# TDTS06: Lab 1 – Flow control and handling corrupt packets

**Juha Takkinen, Ph.D.**

**IDA, Dept. or Computer and Information Science**

## 1.0 Overview

The purpose of the whole lab series in TDTS06 is to learn how a reliable data transfer protocol is designed and implemented in an Internet-like environment, showing the important algorithms and mechanisms needed for such a protocol. You learn it by doing it.

Before you start doing the labs, however, you must set up the lab environment; see “Getting started” on the course web site.

When you have set up the lab environment, you must read up on the lab framework x-kernel, C programming (pointers and memory management) and chapter 3 in the textbook (the rdt protocol).

The x-kernel Tutorial, pp. 17–27, has a detailed walk-through of the example protocol ASP in x-kernel. Also, you might want to read Dan Everett's walk-through of the ASP protocol. You should specifically learn how messages are manipulated in x-kernel (see the Message Library in the x-kernel Programmer’s Manual as a reference). Also, look att how the test program takes an input file and sends it (“pushes” messages, in x-kernel terminology) to a lower layer protocol (the RDT protocol in this case). See the section on x-kernel documentation on the course web site, under “Lab assignments” on the menu. There is also a tutorial session for the lab series and x-kernel scheduled in the course.

Your goal in lab 1 is to develop the protocol rdt2.0 as described in the textbook (see Figure 3.10 in Chapter 3). You need to implement the mechanisms in the protocol that send a packet and also detect corrupt packets. The network will be configured as partly unreliable, which means that it will deliver all packets but some of them will be corrupt when they arrive at the receiving side.

You must also answer a number of follow-up questions concerning the lab assignment.

## 2.0 What to do first

First, you must create a file where to put the RDT protocol and the mechanisms that you will develop in the lab.

Copy the asp.c file from the asp directory to the rdt directory. Rename it to rdt.c. Then, edit the rdt.c file and replace each occurrence of asp with rdt and each occurrence of ASP with RDT (case is important). This is the normal work routine when starting to develop protocols in x-kernel.

Then, update the protocol stack so that it includes your protocol: edit the graph.comp file, which resides in the build directory (TDTS06) of your home directory, and replace each occurrence of asp with rdt, except the directory name before the slashes which should be “lab1”.

Now, you must recompile x-kernel. Issue the command:

```
make again
```

(or use **`make first`**, if this is the first time at this lab occasion; see “Getting started” on the course web site)

You also have a separate test program called rdttest, which you will be suing to test the RDT protocol. This program takes a filename as input, chops it into messages of finite length and sends them to your RDT protocol; see rdttest.c for the details. The test program is activated when you start the server and the client, respectively. First, start the server by changing to the server directory and issuing the following command:

```
xkernel -s
```

The shell window will immediately display:

```
Protocol: RDT
Time: Mon Aug 25 13:55:44 2008
```

```
Host: mina11.ida.liu.se
Participant: server
```

Then, start the client in another terminal window and from the client directory by typing:

```
xkernel -c128.1.2.3 testfile.txt
```

... and you will immediately see:

```
Protocol: RDT
Time: Mon Aug 25 13:56:10 2008
Host: mina11.ida.liu.se
Participant: client
Send file: testfile.txt
The whole file has been sent!
```

This means that the client has now sent the file testfile.txt to the server. In the server window you see the following text:

```
Open file 'foo' for writing
The whole file has been received!
Compare the file '$HOME/TDTS06/server/foo'
with the file you transmitted with diff.
The whole file has been sent!
```

You can check that the received file is the same as the sent file by issuing the following command in the client's shell window:

```
diff testfile.txt ../server/foo
```

Whenever the output from this command is empty then the received file is identical to the sent file.

This is how you run your RDT protocol, when you want to test it. You are now ready for the next step in the lab.

## 3.0 Lab requirements

The messages that the test program sends (pushes) to your RDT protocol correspond to the messages that a real Internet application layer protocol (for example FTP) pushes to the transport layer. The transport layer then creates its own packets from the messages.

### 3.1 Packet type

In order to implement flow control, you must be able to distinguish between different types of packets in your protocol. At least DATA, NAK and ACK packets will be needed in your protocol in lab 1. You may add other types when you discover that they are needed, but you are not allowed to change the semantics of DATA, NAK and ACK. The protocl header is defined in rdt_header.h.

Please note the simplification that we make in the lab series: DATA packets can only flow from the client to the server and never back, and only the server that can send NAKs and ACKs.

When the client has sent a DATA packet it must wait for an ACK. In x-kernel this menas that you will have to use a semaphore, in order to control the session thread. You need the following functions:

- semInit to initialize the semaphore. You do this once and when the session is opened.
- semWait to set the semaphore. You do this after you have sent a DATA packet, most likely in the rdtPush function.
- semSignal to release the semaphore. You do this when you have received an ACK packet, most likely in the rdtPop function.

Because both the client and the server can be using rdtPop and rdt-Push, you need to be able to detect who is currently using the code in these places in rdt.c, so that the server will not try to send DATA, for example. This can be solved in several ways. You can use the knowledge that only the server will be transmitting ACKs and only the client sends DATA, and also the fact that the communication channel is simplex. There are also functions that are only called by the client or the server but not both.

### 3.2 Checksum

Because the packets can become distorted (but not dropped) by the network, you must implement a protocol mechanism to handle corrupted packets.

Use the inCkSum function available in x-kernel to compute the checksum for the whole packet. Make sure to save the checksum in the header. Only DATA packets will carry a checksum value. Please note that a packet can be either in network byte order or host byte order; you will get different results from the inCkSum for the same packet if your byte order differs.

On the receiving side you must check that the checksum in the DATA packet is correct. This must be done when the packet arrives in rdtDemux and before rdtPop is called.

## 4.0 Testing the protocol

You can test the protocol by first starting the server and then running the client, as shown in Section 2.0 above.

Use printout statements and the x-kernel tracing library to debug your protocol.

### 4.1 Configure an unreliable network

In order to simulate an unreliable network, you must use virtual protocols in the x-kernel protocol stack. The virtual protocols of interest to us in the course are:

- VDROP, which will drop a random packet
- VDELAY, which will delay a packet 100 ms
- VDISTORT, which will distort (corrupt) a random packet.

The unreliable network is already configured in a separate file called graph.comp-unreliable_network in the build directory. You will have to edit this file and then save it as graph.comp (make a backup of the old graph.comp file first). For lab 1 you must only use the VDISTORT protocol, so change the protocol stack accordingly.

### 4.2 Use make commands

The following make commands can be used:

- **`make first`** - issue this command the first thing you do when you start work at a lab occasion; this will configure the x-kernel with the current ip address of your computer and also install the protocols listed in your protocol graph (graph.comp) and then compile and link x-kernel in order to create an executable file.
- **`make again`** - use this command subsequently, after having run **`make first`** once and the next time you want compile a new executable of xkernel. It is an alias for all of the three commands below, run in series:

  - **`make compose`** - this reads the graph.comp file and updates the internal protocol graph

- **`make depend`** - this creates dependencies between files
- **`make`** - this creates the binary version of x-kernel (file named xkernel) in your build directory.

Indeed, you can use only **`make`** if all you have done is changed the source code in rdt.c only. However, if you make changes in the graph.comp file then you must issue **`make again`**, i.e., all three commands above all over again.

### 4.3 Select a testfile

If you want to test with a larger testfile, see the English translation of The Kalevala, or why not try text from Projekt Runeberg.

## 5.0 Follow-up questions

When you have finished coding and testing the protocol in lab 1, you must answer the following questions:

1. Change the size of the messages created by the test program (see the code in rdttest.c) and test your protocol. Also try with a larger testfile. How do these expereiments affect the result (the correctness of the received file)? Discuss the results briefly. (Tip: see p. 248 in the textbook.)
2. Discuss where errors can arise in a network and what types of errors there are. (Tip: see the textbook and a relevant RFC.)
3. Is the checksum in the TCP header obligatory or not? What about the checksum in the UDP header? Explain.
4. What is the purpose of the pseudoheader in UDP and how is it related to the checksum calculation?
5. Suppose that a UDP receiver computes the Internet checksum for the received UDP segment and find that it matches the value carried in the checksum fiels. Can the receiver be absoluetly certain that no bit errors have occurred? Explain.

Explain all of your answers and list correct sources, such as RFCs

## 6.0 Demonstrating the solution

Before handing in a lab report you must first demonstrate the resulting protocol to your lab assistant. Remember to remove unnecessary trace printouts and only keep the most important ones. You are recommended to slow down the execution of your protocol, using delays, for an even clearer demonstration.

*Before* you demonstrate your solution to the lab assistant, give him/her a copy of the code on paper.

In the demonstration you must show that your protocol works according to the requirements, that is, the protocol can handle a partly unreliable channel (vdistort installed), which is shown by clear trace printouts of the packets that are corrupted and which ones are received correctly. Be prepared to answer questions on specifics in the code and your solution.

When your lab assistant has seen your demo and approved of the functionality, you can hand in a lab report containing the source code of the changed files on paper (see “Coding guidelines” on the course web site) and also the answers to all of the follow-up questions.

You must finish lab 1 before continuing with lab 2. When you have demonstrated lab 1 for your lab assistant, you can immediately continue with lab 2, while waiting for lab 1 to be passed by your lab assistant.