COSC 264
Introduction to Computer Networks and the Internet
Assignment

Andreas Willig
Dept. of Computer Science and Software Engineering
University of Canterbury
email: andreas.willig@canterbury.ac.nz

June 29, 2016

1 Administrivia

This assignment is part of the COSC 264 assessment process. It is worth 20% of the final marks. It is a programming project in which you develop applications building on the socket interface. You will work in groups of two persons, it is not an option to work alone or in larger groups, unless you have **very** convincing reasons (mere convenience on your side won't convince me) and have obtained **explicit approval** by me (normally by e-mail). Submissions by individuals or larger groups without such approval will not be marked.

You can use Python, Java, Matlab, C or C++, other languages require my permission (basic requirement is that we can read the language). Your program should be a text-mode program, no extra marks will be given for graphical interfaces.

It is quite likely that the problem description below leaves a lot of things unclear to you. Please do not hesitate to use the "Question and Answer Forum" on the Learn platform for raising and discussing any unclear issues. Please **do not** send emails with technical questions directly to me or the tutor, instead use the learn forum. This way other people can benefit from the question (and the answer).

In the following I am going to assume that you work under Linux on the command line, as I am going to mention a few tools that are available there. You are free to choose your development environment as you see fit, but please note that I have no development experience whatsoever under Windows or Mac OS X and I am not in a position to help you there.

2 Problem Description

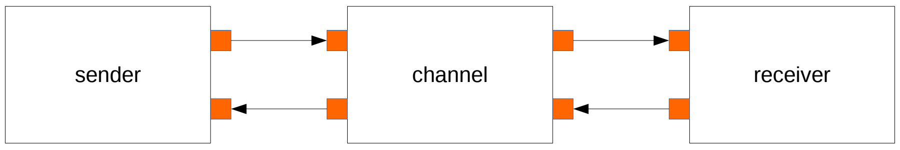

Figure 1: System Setup

You will write three different programs called `sender`, `channel` and `receiver`. All three programs should run on the same machine in different processes, and they should use UDP sockets to communicate with each other. The setup is shown in Figure 1, the sockets are indicated by the small filled (orange) boxes. Broadly, the `sender` program wants to transfer a file to the `receiver` program. Both communicate with each other using packets of a certain size and sending these through the `channel`. The `channel` will lose packets with a certain probability. Nonetheless, the `sender` and `receiver` shall (through a proper protocol which in the literature is known as the Alternating-Bit or Send-And-Wait Protocol) ensure that the file is received *completely* and *in-order*. You will furthermore be asked to perform measurements of the average number of data packet transmissions (including retransmissions) needed for data transfer for varying channel error rate.

To make the following description more concise, the sockets will be given names as follows:

- The `sender` has two sockets s_{in} and s_{out}.
- The `channel` has four sockets $c_{s,in}$, $c_{s,out}$, $c_{r,in}$ and $c_{r,out}$.
- The `receiver` has two sockets r_{in} and r_{out}.

These sockets are to be used as follows (refer to Figure 1):

- The `senders` s_{out} socket will be used to send packets to the `channels` $c_{s,in}$ socket.
- The `channels` $c_{s,out}$ socket will be used to send packets to the `senders` s_{in} socket.
- The `receiver`'s r_{out} socket will be used to send packets to the `channels` $c_{r,in}$ socket.
- The `channel`'s $c_{r,out}$ socket will be used to send packets to the `receivers` r_{in} socket.

As stated before, all the sockets are to be used as UDP sockets.

2.1 The `Packet` Type

All packets to be exchanged by the three processes via UDP sockets are of the same data type, which we will call `Packet` in the following. The `Packet` type is a structure / class / record containing five fields:

- A field called `magicno`, which for all packets in our protocol should contain the (hexadecimal) value `0x497E`. When `sender`, `channel` or `receiver` receives a packet with a different value in this field, it should print an error message and drop the packet without processing it any further.
- A field called `type`, which distinguishes the two available packet types: `dataPacket` and `acknowledgementPacket`.
- A field called `seqno`, which is an integer value. We will in fact use this field as a bit value, and restrict to the values 0 and 1.
- A field called `dataLen`, which is an integer value between 0 and 512. It signifies the number of user data bytes carried in this packet, with a maximum of 512 bytes. If this value is 0 in a packet of type `dataPacket`, we call this an empty data packet and use it to indicate the end of the transmitted file to the receiver. This value is 0 in all packets of type `acknowledgementPacket`. If the sender receives an `acknowledgementPacket` with a different value, it should drop the packet.
- A variable-length field called `data`, which contains the actual user data. The length of this field is indicated by the previous `dataLen` field.

The precise representation of this data type will depend on the programming language used. Note that if you only have ten bytes to send, then the `data` field should really only contain 10 bytes.

A declaration in C could look as follows:

```
#define PTYPE_DATA  0
#define PTYPE_ACK   1

typedef struct {
  int  magicno;
  int  type;
  int  seqno;
  int  dataLen;
  char data
} Packet;
```

In this declaration the actual `data` field occupies only one character / byte and is only meant to provide access to the *start* of the actual data (by using the address of the field as a pointer to the data buffer).

2.2 The `channel` Program

After startup, the `channel` program will read seven parameters from the command line (for this please find a way in your preferred programming language to access command line parameters). These are:

- Four port numbers to use for the four channel sockets $c_{s,in}$, $c_{s,out}$, $c_{r,in}$ and $c_{r,out}$. Each of these four numbers should be checked whether they are indeed integer numbers and whether they are in the range between 1,024 and 64,000.
- One port number for the s_{in} socket of the `sender`, to which the `channel` will send packets destined to the `sender` (using its own socket $c_{s,out}$).
- One port number for the r_{in} socket of the `receiver`, to which the `channel` will send packets destined to the `receiver` (using its own socket $c_{r,out}$).
- A floating point or double precision value P, which we interpret as the packet loss rate. This value has to satisfy $0 \leq P < 1$.

After its start, the `channel` program will read these parameters from the command line, check them and create / bind all of its four sockets. I would furthermore suggest to use `connect()` on the two sockets $c_{s,out}$ and $c_{r,out}$ and set their default receiver to the port numbers used by `sender` and `receiver` for the sockets s_{in} and r_{in}, respectively. **Important**: when you run this and the other programs, make sure that the port numbers you use are all distinct.

If all of this is successful, `channel` enters an infinite loop, in which it performs the following tasks:

- It waits for input on any one of the two sockets $c_{s,in}$ and $c_{r,in}$. For this it should use the `select()` system call. Note that `select()` in C returns a value indicating the number of sockets that have data ready. With two input sockets this number can actually be greater than one, as data can arrive on both sockets simultaneously. In that case make sure that you process **all** packets. Furthermore, make sure you use `select()` in a blocking fashion to save CPU time.
- Suppose `channel` receives a packet on socket $c_{s,in}$. It performs the following steps:
  - It checks the contents of the `magicno` field and compares it against the fixed value `0x497E`. If they are different, then processing is stopped and we go back to the start of the loop.
  - Next generate a uniformly distributed random variate between 0 and 1, which we call u. When $u < P$ then the `channel` will drop the packet (i.e. stop any further processing and go back to the start of the loop), otherwise `channel` will send the packet via its own socket $c_{r,out}$ to the `receiver` (which will get it on socket r_{in}).
- If the `channel` receives a packet on socket $c_{r,in}$ it will go through exactly the same steps, but in the end will forward the packet to socket $c_{s,out}$ (and the `sender` will get it on socket s_{in}).

In summary, the `channel` program emulates a transmission medium which can lose packets with a packet loss probability P.

Important: you will need to find out how you can generate **pseudo-random numbers** in your programming language of choice. I suggest to use pseudo-random numbers instead of "true" random numbers for reproducibility and because they are easily available in libraries. Your programming language / its libraries should at least provide facilities to generate uniformly distributed (floating-point or double-precision) random values between 0 and 1. Typically, you will have to provide your pseudo random number generator with a **seed** or initial value, which completely determines the sequence of generated numbers. For debugging purposes it is useful to use the same seed throughout, but for the "production runs" you will need to use different seeds.

2.3 The `receiver` Program

The `receiver` program takes the following parameters on the command line:

- Two port numbers to use for the two `receiver` sockets r_{in} and r_{out}. Each of these two numbers should be checked whether they are indeed integer numbers and whether they are in the range between 1,024 and 64,000.
- One port number to use for the `channel` socket $c_{r,in}$, to which the `receiver` will send packets destined to the `channel` through its own socket r_{out}.
- A file name, in which the received file will be stored.

After its start, the `receiver` program will read these parameters from the command line, check them and create / bind both of its sockets. I would furthermore suggest to use `connect()` on the r_{out} socket and set its default receiver to the port number used by `channel` for its $c_{r,in}$ socket. As a second step in its initialization, the `receiver` opens a file with the supplied filename for writing (I would suggest to abort the `receiver` program when the file already exists, just as a precaution). Finally, the `receiver` initializes a local integer variable called `expected` to the value 0.

If all of this is successful, `receiver` enters a loop, in which it performs the following tasks:

- It waits on socket r_{in} for an incoming packet. Make sure you use a blocking system call for this.
- If the received packet has a `magicno` different from `0x497E` then stop processing. Here and in the following the term "stop processing" means that `receiver` goes back to the start of the loop without any further action.
- If the packet has a `type` field different from `dataPacket`, then stop processing.
- If the `seqno` field of the received packet, `rcvd.seqno`, is different from `expected`, then:
  - Prepare an acknowledgement packet with:

    - `magicno = 0x497E`
    - `type = acknowledgementPacket`
    - `seqno = rcvd.seqno`
    - `dataLen = 0`
  - Send it via socket r_{out} to the `channel`.
  - Stop processing.

- If the `seqno` field of the received packet, `rcvd.seqno`, is equal to `expected`, then:
  - Prepare an acknowledgement packet with:
    - `magicno = 0x497E`
    - `type = acknowledgementPacket`
    - `seqno = rcvd.seqno`
    - `dataLen = 0`

    and empty `data` field.
  - Send it via socket r_{out} to the `channel`.
  - Toggle the value of `expected`, i.e. set it to `expected := 1 - expected`
  - If the received packet contains actual data (i.e. `rcvd.dataLen` > 0), then append the data to the output file and stop processing.
  - Otherwise, if the received packet contains no data (i.e. `rcvd.dataLen` $== 0$), then:
    - Close the output file and all sockets
    - Exit the program

2.4 The `sender` Program

The `sender` program takes the following parameters on the command line:

- Two port numbers to use for the two `sender` sockets s_{in} and s_{out}. Each of these two numbers should be checked whether they are indeed integer numbers and whether they are in the range between 1,024 and 64,000.
- One port number to use for the `channel` socket $c_{s,in}$, to which the `sender` will send packets destined to the `channel` (using its own s_{out} socket).
- A file name, indicating the file to send.

After its start, the `sender` program will read these parameters from the command line, check them and create / bind both of its sockets. I would furthermore suggest to use `connect()` on the s_{out} socket and set its default receiver to the port number used by `channel` for its $c_{s,in}$ socket. As a second step in its initialization, the `sender` checks whether a file with the supplied filename exists – if not, exit the `sender` program. Finally, `sender` initializes a local integer variable called `next` to the value 0, and a local boolean flag called `exitFlag` to the value `FALSE`.

If all of this is successful, `sender` enters a loop, in which it performs the following tasks:

- It attempts to read up to 512 bytes from the open file into a local buffer. Suppose that n is the number of bytes that you actually managed to read from the file.
- If $n == 0$, then prepare a data packet as follows:
  - `magicno = 0x497E`
  - `type = dataPacket`
  - `seqno = next`
  - `dataLen = 0`

  and an empty data field. Furthermore set the flag `exitFlag` to `TRUE`.
- Otherwise, if $n > 0$ then prepare a data packet as follows:
  - `magicno = 0x497E`
  - `type = dataPacket`
  - `seqno = next`
  - `dataLen =` n

  and append the n bytes of data to it.
- In both of the previous two cases, place the prepared packet into a separate buffer, which in the following will be referred to as `packetBuffer`.
- We now enter an inner loop, in which the following actions take place:
  - Send out the packet stored in `packetBuffer` to the channel using socket s_{out}.
  - Wait for a response packet on socket s_{in} for *at most one second* (this is your timeout value). You could use `select()` for this purpose.
  - If no response packet occurs, go back to the start of the inner loop (so you re-transmit the contents of `packetBuffer`).
  - If a response packet is received (we denote it as `rcvd`) then:
    * Check whether `rcvd.magicno == 0x497E`, whether `rcvd.type == acknowledgementPacket` and whether `rcvd.dataLen == 0`. If any of these checks fails, then go back to the start of the inner loop (and re-transmit `packetBuffer`).
    * If `rcvd.seqno` differs from `next`, then go back to the start of the inner loop (and re-transmit `packetBuffer`).
    * Otherwise, if `rcvd.seqno` equals `next`, then perform the following steps:

  · toggle `next`, i.e. set `next := 1 - next`
  · If `exitFlag == TRUE` then close the file and exit the `sender` program.
  · Otherwise, if `exitFlag == FALSE` then go back to the beginning of the *outer loop* (i.e. read the next block of data from disk and try to transmit that).

In addition to that, please add code by which the `sender` can count how many packets it has sent in total over the s_{out} socket and print this number when the program exits.

In all programs, when you are asked to "exit the program" make sure that you properly return all the resources to the operating system, e.g. by calling `close()` on any open sockets or files at that time.

2.5 A Few Hints

- I suggest to assume that all three programs will always run on the same computer (IP address `127.0.0.1`, also referred to as the **loopback address**), so that it is not necessary to ask for IP addresses on the command line.
- Watch your memory and CPU usage. Under Linux you can use `ps` and `top` for that, and if you observe that any of your programs takes about 100% CPU time all the time, then you have used a non-blocking call somewhere, which you should avoid.
- There is an incredible amount of tutorial and reference information about socket programming available in the web (including for specific programming languages), use that.
- For debugging purposes it might be useful to have the three programs print out all packets they receive. If you happen to work on your own Linux computer and have `root` access to it, I recommend that you install the `tshark` and `wireshark` programs. The command line incantation `tshark -V -i lo` prints very detailed information about every packet that is sent via the loopback interface.
- To check whether transmitted and received file are really the same you can use the `md5sum` tool. If it is a text file, you can also use the `diff` tool.
- When testing, make sure that all port numbers you use are really different.

3 Deliverables

Each student pair has to submit **a single pdf/a file** (a pdf file with all fonts embeded, other formats will not be accepted) which includes the following items:

- The names and student-id's of both group members. If you have received approval to work as an individual, then please state this and also give the date on which you have received this approval from me.

- You need to give an **agreed-upon** percentage contribution for each partner, reflecting how much work either partner has spent. The relative weights will influence marking.
- Responses to **all** questions given below.
- A listing of your source code. We would appreciate if you use a pretty printer.

The pdf file has to be submitted to the departmental coursework dropbox, see `http://cdb.cosc.canterbury.ac.nz`. Please submit no later than **Friday, September 9, 2016, 11.59pm**. Late submissions are **not** accepted. Note that you can submit several versions of the pdf file to CDB, you do not have to wait until the last minute. We will mark the latest submitted version only.

4 Questions and Marking

Each pair of students should submit answers to the following questions:

1. The protocol between `sender` and `receiver` as described above has (at least) one weakness: it has a **deadlock**. Please explain the notion of a deadlock in the context of networking protocols and describe the particular deadlock situation in our case. A guiding question is: what can go wrong and when in case certain packets are lost?
2. What is the `magicno` field good for?
3. Please explain what the `select()` function is doing and why it is useful for the `channel` (and in another way for the `sender`).
4. Please explain how you have checked whether or not the file was transferred correctly (i.e. the receivers copy is identical to the transmitters copy).
5. We consider different packet loss probabilities of $P \in \{0.0, 0.01, 0.05, 0.1, 0.2, 0.3\}$ and a source file of length $M = 512 * 100 = 51,200$ bytes (you need to create such a file). For each value of P make ten repetitions of the file transfer and for each repetition record how many packets the `sender` has sent in total. Draw a graph that shows the different values of P on the x-axis and for each such value the *average* number of total packets (the average being taken over the ten repetitions) on the y-axis. Explain the results.

   Note: To produce graphs, the tool `gnuplot` can be useful under Linux. Its main advantage is that it allows for script-based (i.e. non-interactive) creation of graphs, but admittedly its command syntax needs some getting used to. However, you are free to use any tool you like (including Excel, Matlab, etc.) for producing graphs. **Under all circumstances you need to make sure that axes and curves are properly labeled**. You will lose marks otherwise.
6. Assume the following:

- The probability to loose an individual packet (either a `dataPacket` or an `acknowledgementPacket`) is P,
- Packet loss events are statistically independent of each other.
- The size of the file to be transmitted requires N packets.

Please derive and justify an expression for the average total number of packets that need to be sent (including retransmissions) to transmit the entire file. Compare this to the (average) total number of packets you have observed in your experiments.

Marking will be based on these answers and the source code. We will not mark the source code for style (only really ugly or messy source code will get deductions) but for its ability to produce the right results. We will also mark it for the amount of error checking you do – if we find that you use system-/socket calls without any error checking we will apply deductions. We will also check whether you have returned all resources (sockets, files) to the operating system.