Transport Services and Protocols

Transport Services and Protocols

- The **transport layer** is responsible for providing **logical communication** between **application procceses** running on **different hosts**.
- There are two actions the transport layer needs to achieve:
  1. **Send action**: The transport layer needs to **divide application messages into segments**, and pass them to the **network layer**.
  2. **Recieve action**: The transport layer needs to **reassemble segments** from the **network layer** into **messages**, and pass them to the **application layer**.
- There are **two transport protocols** available: the **User Datagram Protocol (UDP)**, and the **Transmission Control Protocol (TCP)**.

The Application Layer, The Transport Layer, and The Network Layer

- The **application layer** is the **process** that is running on the **network host**.
- The **transport layer** is responsible for the **logical communcation** between **processes**.
- The **network layer** is responseible for the **logical communication** between **network hosts**.
- When a **message is sent** from an application over the network, it follow the following order: application layer → transport layer → network layer.
- When a **message is received** from the network, and sent to an application, it follows the following order: network layer → transport layer → application layer.

Transport Layer Actions

- When a message is **sent from an application**, the **transport layer** does the following:
  1. Recieves the application-layer message.
  2. Determines segment header field values.
  3. Creates the segment.
  4. Passes the segment to the internet protocol.
- When a message is **recieved from the network**, the **transport layer** does the following:
  1. Recieves the segment from the internet protocol.
  2. Checks the header values.
  3. Extracts the application-layer message.
  4. Demultiplexes message up to the application via a socket.

User Datagram Protocol and Transmission Control Protocol

- The **User Datagram Protocol (UDP)** is an **unreliable**, **unordered** delivery protocol. This protocol has **minimal overhead** but is **not reliable**.
- The **Transmission Control Protocol (TCP)** is a **reliable**, **in-order delivery** protocol that supports **congestion control**, **flow control**, and **connection setup**. This protocol has **significant overhead**, but is **very reliable**.
- Both **UDP** and **TCP** do not provide **delay guarantees** and **bandwith guarantees**.

Multiplexing

Multiplexing

- **Multiplexing** is a method used by **networks** to **consolidate multiple signals** into a **single composit signal** that is then **transported over a common medium**.
- When **sending data**, **multiplexing** is used to **transmit segments** from **several sockets** via **transport headers**.
- When **recieving data**, the **header info** is used to **demultiplex** the data, sending it to the **correct socket**.
- **Demultiplexing** works by receiving **IP datagrams** containing a **source and destination IP address and port as headers**. Each datagram contains a **single segment**.
  - The IP addresses and port numbers are used to route the segment to the correct socket.
  - This is why you must specify a port (and sometimes an address) number when creating a socket.

Connectionless Multiplexing

- When creating a **server socket** you must specify a **port**, that the **socket will be bound to**.
- When creating a **datagram (UDP)** you must specify the **destination host and port**.
- When a **host** receives a **datagram** with the **port destination** that matches the **port the server socket is bound to**, it will **route the datagram** to **said socket**.
  - The host address does not matter when a datagram is recieved by a host, it will simply attempt to route it to the destination port. This means that several hosts can send datagrams to the same port, and they will all be available at the same socket.

Connection-Oriented Multiplexing

- When creating a **connection oriented socket (TCP)**, you create a **4-tuple** containing the **source address, source port, destination address, and destination port**.
- The **4-tuple** is used to **route inbound packets** to the **correct socket**.
- **Servers** may support **several simultaneous TCP sockets**, typically one for **each client**. Each socket will have a **unique 4-tuple**.

Summary

- Multiplexing and demultiplexing are based on segman, datagram header values.
- UDP uses the port for demultiplexing.
- TCP uses the 4-tuple for demultiplexing.
- Multiplexing happens at all layers.

User Datagram Protocol

User Datagram Protocol

- The **User Datagram Protocol (UDP)** is a **connectionless (no handshake) communication protocol** that uses the **internet protcol**.
- **UDP** has **minimal overhead**, at the cost of having **no connection state, no congestion control, and no packet delivery verification**.
- **UDP** is used for **loss-tolerant, rate sensitive** applications.
- It is possible to have **reliable UDP communication** by implementing it at the **application layer**.

```
                    User Datagram Protocol
                    ----------------------

Introduction
------------

This User Datagram  Protocol  (UDP)  is  defined  to  make  available  a
datagram   mode  of  packet-switched   computer   communication  in  the
environment  of  an  interconnected  set  of  computer  networks.   This
protocol  assumes  that the Internet  Protocol  (IP)  [1] is used as the
underlying protocol.

This protocol  provides  a procedure  for application  programs  to send
messages  to other programs  with a minimum  of protocol mechanism.  The
protocol  is transaction oriented, and delivery and duplicate protection
are not guaranteed.  Applications requiring ordered reliable delivery of
streams of data should use the Transmission Control Protocol (TCP) [2].

Format
------


                  0      7 8     15 16    23 24    31
                 +--------+--------+--------+--------+
                 |     Source      |   Destination   |
                 |      Port       |      Port       |
                 +--------+--------+--------+--------+
                 |                 |                 |
                 |     Length      |    Checksum     |
                 +--------+--------+--------+--------+
                 |
                 |          data octets ...
                 +---------------- ...

                      User Datagram Header Format
```

- **UDP datagrams** have the following headers:
  1. **Destination port** — The port the datagram is being sent to.
  2. **Length** — The length in octets of the datagram.
  3. **Checksum** — The checksum is used to ensure the packet received was not corrupted during transmission.
  4. **Pseudo** — A header containing the source address, destination address, the protocol, and the UDP length. This information is used to give protection against misrouted datagrams.
- The **checksum** is computed by taking the **one's complement sum** of the **entire content** of the **datagram**, treating the bits of the datagram as a **series of 16-bit integers**.
  - The checksum is not perfect, several different datagrams can result in the same checksum.

Reliable Data Transfer

Reliable Data Transfer Overview

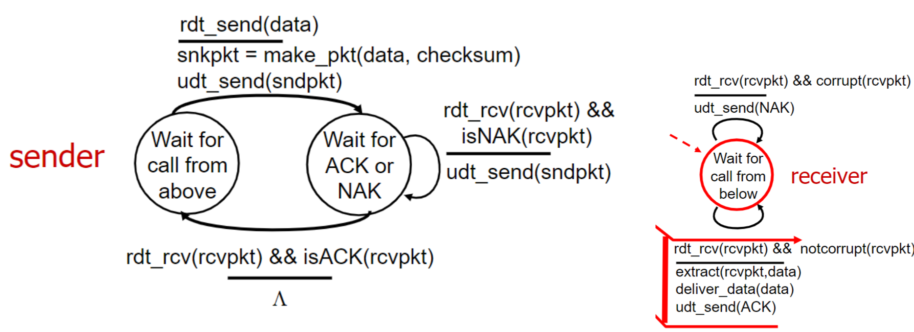
- The **complexity** of the **reliable data transfer protocol** will strongly **depend** on the **characteristics** of the **unreliable channel** (drop packes, corruption, data reordering, etc).
- The **sender** and **receiver do not know the state of each other**, unless it is **communicated**.

RDT 1.0 — Reliable Transfer over a Reliable Channel

- In this case, the **underlying channel** is perfectly **reliable**; there are **no bit errors**, and **no loss of packets**.
- This is the most primitive form of reliable data transfer; the **sender sends the data**, and the **receiver reads the data**.

RDT 2.0 — Channel with Bit Errors

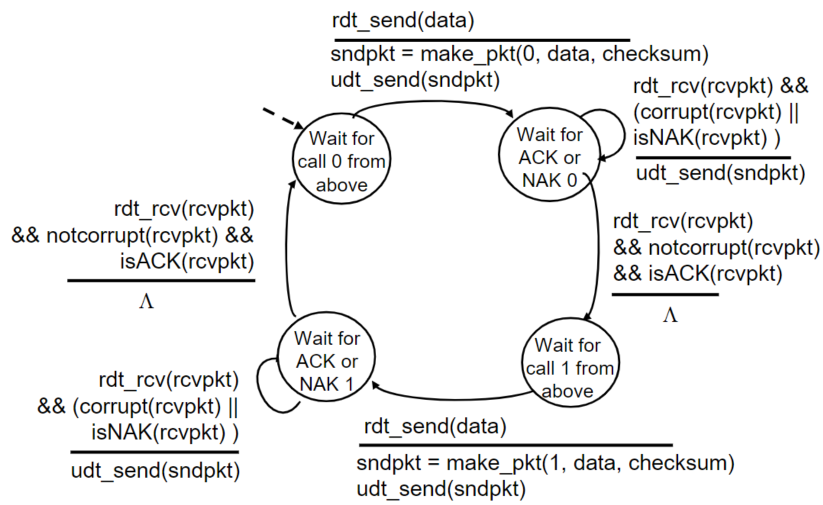
- In this case, the **underlying channel** may **flip bits in packets randomly**. To determine if this happend we can use a **checksum**.
- To **recover from errors**, we can have the **receiver send a respone**.
  - The **acknowledgement (ACK) response** indicates that the **packet was successfully transmitted**.
  - The **negative acknowledgement (NAK) response** indicates that the **packet was not successfully transmitted**.
  - If the **sender** recieves a **NAK response**, they must **retransmit the packet**.
- The **sender** must **stop and wait** for a resonse, before sending the next packet.

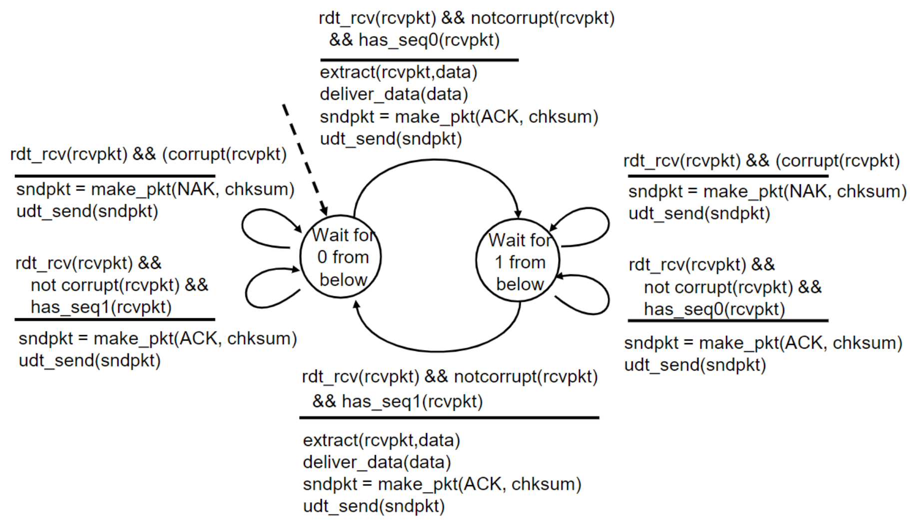


- **RDT 2.0 has a fatal flaw**, if the **ACK** or **NAK** gets **corrupted**, the **sender** won't know if the **receiver** received the packet.
  - You can't retransmit the packet, it could result in a possible duplicate.

RDT 2.1 — Handling Corrupted ACKs and NAKs

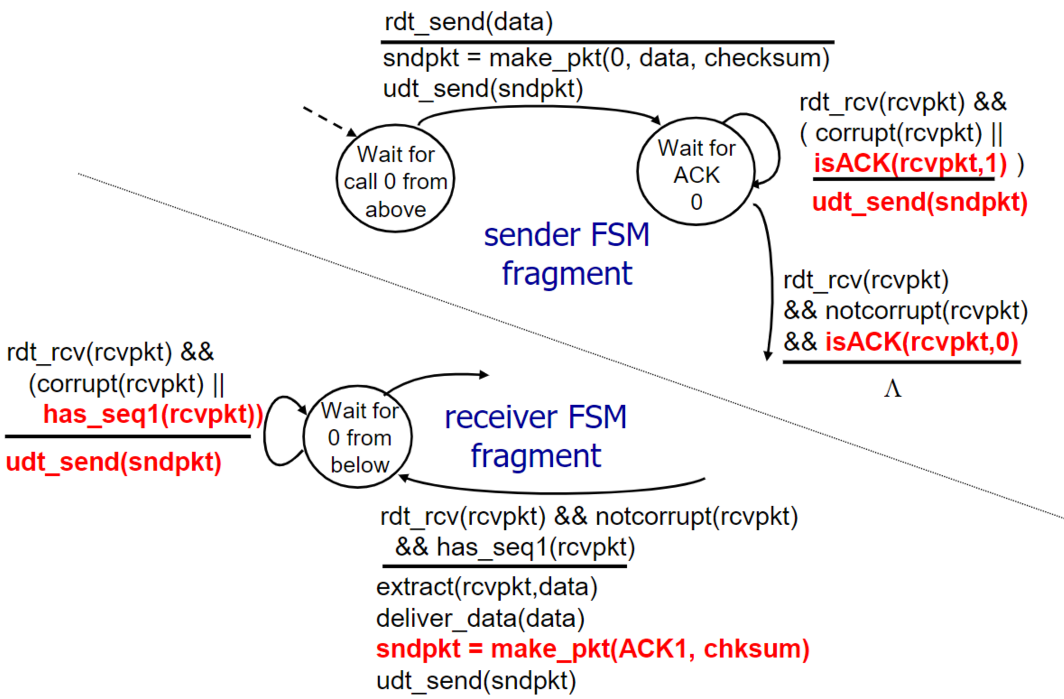
- In the case where the **receiver's response** gets **corrupted**, we can add a **sequence number to each packet**. This will allow the reciever to detect duplicate packets.





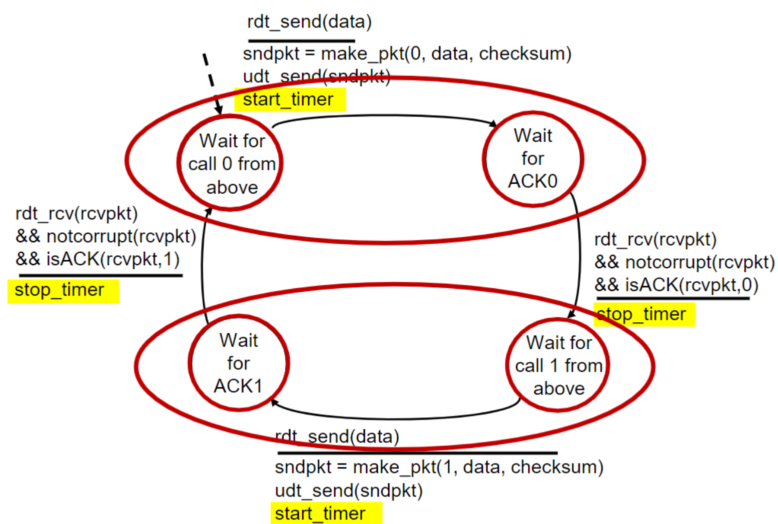
RDT 2.2 — A NAK-free Protocol

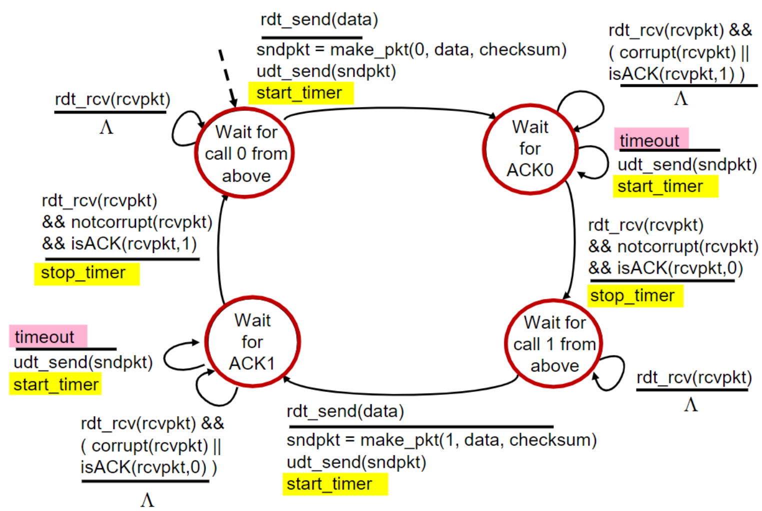
- This protocol has the **same functionality as 2.1**, but it only uses **ACKs**.
- **Instead of NAK**, the **receiver** sends **ACK** for the **last packet recieved OK**.
  - The receiver must explicitally include the sequence number of the packet being ACKed.
- A **duplicated ACK** at the **sender** results in the **same action as NAK**; retransmit the current packet.



RDT 3.0 — Channels with Bit Errors and Packet Loss

- In this case the **underlying channel** can **lose packets, and corrupt packets**.
- The checksum, sequence of numbers, ACKs, and restransmission will help, but will not be enough.
- In this protocol, the **sender waits a "reasonable" anout of time** for an **ACK**, if **ACK is not recieved** within a reasonable amount of time, the **sender** will **retransmit the packet**.
  - If the packet ACK was just delayed and not lost, the sequence number will allow to reciever to discard the duplicate packet.
  - When the reciever is sending the ACK, they must specify the packet number they are ACKing.





rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
Wait for ACK1
Wait for call 1 from above
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ