**Principles of Reliable Data Transfer**

Reliable Data Transfer = RDT

**Introduction**

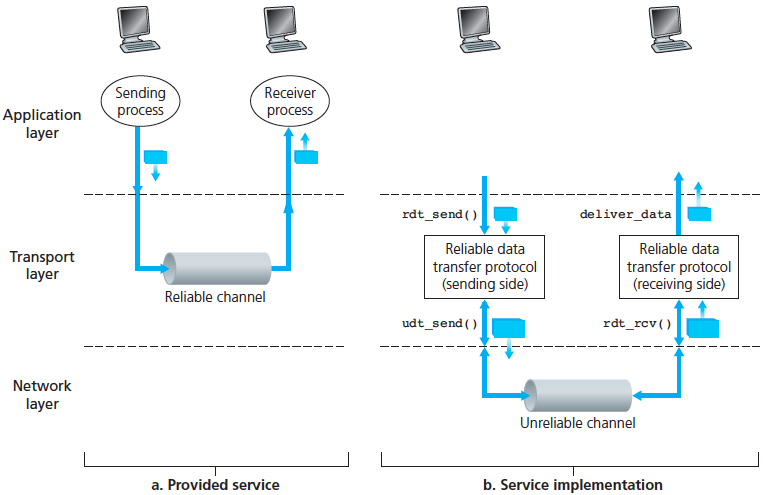
With RDT, transferred data bits are: 'not corrupted' | 'not lost' | 'delivered in order'

* TCP offers this service model to internet applications that use it
* However, layers below RDT Protocol may be unreliable.

**'rdt\_send()'** = Pass data from sender-side app to be delivered to the receiver-side app

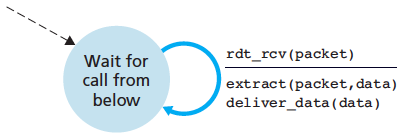
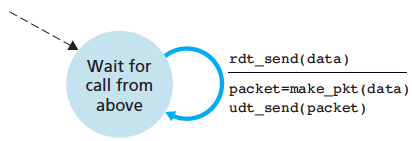
**'rdt\_rcv()'** = Called when packet arrives to receiver-side channel

**'deliver\_data()'** = Delivers the data to receiver-side app



**Building a Reliable Data Transfer Protocol**

**RDT 1.0 – Transfer over a perfectly reliable channel (not a realistic model)**



All packet flow is from sender 🡪 receiver.

No need for receiver-side to provide feedback to sender.

Also assumption that receiver is able to receive data as fast as the sender sending data, thus no need for flow/cong control.

**RDT 2.0 – Transfer over a channel with bit errors (more realistic model)**

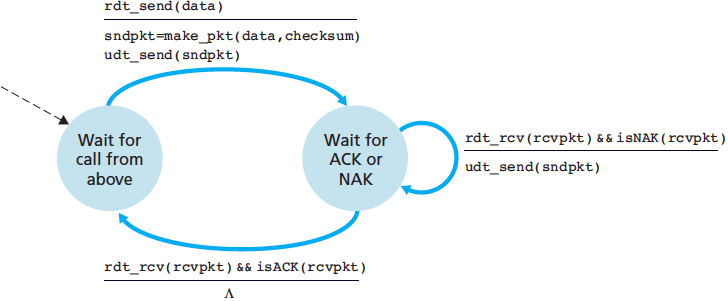
In this model, we assume packets can be corrupted. Bit errors occur in physical components of a network as a packet is transmitted, propagated or buffered.

This model is based on retransmissions of data, known as: ARQ: Automatic Repeat Request Protocols.

Three additional protocol features are required in ARQ protocols to handle bit errors:

1. Error Detection: E.g. UDP using internet checksum (gathered in checksum field of a data packet)
2. Receiver Feedback: Positive ACK and Negative NACK replies from receiver 🡪 sender.  
   Only needs to be 1 bit-long. 0 = NAK | 1 = ACK
3. Retransmission: Packet received in error will be retransmitted by the sender

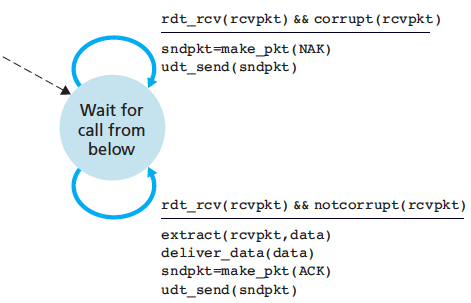
RDT 2.0 is



* Sender waits for data to be passed down
* When **rdt\_send() occurs**, sender creates a packet **sndpkt + checksum**
* Send packet via. **Udp\_send()**
* Wait for ACK packet: **rdt\_rcv(rcvpkt) && isACK**
* If true, go back to waiting for data from app layer.
* Else retransmit last packet + wait for ACK again

NOTE: If sender is waiting for ACK/NACK it can’t receive more data from upper layer. ( rdt\_send() can’t occur)

**This is a STOP-AND-WAIT protocol**.



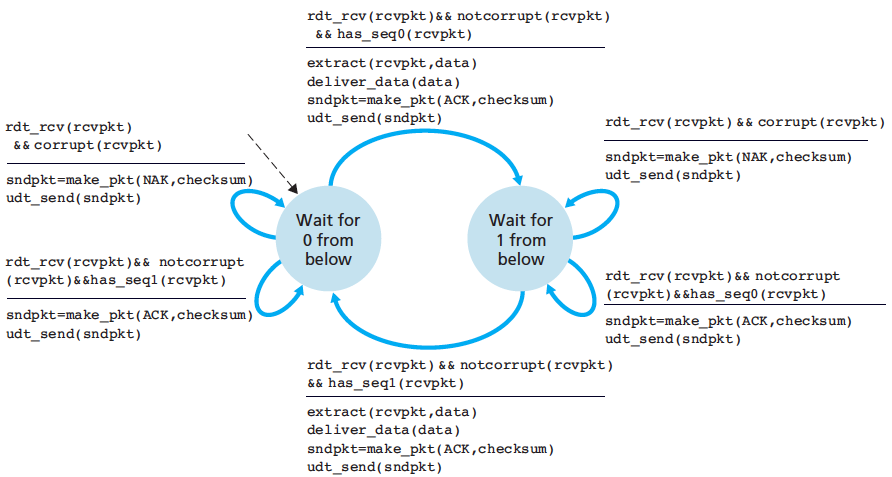
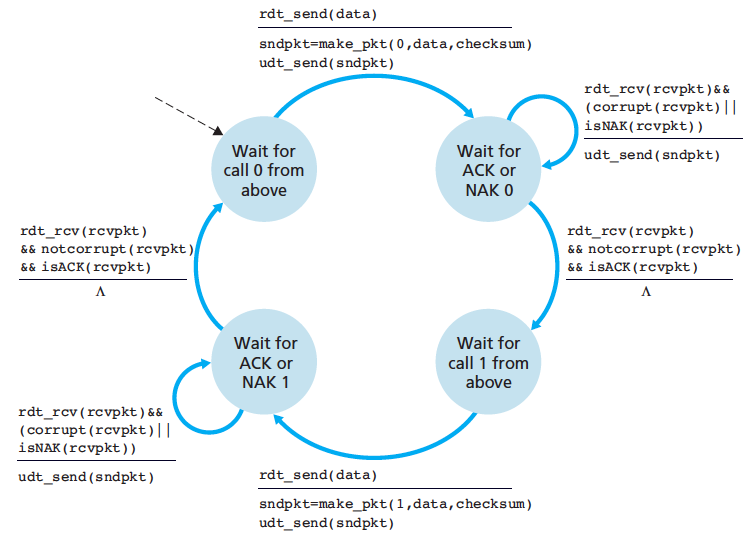
* On packet arrival, receiver replies with ACK/NACK depending on if packet is corrupted.
* Corrupted = **rdt\_rcv(rcvpkt) && corrupt()**  
  -> send NAK packet via. Udt\_send()
* Uncorrupted = **rdt\_rcv(rcvpkt) && notcorr()**  
  -> extract data  
  -> deliver to upper layer  
  -> send ACK packet via. Udt\_send()

Fatal flaw with RDT 2.0 = ACK/NACK packets themselves could be corrupted.

* Solution: add a new field into the data packet and have the sender number its data packets by putting a **sequence number**.
* Receiver only needs to check this sequence number to determine if the received packet is a retransmission.

**RDT 2.1 – Protocol state now reflects whether a packet being sent or expected should have sequence number 0 or 1**

0-numbered packet being sent or expected = 1-numbered packet is being sent or expected. (if correctly sending/recv)



* When out-of-order packet is received, receiver sends **pos acknowledgement** for the packet it has received.
* When corrupted packet is received, receiver sends **neg acknowledgement**.
* A sender that receives two ACKS for the same packet = receiver did not correctly receive the packet

**RDT 3.0 – Transfer over a LOSSY CHANNEL with BIT ERRORS**

Suppose in addition to corrupt bits, the underlying channel can lose packets. This means two additional concerns:

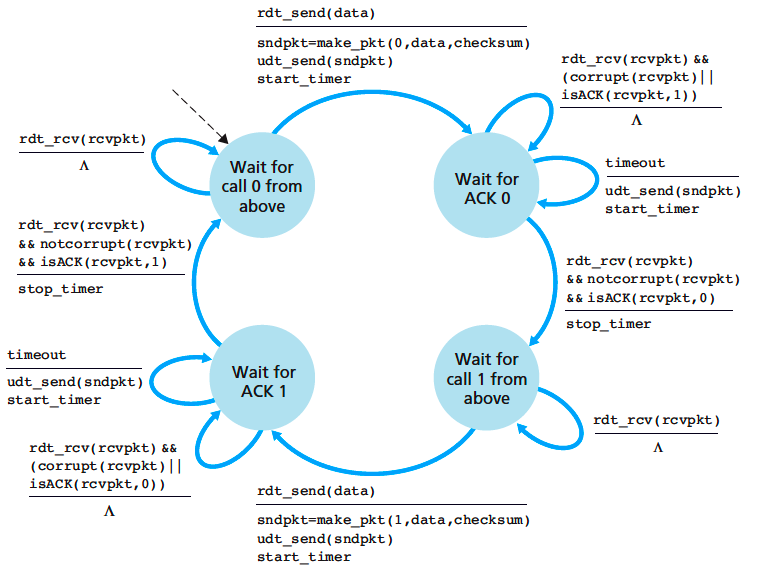
* (1) How to detect packet loss (2) What to do when packet loss occurs.
* Checksums, seq numbers, ACK packets and retransmission solve error 2.
* New protocol methods are needed to detect packet loss.
* We will assume both DETECTON and RECOVERING from lost packets will addressed by sender side.

Suppose the sender transmits a data packet:

* Either that packet or receiver’s ACK of the packet gets lost.
* If the sender is willing to wait long enough so that it is certain the packet is lost, it can simply re-transmit the packet.
* ^However, **large delays =** packet is not actually lost, but sender may retransmit the packet anyway = **duplicate packets**

From the sender’s viewpoint, retransmission is an all-in-one solution: doesn’t matter if packet/ACK was lost or overdelayed

* In all cases, the sender will just need to re-transmit the data.
* Implementing time-based re-transmission requires a **countdown timer** that can interrupt the sender after a given amount of time has expired.
* The sender will need to be able to:  
  **(1) start time after each packet – either 1st time or retransmission packet is sent  
  (2) respond to a timer interrupt – taking appropriate actions  
  (3) stop the timer**



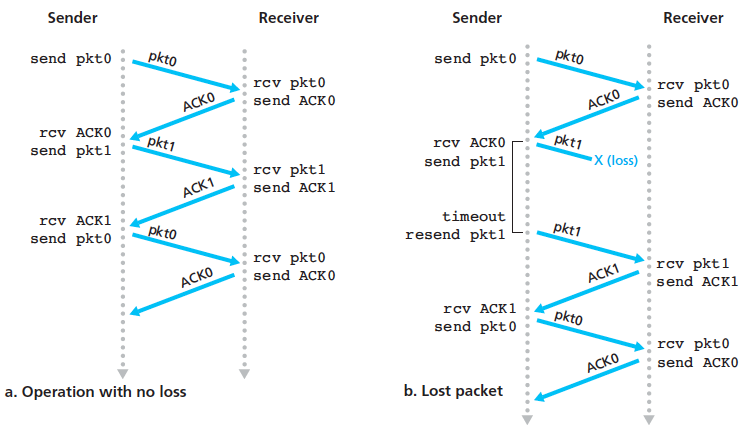
Now we have all the features required for a working, reliable data transfer protocol:

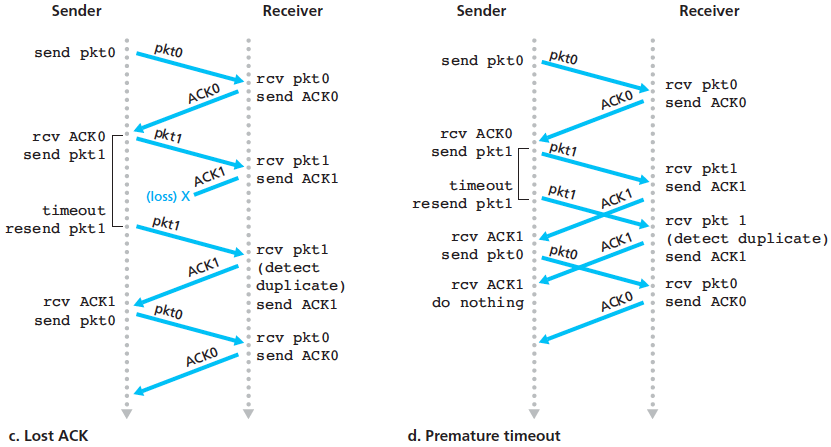
* **Checksums, sequence numbers, timers, positive and negative acknowledgement packets**.

**Pipelined Reliable Data Transfer Protocols**

The RDT 3.0 is a functionally correct protocol, however it is unlikely that anyone would be happy with its performance, due to the fact that RDT 3.0 is a **Stop-And-Wait** protocol.

Consider the performance of Stop-And-Wait behaviour: NO LOSS vs. LOST PACKET vs. LOST ACK vs. PREMATURE TIMEOUT





**SKIP TO 3.5 CONNECTION-ORIENTATED TRANSPORT: TCP**

**Reliable Data Transfer**

TCP creates a RDT on top of IP’s unreliable best-effort service.

* TCP ensures that the data stream is uncorrupted, without gaps, without duplication and in sequence (byte stream is the same byte stream sent by the end system on the other side of the connection)

Earlier, we assume that an individual timer is associated with each transmitted (but not ACK’d) segment.

* This is great in theory, but timer management can require considerable overehead.
* Recommended TCP timer mgmt. uses only a **single retransmission timer**, even if there are multiple transmitted but not yet acknowledged segments.

**SKIPPED SOME NOTES HERE**

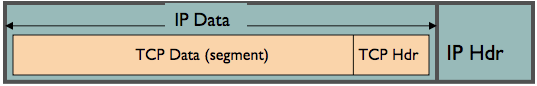
**SUMMARY: Components of Reliable Data Transfer**

1. **Checksums** (for error detection)
2. **Timers** (for loss detection)
3. **Acknowledgements**
   * Cumulative, Selective
4. **Sequence numbers** (duplicates, windows)
5. **Sliding Windows** (for efficiency)
   * Go-Back-N (GBN)
   * Selective Replay (SR)

**TCP is similar to previous components of RDT, but with some key differences**

* **Checksum:** TCP calculates checksum over both the HEADER (TPC “pseudo” header) and DATA (TCP segment)

<http://lateblt.tripod.com/bit34.txt>

* **Sequence & Acknowledgement numbers:** Are byte offsets
  + TCP segments are sent when (1) Segment is full (2) A timeout occurs
  + Segment Size:  
    
    - IP Data: No bigger than **Max Transmission Unit (MTU)**
    - TCP packet: IP packet with a TCP header and data
    - TCP segment: No more than **Max Segment Size (MSS)** **[ MSS = MTU – IP\_HEADER – TCP\_HEADER ]**
  + How it works: <http://www.rhyshaden.com/tcp.htm>

Host 2 receives the **SYN** with the Sequence number **A** and sends a **SYN** segment with its own totally independent ISS number **B** in the Sequence number field. In addition, it sends an increment on the Sequence number of the last received segment (i.e. **A+x** where **x** is the number of octets that make up the data in this segment) in its Acknowledgment field. This **Acknowledgment** number informs the recipient that its data was received at the other end and it expects the next segment of data bytes to be sent, to start at sequence number **A+x**. This stage is often called the **SYN-ACK**. It is here that the **MSS** is agreed.

Host 1 receives this **SYN-ACK** segment and sends an **ACK** segment containing the next sequence number (**B+y** where **y**is the number of octets in this particular segment), this is called **Forward Acknowledgement** and is received by Host 2. The **ACK** segment is identified by the fact that the **ACK** field is set. Segments that are not acknowledged within a certain time span, are retransmitted.

Example: One-way transmission of data from Client 🡪 Server

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **C-S** | **S** | **A** | **L** | **Description** |
| **Client** | 0 | 0 | 0 | I want to initiate a connection (SYN) |
| **Server** | 0 | 1 | 0 | I want to initiate a connection AND acknowledge your SYN packet (SYN-ACK) |
| **Client** | 1 | 1 | 0 | I acknowledge your SYN packet (ACK) |
| **Three-Way-Handshake Complete** | | | |  |
| **Client** | 1 | 1 | 25 | I am sending you data of size 25 bytes |
| **Server** | 1 | 1+25 | 0 | I acknowledge your data. I expect the next packet to have bytes starting at seq 25+1. |
| **Client** | 1+25 | 1 |  |  |
| **Server** |  |  |  |  |
| **Client** |  |  |  |  |
| **Server** |  |  |  |  |
| **Client** |  |  |  |  |

**C**: seq=0 | ack=0 | len=0 🡪 I want to initiate a connection (SYN)

**S**: seq=0 | ack=1 | len=0 🡪 I want to both initiate a connection AND acknowledge your SYN packet (SYN-ACK)

**C:** seq=1 | ack=1 | len=0 🡪 I acknowledge your SYN packet (ACK)

**C**: seq=1 | ack=1 | len=25 🡪 I am sending you data of size 25 bytes

**S**: seq=1 | ack=1+25 | len=25 🡪 I acknowledge the packet sent | I expect the next packet to start at seqno X+B+B

**C** seq = X+2B | len = B 🡪 I am sending you a packet

Sequence number + length = confirmation of ACK