COMP2221 Networks

David Head

University of Leeds

Lecture 16

Previous lectures

After a brief survey of ports, DNS and IP addressing, we have largely focussed on the Application layer.

- Network application development in Java.
- Either use TCP (`Sockets` and `ServerSockets`) or UDP (`DatagramPacket` and `DatagramSocket`).
- Assume these Transport layer services work as per their protocol.
- No knowledge required of the lower layers (Network, Link and Physical).

Today's lecture

Today's lecture is the first of 4 in which we look at key aspects of these lower levels.

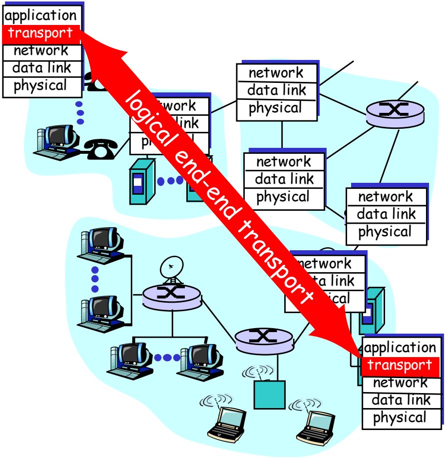
- Useful to understand the **performance** of network applications.

Today we look at the **Transport layer**:

- The structure of UDP and TCP **headers**.
- How to achieve reliable data transfer over unreliable channels.
- **Connection management** and **congestion control** with TCP.

Transport services

- Provide **logical** communication between applications running on different **hosts**/**end systems**.
- Transport protocols run in hosts/end systems.
- Controls the data transfer between **processes**.
- Data transfer between **hosts** handled by lower layers.



Kurose and Ross, *Computer networking: A top-down approach*

UDP: User Datagram Protocol

- **'Best effort'** — UDP segments may be delivered out-of-order, or lost altogether.
- **Connectionless** — no **handshaking** between sender and receiver.
- Each UDP segment is handled independently of the others.

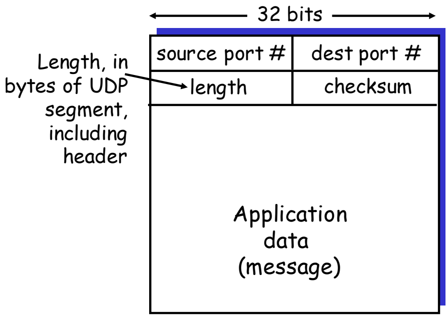
Why is there a UDP?

- Connection establishment can add a **delay**.
- Smaller segment header.
- No **congestion control**, so can send as fast as desired[1].

[1] Although some internet domains may block UDP (except DNS requests) for **precisely** this reason.

UDP Segment structure

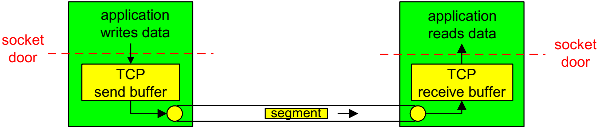
- Often used for **streaming** applications that are **loss tolerant** but **rate sensitive**.
- Can add reliability at the Application layer if needed.
- **DNS** uses UDP (fast; small messages).
- So does **SNMP** (Simple Network Management Protocol) for similar reasons.



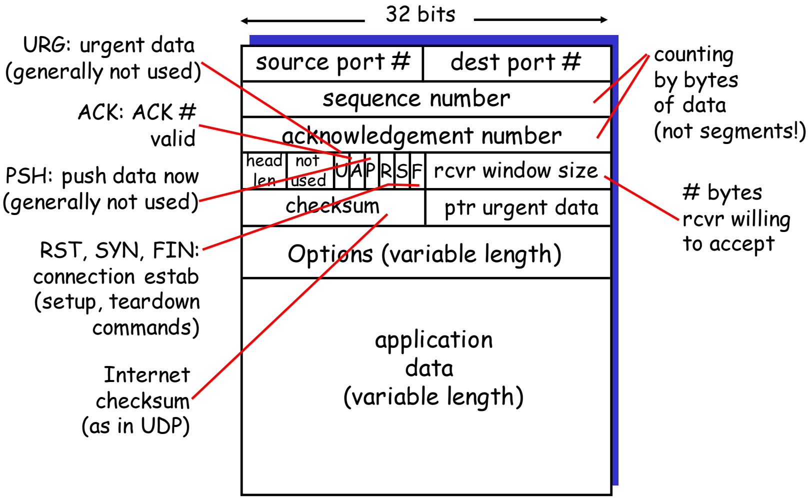
UDP segment format

TCP: Transmission Control Protocol

- **Point-to-point**: One sender, one receiver.
- **Reliable**.
- Send and receive **buffers**.
- **Pipelined**: Send multiple packets without waiting for acknowledgement of the first.
- **Bidirectional** data flow in same connection.
- **Connection oriented**: **Handshaking** (exchange of control messages) **before** actual packet transmission.
- **Flow control**: Sender will not overwhelm the receiver.



TCP Segment Structure



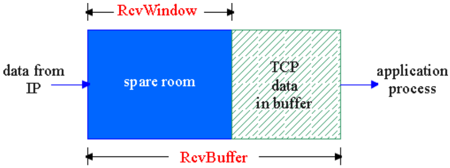
TCP Flow Control

The idea of **flow control** is that the sender won't overrun receiver's buffer by transmitting too much, too fast.

- **Receiver:** Explicitly informs the sender the (dynamically changing) amount of free buffer space.
  - The `RcvWindow` field in TCP segment.
- **Sender:** Keeps the amount of transmitted, unacknowledged data less than most recently received `RcvWindow`.

RcvBuffer = size or TCP Receive Buffer

RcvWindow = amount of spare room in Buffer

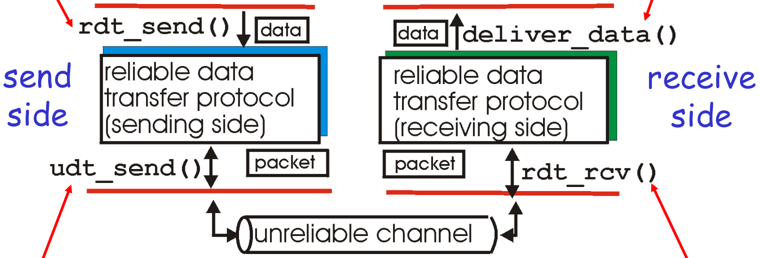


receiver buffering

RDT: Reliable Data Transfer

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

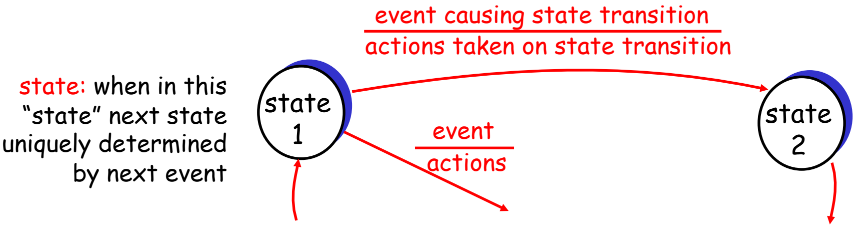
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable transfer over unreliable channels

The principles of reliable data transfer can be considered using **FSMs** = Finite State Machines.

- Need one FSM **each** for sender and receiver.

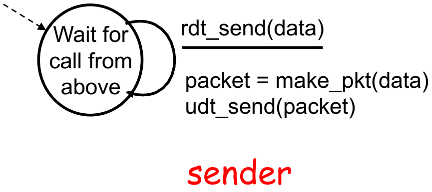
FSM notation — *e.g.* a single FSM with 2 states shown:

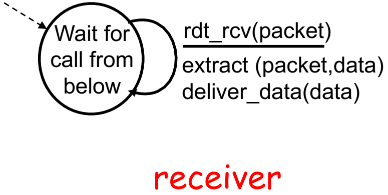


Simplest case: Reliable channel

What it the channel **was** reliable?

- When the **sender** is called 'from above' (*i.e.* by an Application layer process), it **sends** the data.
- When the **receiver** is called 'from below' (*i.e.* from the Network layer), it **extracts** the data.





Channel with bit errors

Assume all packets received, and in order, but some may be corrupted with **bit errors**.

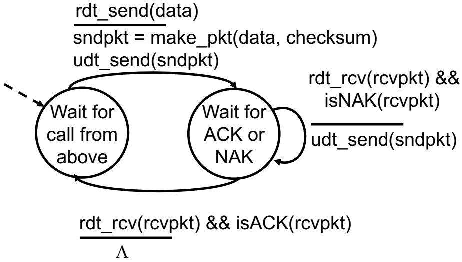
- Can detect for occurrence of by using the **checksum**.
- Assume not enough checksum bits to **correct** the error.

Error recovery utilises positive and negative **acknowledgements**:

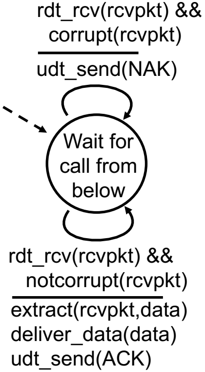
- Receiver sends a **positive** acknowledgement **ACK** if the packet was received without errors.
- Conversely, it sends a **negative** acknowledgement **NAK** if errors were detected.

The sender waits until one acknowledgement or another is received and re-sends the packet if necessary.

Sender[1]**:**



Receiver:



[1] Dashed arrow points to initial state; Λ means no action (just change state).

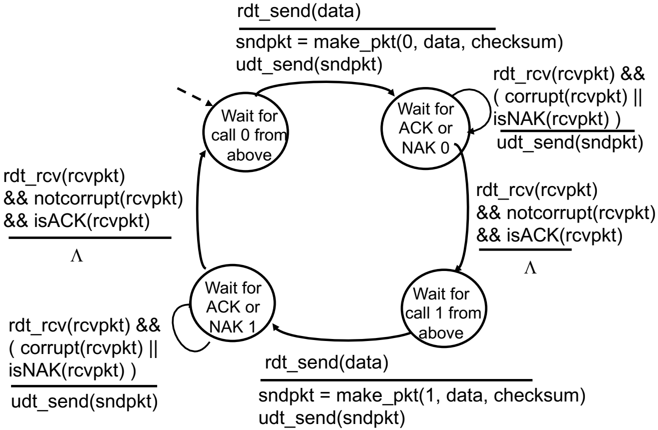
Corrupted acknowledgements

However, this scheme has a fatal flaw — the **ACK** / **NAK** messages can **themselves become corrupted**!

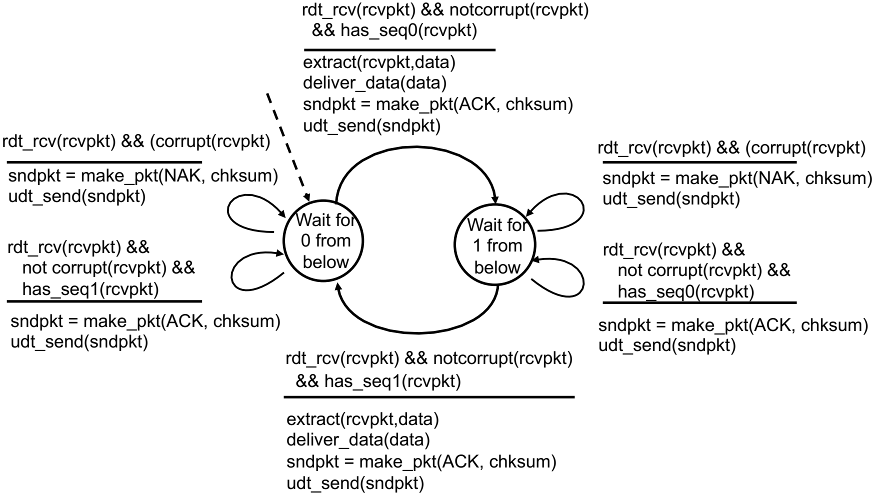
- If the sender detects an error in an acknowledgement, it can re-send the packet.
- But the receiver does not know its acknowledgement was corrupted, and would interpret this as the **next** packet.
- We have **duplicate packets**.

Can solve this by using **sequence numbers**, which is exactly what TCP does — see TCP header structure on an earlier slide.

Sender FSM with sequence numbers 0, 1



Receiver FSM with sequence numbers 0, 1



In principle need just 2 sequence numbers, 0 and 1.

- Actions in state 1 are 'mirror images' of those in state 0.

More sequence numbers required for **performance**, *i.e.* sending **multiple** packets before expecting acknowledgements.

- This is known as **pipelining**.

Have also not yet considered **lossy channels**, which can lose packets entirely.

- Requires more complex program logic / finite state machines.

TCP handles all of these issues for you, albeit with a potential performance loss.

Principles of congestion control

Routers typically have multiple **input** and **output** lines.

- If streams of packets arriving on multiple lines and all need the same output line, a **queue** will build up.
- If **capacity** of the buffer is exceeded, packets will be discarded (lost).

Slow processors can also cause congestion:

- Queueing buffers, updating router tables *etc.* usually require processing.

Different to **flow control**, which is **end-to-end** and does not **explicitly** involve the Network layer.

Congestion control approaches

There are broadly speaking two approaches to congestion control:

End-to-end congestion control:

- *i.e.* **flow control** (see earlier).
- No explicit feedback from the network.
- Congestion inferred from loss and delay observed by the **end systems** or **hosts**.
- This is the approach adopted by TCP.

Network assisted congestion control:

- **Routers** provide feedback to end systems; either ...
- Single bit indicating congestion; or ...
- Explicit rate sender should send at.

TCP Congestion Control

The basic idea is to **probe** for available bandwidth.

- Have a **congestion window** `CongWin`, which would ideally be as large as possible so as to transmit as fast as possible (without loss).
- Increase `CongWin` until loss.
- Decrease `CongWin` if loss, then begin probing (increasing) again.

There are two phases:

1. **Slow start**;
2. **Congestion avoidance**.

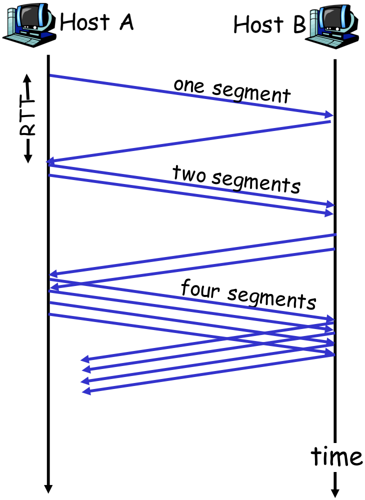
A **threshold** defines when to switch between these two phases.

TCP Slow start

Slow start algorithm:

Initialise `CongWin=1` MSS.
for each segment ACKed: `CongWin++`
until(loss OR `CongWin`>threshold)

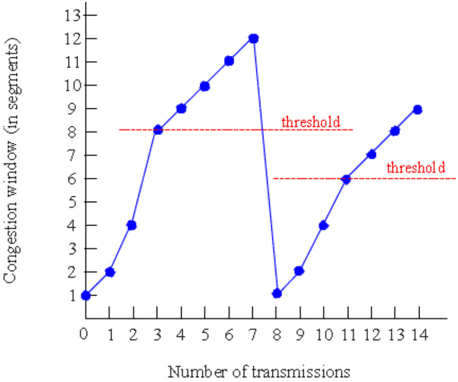
- **MSS**: Maximum Segment Size.
- **RTT**: Round Trip Time.
- **Exponential** increase as increments happen more quickly (not such a slow start!)
- **Loss event**: Timeout and/or three duplicate ACKs.



TCP Congestion Avoidance

Congestion avoidance algorithm:

```
/* slow start is over */
/* CongWin>threshold */
Until( loss event ) {
    every w segments ACKed:
        CongWin++
}
threshold = CongWin/2
CongWin=1
perform slow start
```



Overview and next lecture

Today we have covered the Transport layer that lies immedialetly underneath the Application layer.

- Services provided by TCP and UDP, and the structure of their respective headers.
- Connection management and congestion control in UDP.

See chapter 3 of Kurose and Ross, *Computer Networking: A Top-Down Approach* (7^{th} ed.) for more details.

For the next two lectures we will look at the next layer down, the Network layer.