Transport Layer Part II

Jack Schefer (js7ke)

February 27, 2020

1 Review: Socket Programming

When writing application-layer code, programmers typically do not want to have to deal with the nuances of physical data transport. Application programmers want an interface that abstracts all of that away so they can communicate between machines in a simple manner. **The goal of sockets is to provide a usable interface for programs on different machines to communicate with each other across the network.**

When initializing a socket, both an IP address and a port must be specified. The IP address distinguishes which machine on the network and the port selects an individual program running on that machine. This notion of using ports to differentiate individual programs is called **multiplexing/de-multiplexing**. Some port numbers are conventionally used for different types of programs. For example: port 80 is used for HTTP traffic, port 443 is for HTTPS traffic, port 22 is for SSH, and so on.

Sockets can support either the Transport Control Protocol (TCP) or the User Datagram Protocol (UDP). TCP is generally used to keep a session open for multiple messages going back and forth. To create a TCP socket, use the SOCK_STREAM argument. UDP does one-way datagram communication and you can create a UDP socket with the SOCK_DGRAM argument.

2 Reliable Data Transfer Designs

We now begin the design of a reliable data transfer protocol. **The lower levels of the OSI stack are inherently unreliable** (IQ sampling noise errors, packet loss during queuing, etc.) but application programmers do not want to have to deal with that. This is the purpose of the transport layer: to add reliability to network communication to ensure data is sent and received when expected.

In designing our reliable data transfer protocol (rdt), we begin by assuming much about the reliability of our network. We then sequentially relax our assumptions until we have a fully unreliable network. We will make use of a couple functions we assume the existence of a few key helper functions as described in Figure 1. By "design a protocol" we mean to fill in the logic of the blue and green boxes to ensure reliability from the point of the view of the application code.

To summarize these functions we have:

1. rdt_send(): called by an application-layer program when it wants to send data across the network.
2. udt_send(): attempts to send a packet over the unreliable channel, may or may not fail.
3. rdt_rcv(): called when a packet is received over the unreliable channel.

4. deliver_data(): a function that passes data from the receiving machine to the application code that is waiting for information from the network.

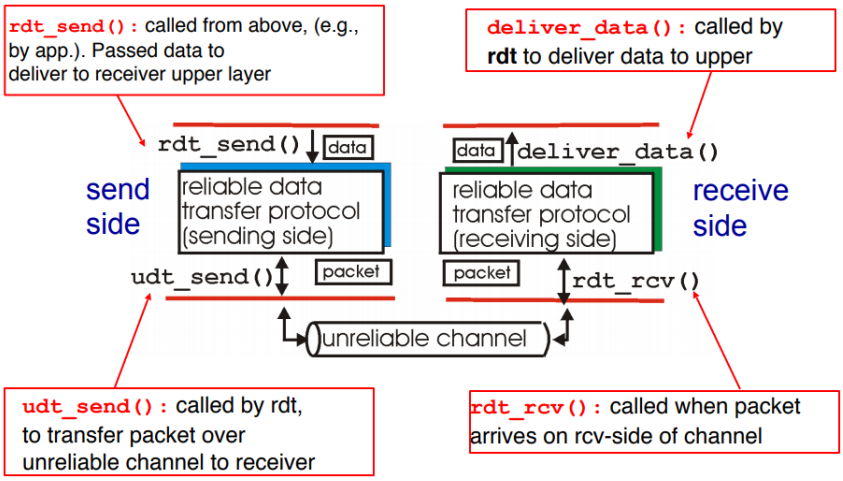

Figure 1: Basic flow of rdt algorithms

In the Finite State Machine (FSM) diagrams below, note that the horizontal bars on each state transition show both when the transition should take place (above the bar) and what additional side effects the transition should make (below the bar). An uppercase lambda (Λ) represents that the process is finished and the FSM will do nothing, waiting for the next stimulus.

rdt1.0

For version 1.0, we assume that the physical network channel is perfectly reliable. That is, there are no bit errors or packets lost during transport. This makes our protocol very easy: when a program calls rdt_send() our protocol just immediately creates a packet and sends it over the channel. On the receiving side, the protocol unwraps the packet and delivers the data.

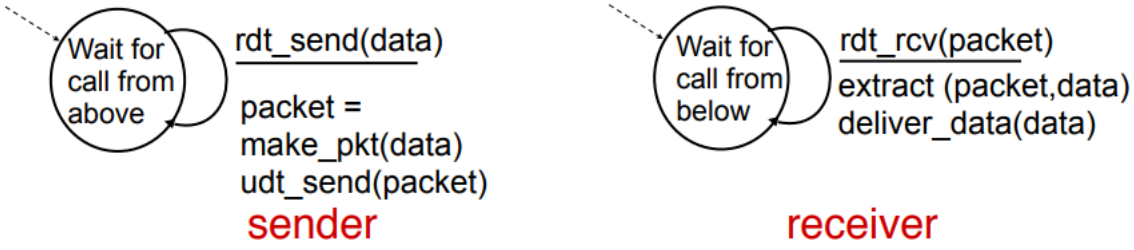

Figure 2: rdt1.0 Finite state machine design

rdt2.0

For version 2.0, we relax the bit-error assumption. That is, we assume that no packets are lost but we assume that the channel may flip some bits in our packets.

We handle bit errors by adding a checksum to our packet so that bit errors can be detected and by adding acknowledgement messages to communicate when packets have been received correctly. That is, if a receiver gets an uncorrupted packet it will respond with an ACK message and if it gets a corrupted packet it will respond with a NAK message.

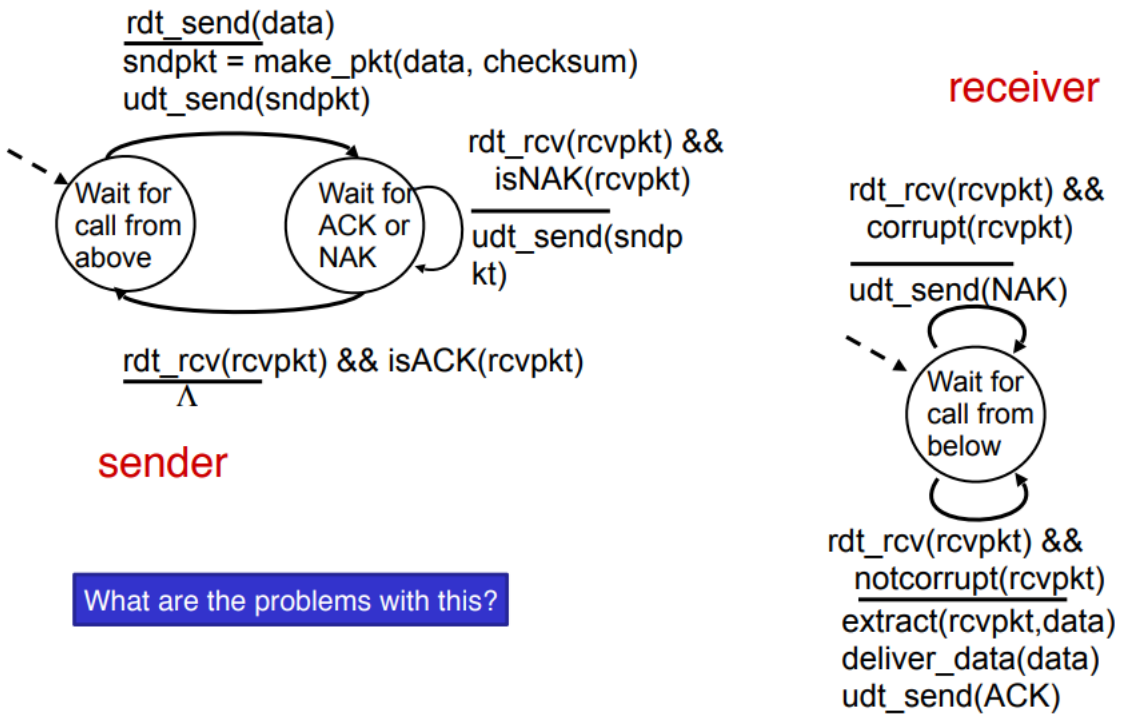

Figure 3: rdt2.0 Finite state machine design

rdt2.1

As Figure 3 alludes to, there are potential problems with the rdt2.0 FSM design. Consider what would happen if the ACK/NAK response got corrupted in transport. According to the prior design, the sender would re-send that packet over the channel and the receiver would then receive it again. Assuming there is no corruption, the receiver will deliver the same packet *a second time* to the application layer causing duplicates.

To fix the duplication problem, we introduce the notion of packet sequence numbers. This allows the receiver to check the sequence number it's receiving to determine if it is duplicate of the previously sent packet. Cleverly, we only need two sequence numbers (0 or 1) because an ACK/NAK can only refer to the last sent packet or the one before that. We encode these sequence numbers as part of the states in the state machines, doubling the number of states on each end.

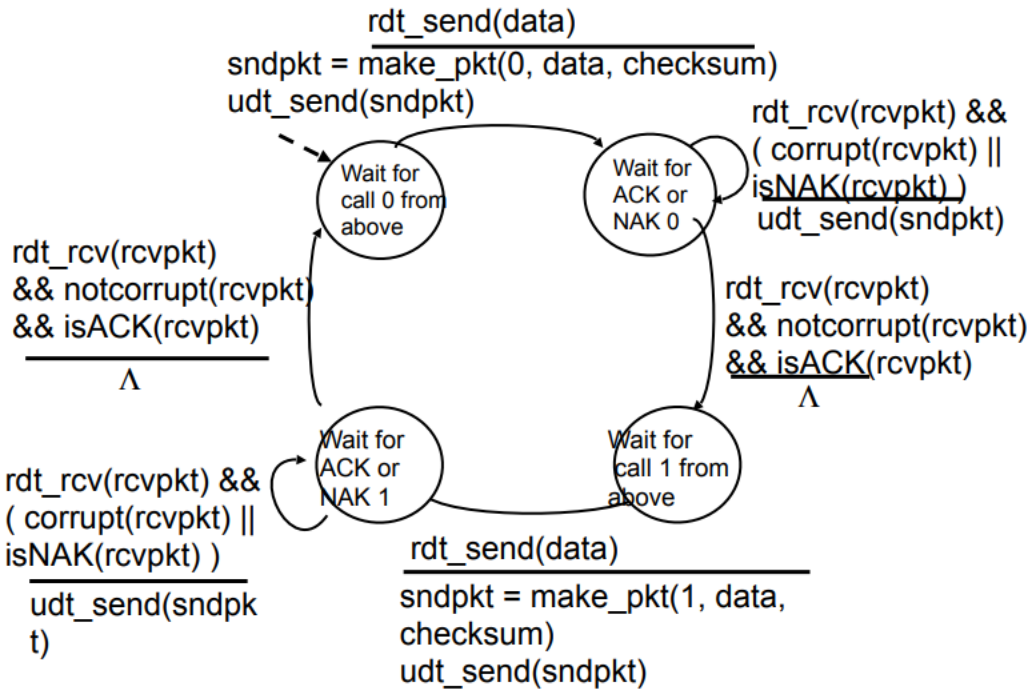

Figure 4: rdt2.1 Sender finite state machine design

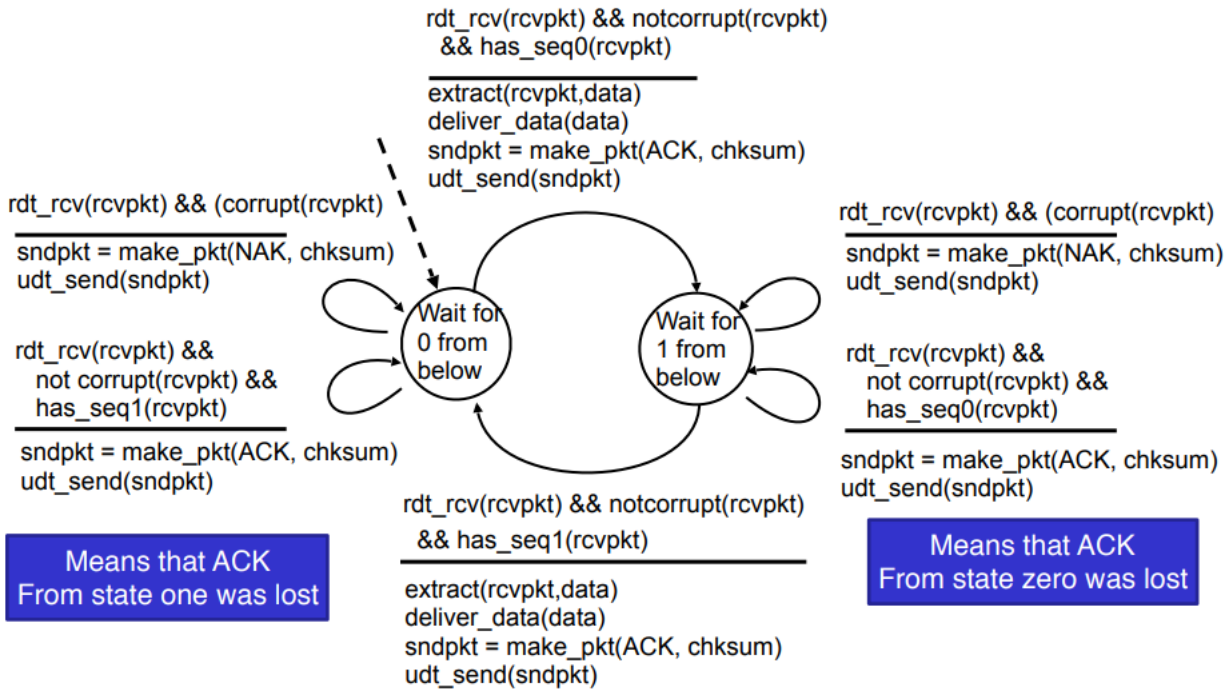

Figure 5: rdt2.1 Receiver finite state machine design

rdt2.2

One optimization we can make on the protocol is to only send ACK messages with no NAKs. In the previous version, when the receiver wanted to NAK a packet it sent NAK with the current sequence number. We can change this to instead send a duplicate ACK for the previous packet received. The sender will then use the sequence number of the response to determine if the packet was successful rather than the actually ACK/NAK value.

rdt3.0

Finally, we relax our last assumption that no packets are dropped in the unreliable channel. While sequence numbers, checksums, and ACK messages will be helpful we will need to additional functionality to potentially re-send packets. This will be accomplished with timers. If the sender does not receive an ACK from the receiver within a "reasonable" amount of time, it will attempt to re-send the packet. This may cause duplicates if the packet was just transmitted slowly and not lost, but using the sequence numbers lets us detect duplicates and ignore them.

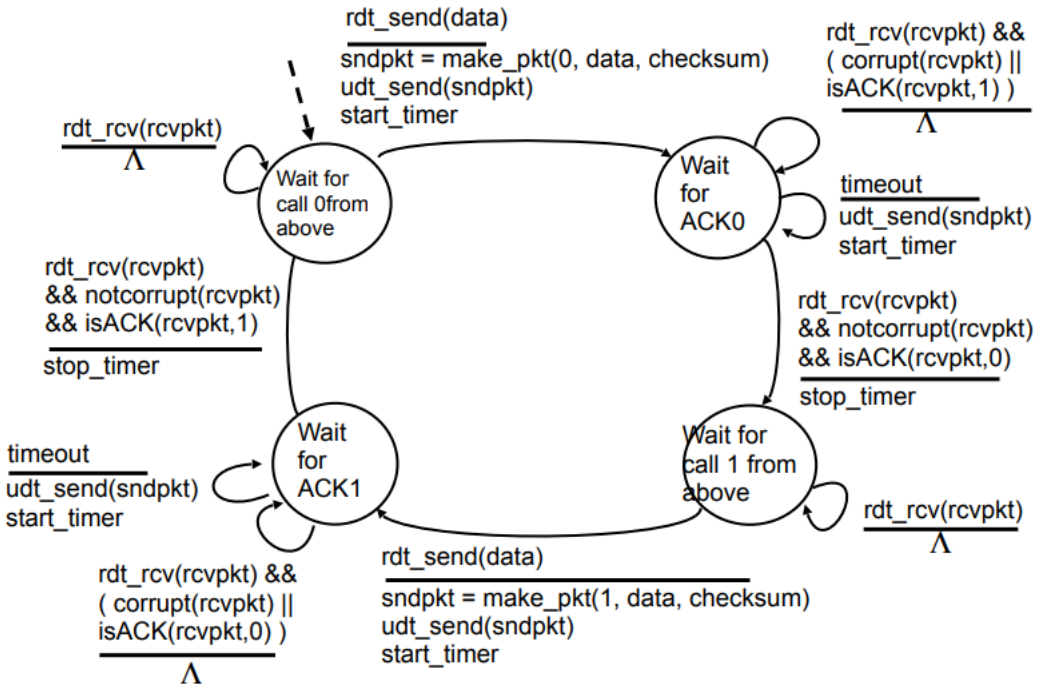

Figure 6: rdt3.0 Sender finite state machine design

rdt3.0 Performance

Having finally implemented a reliable data transfer protocol that tolerates packet loss and bit errors, we now explore the performance of our protocol. We first consider an example in which the sender transfers 8000 bits on a 1 $Gbps$ link, and the round trip time of the channel is 30 ms. We can calculate the time the sender is actually active as $D_{trans} = \frac{8000\ bits}{1\ Gbps} = 8\ \mu s$. Then, we calculate the net utilization of the sender as:

$$U_{sender} = \frac{D_{trans}}{D_{trans} + RTT} = \frac{8\ \mu s}{8\ \mu s + 30\ ms} = 0.0027$$

This means that sender is only active for less than a percentage of the time. This stems from the protocol's need to wait for a response on a packet with sequence number 0 before even attempting to send a packet with sequence number 1. To make our protocol more performant, we can try to alter it to allow for multiple packets to be *in flight* at once. The term for this sort of approach is **pipelining**, which will allow for greater throughput for our physical channel.

2.1 Go-Back-N

We would like to add pipelining to our rdt3.0 protocol so that multiple packets can be in-flight at any given time for better throughput. This will mean adjusting our sender to send multiple packets before getting responses from the receiver adjusting the receiver to be able to acknowledge various different packets potentially at once.

We will have to expand our protocol to use more than 2 sequence numbers and each ACK message from the receiver for sequence number n will now mean an acknowledgement of all sequence numbers up to and including n. Thus, if the receiver gets packets with sequence numbers 2, 3, and 5, it will send corresponding ACKs with sequence numbers 2, 3, and 3. Note that since it received a packet for 5 but not 4, the receiver doesn't acknowledge the 5 and instead re-sends the highest sequence number it can ACK.

To keep track of all the packets the sender has sent out and is waiting for responses to, we will use a technique called **windowing**. The sender will keep track of what packet's it has sent, and whether it has received ACKs for those packets. The window can be visualized as shown below.

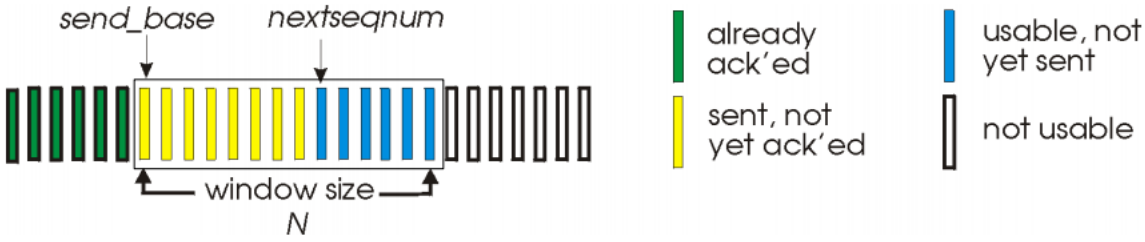

Figure 7: Windowing illustration

It's important to note that buffering is only done on the sender-side of the protocol. The receiver only has to keep track of the current expected sequence number and increment it periodically. Let's now take a look at the adjusted FSM designs for both the sender and the receiver.

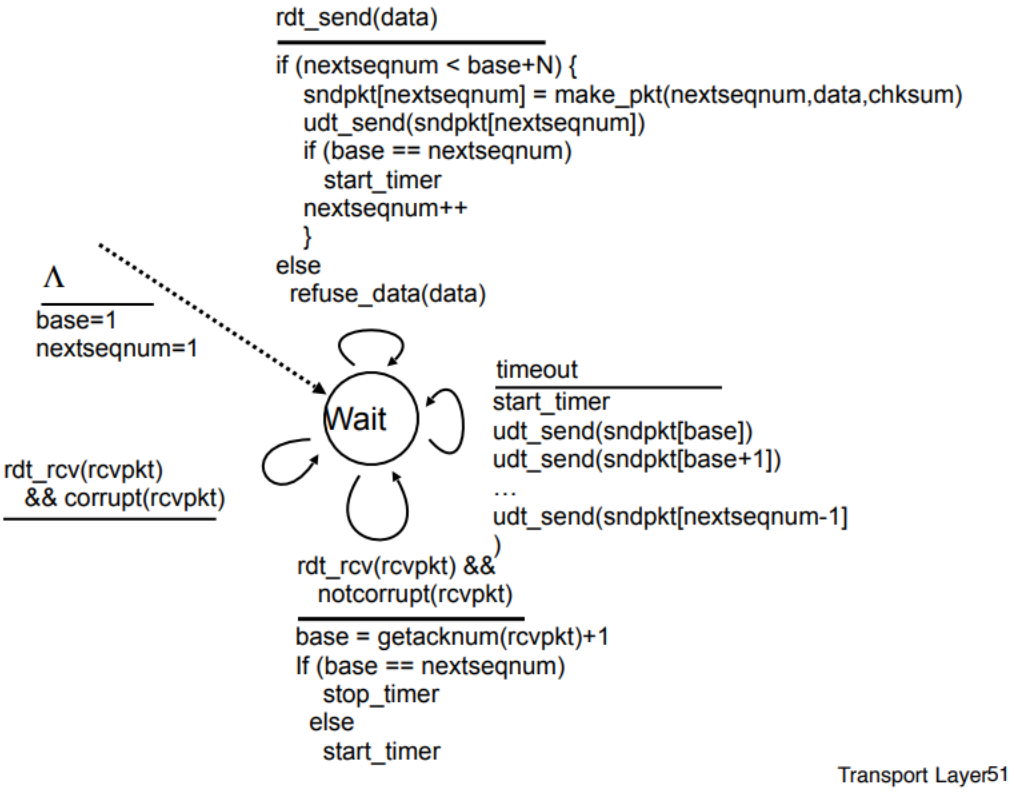

Figure 8: Go-Back-N sender FSM design

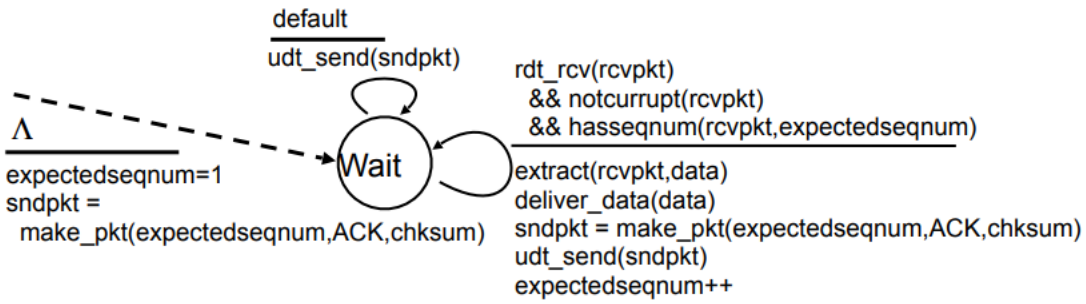

Figure 9: Go-Back-N receiver FSM design