatal("accept failed");

    read(sa, buf, BUF_SIZE);

    /* Get and return the file. */
    fd = open(buf, O_RDONLY);
    if (fd < 0) fatal("open failed");

    while (1) {
        bytes = read(fd, buf, BUF_SIZE);
        if (bytes <= 0) break;
        write(sa, buf, bytes);
    }
    close(fd);
    close(sa);
  }
}
```

Block waiting for the next connection

Read (receive) request and treat as file name

Write (send) all file data

Done, so close this connection

Connection Establishment

Main aim is to ensure reliability even though packets may be **lost**, **corrupted**, **delayed**, and **duplicated**

- Don't treat an old or duplicate packet as new
- Use ARQ and checksums for loss/corruption

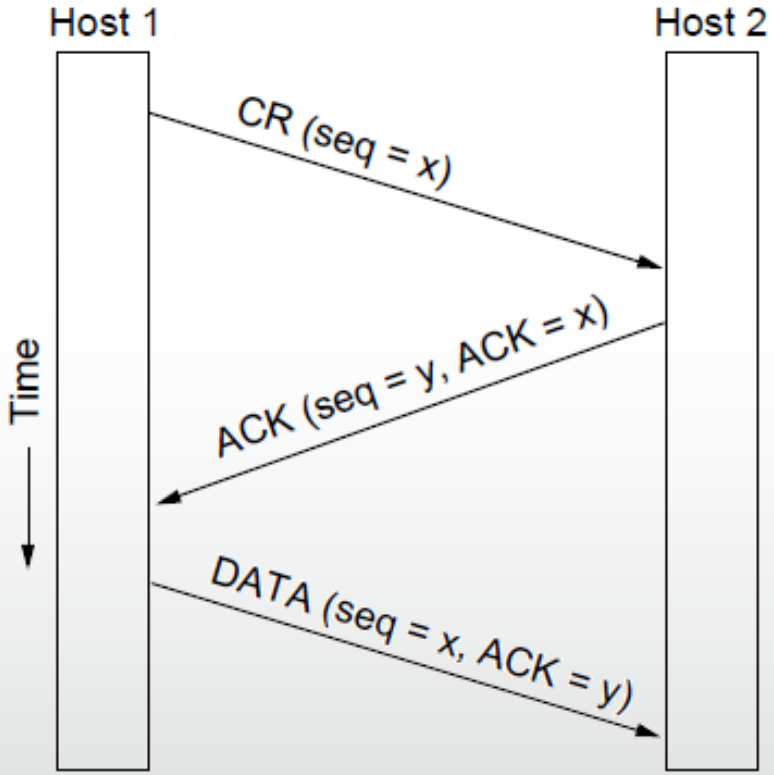
Approach:

- Don't reuse sequence numbers within twice the MSL (Maximum Segment Lifetime) of 2T = 240 secs
- Three-way handshake for establishing connection

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Connection Establishment

- Three-way handshake used for initial packet
  - Since no state from previous connection
  - Both hosts contribute fresh seq. numbers
  - CR = Connect Request



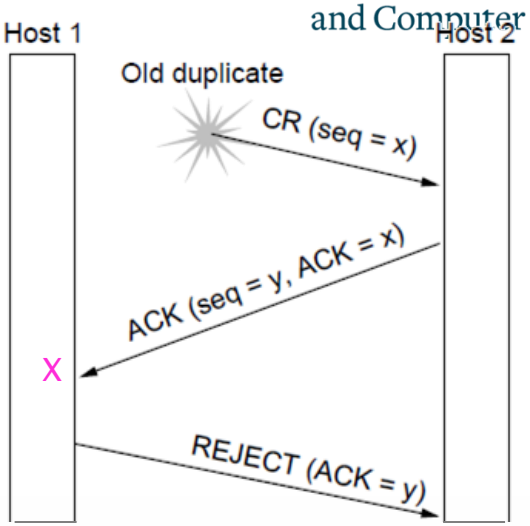
Connection Establishment

- Three-way handshake protects against odd cases:

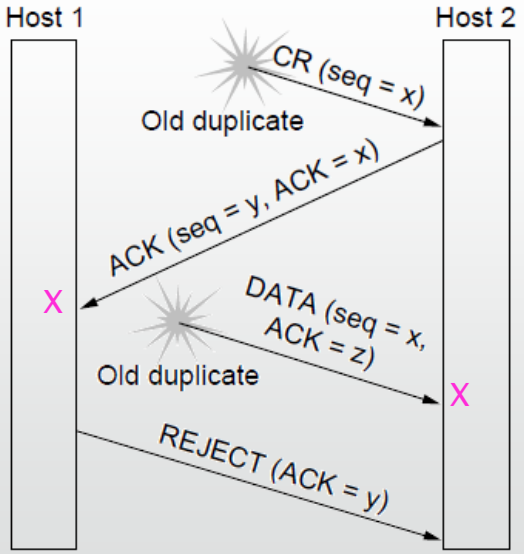
a) Duplicate CR. Spurious ACK does not connect

b) Duplicate CR and DATA. Same plus DATA will be rejected (wrong ACK).

a)



b)

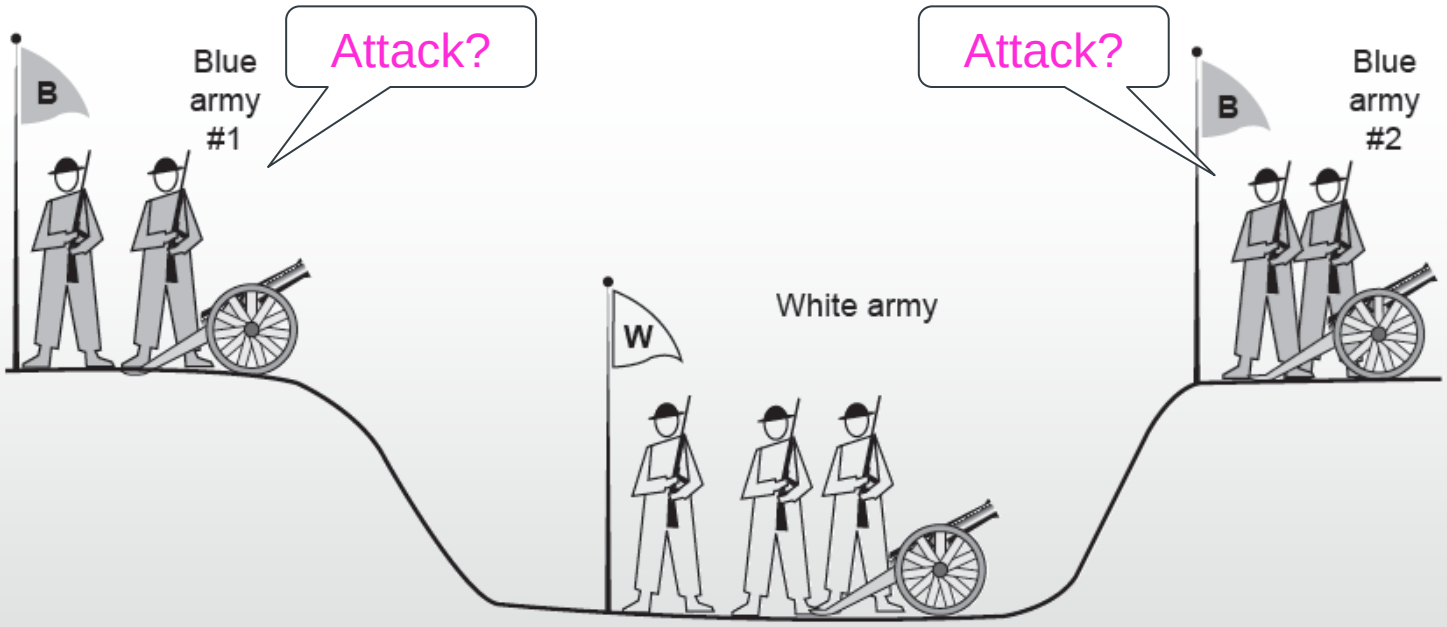


UNIVERSITY OF Southampton
School of Electronics and Computer Science

Connection Release

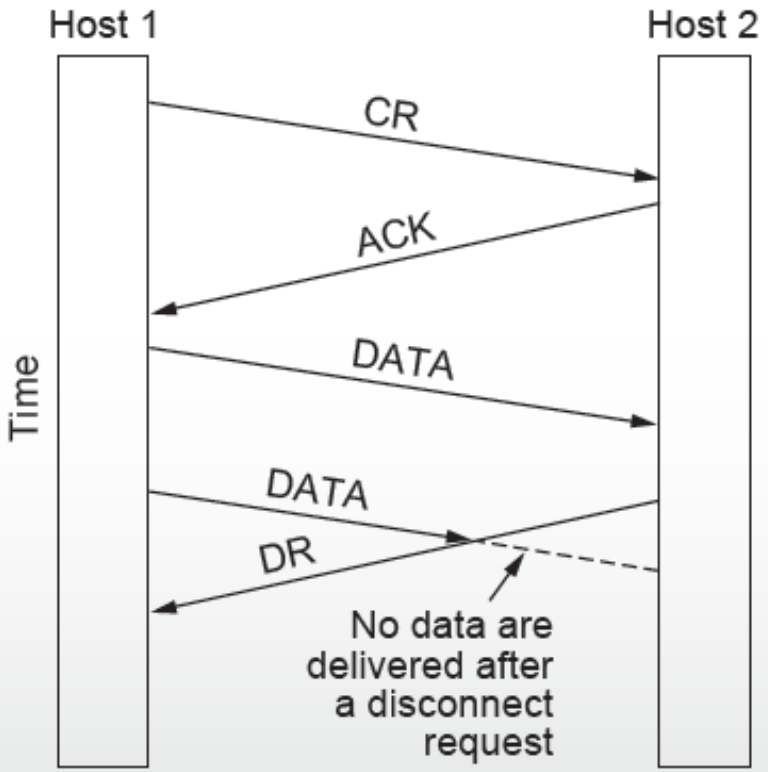
Symmetric release (both sides agree to release) can't be handled solely by the transport layer

- Two-army problem shows pitfall of agreement



Connection Release

- Main aim is to ensure reliability while releasing
- Asymmetric release (when one side breaks connection) is abrupt and may lose data

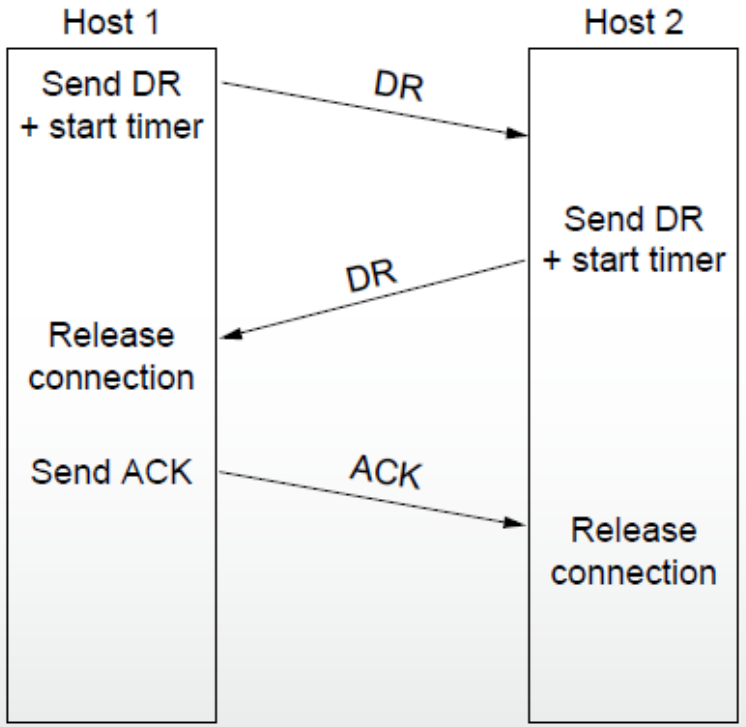


UNIVERSITY OF Southampton
School of Electronics and Computer Science

Connection Release

Normal release sequence, initiated by transport user on Host 1

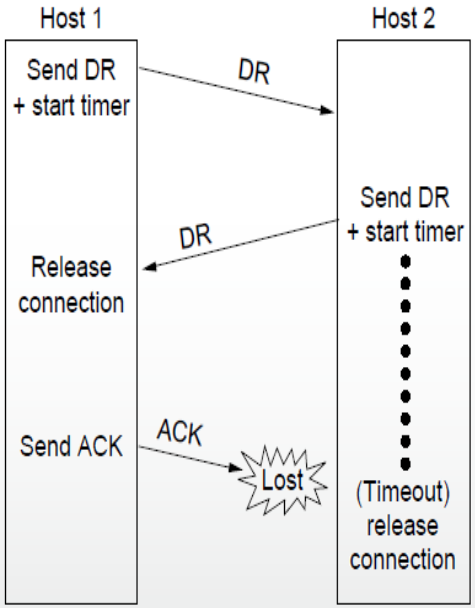
- DR=Disconnect Request
- Both DRs are ACKed by the other side



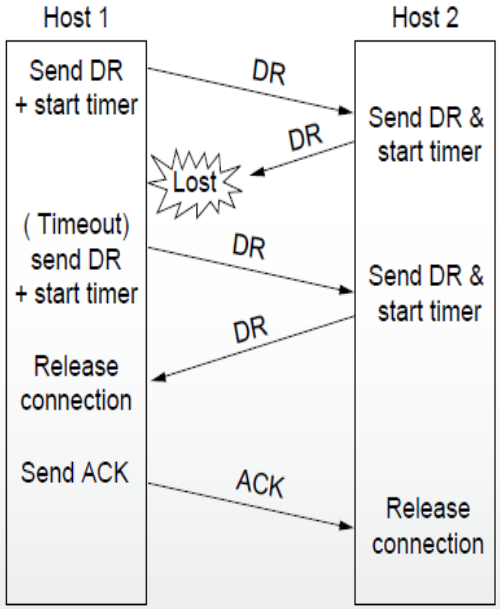
UNIVERSITY OF Southampton
School of Electronics and Computer Science

Connection Release

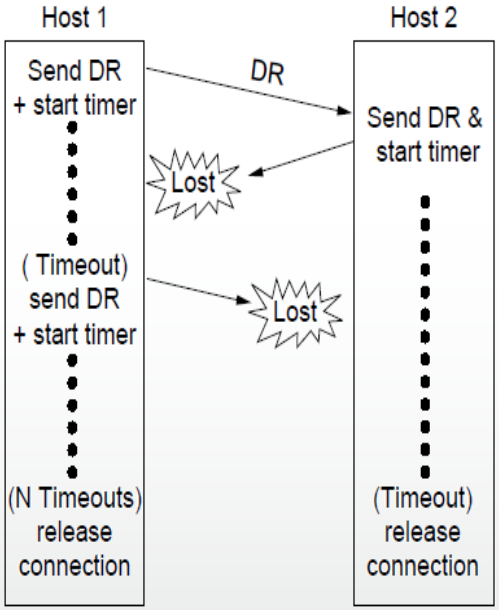
- Error cases are handled with timer and retransmission



Final ACK lost, Host 2 times out



Lost DR causes retransmissions



Extreme: Many lost DRs cause both hosts to timeout

Error Control and Flow Control

Foundation for **error control** is a sliding window (from Link layer) with checksums and retransmissions

Flow control manages buffering at sender/receiver

- Issue is that data goes to/from the network and applications at different times
- Window tells sender available buffering at receiver
- Makes a variable-size sliding window

Error Control and Flow Control

- Flow control example: A's data is limited by B's buffer

	A	Message	B	B's Buffer	Comments
1	→	< request 8 buffers>	→		A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	0 1 2 3	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	0 1 2 3	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	0 1 2 3	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	0 1 2 3	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	1 2 3 4	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	1 2 3 4	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	1 2 3 4	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	1 2 3 4	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	1 2 3 4	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	2 3 4 5	A may now send 5
12	←	<ack = 4, buf = 2>	←	3 4 5 6	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	3 4 5 6	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	3 4 5 6	A is now blocked again
15	←	<ack = 6, buf = 0>	←	3 4 5 6	A is still blocked
16	• • •	<ack = 6, buf = 4>	←	7 8 9 10	Potential deadlock

Summary

- Services provided by transport layer
- Transport service primitives
- Sockets
- Connection establishment and release
- Error/flow control