Programming Assignment 2

COEN 233 Computer Networks

- Projects are individual. Discussions are ok, but each student should have his/her own code.
- Each project requires a demo, during which the student should explain how the code works. Demos are part of the grade. You only get full credit if you demo.
- Besides the demo, each student should submit the code.
- If you do the project following the specifications, deliver the code by the deadline, demo by the deadline, and it works properly, you get full credit for it.
- Keep the solutions simple!

Part 2: SFTP - Reliable Transfer over an Unreliable Channel with Bit Errors that can also loose packets

This project is a continuation of the 1st programming assignment. This means you are still building a SFTP (Simple File Transfer Protocol) to transfer a **file** from the client to the server. You should make the changes and add the new requirements on top of your previous programming assignment. Remember, all the previous requirements as client arguments and chunk sizes still apply.

Now that we learned how to work with sockets and TCP, we will apply this knowledge to build a SFTP over UDP. UDP is a Transport Layer protocol that it is unreliable. This means that bit errors can occur during the transmission and packets can be lost. Because UDP will not automatically correct these errors, you need a protocol to check if the data has a bit error using a checksum and retransmit to the packet. Also the protocol will use sequence numbers to detect and retransmit them while avoiding duplicates. The protocol we will use is called Stop and Wait protocol.

The project consists of building a Stop and Wait (S&W) reliable protocol. The S&W is going to be built on top of UDP, and it is supposed to provide a reliable transport service to the SFTP application (developed in part 1, which needs to change to call your new send and receive functions). Messages are sent one at a time, and each message needs to be acknowledged when received, before a new message can be sent.

The S&W consists of a client and a server. Communication is unidirectional, i.e., data flows from the client to the server. The server starts first and waits for packets. The client starts the communication. Packets have sequence number 0 or 1. Before sending each packet, a checksum is calculated and added to the S&W header. After sending each packet, the client starts a timer (use alarm or sleep). When the timer goes off, the client tries to read a corresponding ACK answer. If the corresponding ACK is not there, or it is not the corresponding ACK (or if the checksum does not match), the packet is sent again and the timer is started again. If the corresponding ACK is there, the client changes the state and returns to the application which can now send one more packet. This means that the program blocks on writes.

The server, after receiving a packet, checks its checksum. If the packet is correct and has the right sequence number, the server sends an ACK0 or ACK1 answer (according to the sequence number) to the client, changes state accordingly, and deliver data to the application.

If the message is not correct, the server repeats the last ACK message.

The protocol should deal properly with duplicate data packets and duplicate ACK messages.

The S&W packet contains the header and the application data. No reordering is necessary, since the S&W is sending the exact data given by the application, one by one.

To verify your protocol, use the result of a random function to decide to send or skip a message, to decide to send or skip an ACK message, and to decide whether to send the right checksum or just zero. This will fake the packet error and loss of a packet effect.

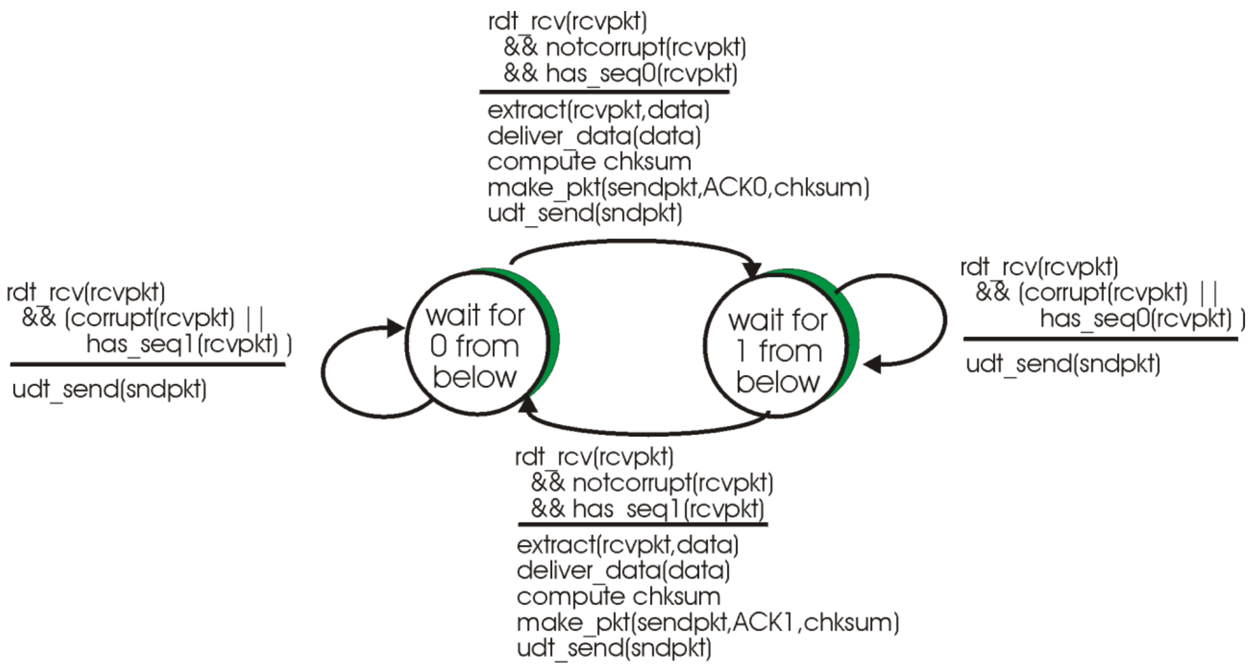

Fig. 1: Receiver

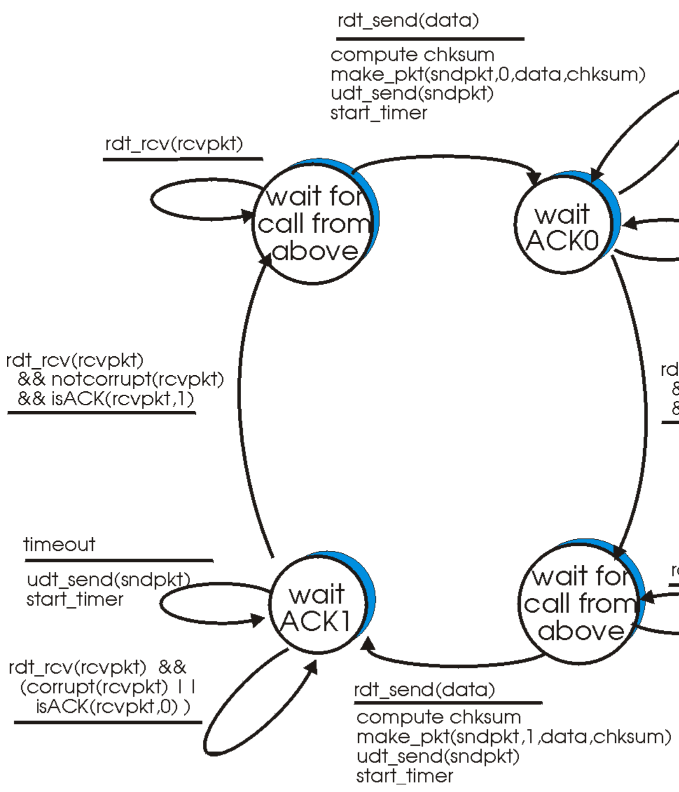
rdt_send(data)
compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
rdt_rcv(rcvpkt)
wait for call from above
wait ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
timeout
udt_send(sndpkt)
start_timer
wait ACK1
wait for call from above
rdt_rcv(rcvpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
rdt_send(data)
compute chksum
make_pkt(sndpkt,1,data,chksum)
udt_send(sndpkt)
start_timer

Fig. 2: Sender