IN2140 Home Exam in spring 2025

The IN2140 home exam is meant to be solved in groups. Due to limitations of the Inspera delivery system, everyone in the group must deliver the solution. Since the solution is meant to be anonymous, we cannot use the Google Doc to match your submissions for you.

We therefore ask you to include a GROUP.txt file into your submission. This GROUP.txt file should contain one candidate number per line, and nothing else.

We will not be able to grade group members who haven't delivered their copy of the group's work. They will fail.

Those are the instructions we received. We are sorry about this complication.

The Labyrinth

Introduction

In this assignment, you implement a networked system that emulates a few communication tasks that the various layers in networked systems must solve. You will implement one side of a link layer (L2) emulation, one side of a transport layer (L4) emulation, and the client of an application layer (L5) system. The latter will be used to contact a server to request a labyrinth, which you should solve, and then return the solution to the server.

On the link layer (L2), we focus on the task of sending frames between neighbouring computers. This layer defines the size limit of frames, refusing to receive data for transmission that exceeds the size of the frame after adding an L2 header. It also adds a checksum field, which covers all bytes of the frame except for the checksum field itself. When a frame arrives at the remote L2 entity and its checksum field is incorrect, the frame is discarded. If the checksum is correct, the payload of the frame is passed to the layer above.

We drop the network layer (L3) in this home exam and work only with computers that are direct neighbours. In practice, we emulate neighbourship on L2 using IPv4 addresses and ports, which means that every pair of computers can be considered direct neighbours. As a consequence, L4 uses L2 functions directly.

On the transport layer (L4), we focus on the task of sending data between two computers in terms of packets. Packets cannot be larger than the payload of L2 frames, but instead of discarding data that is too large, the L4 send function truncates the data that it is requested to send by an application. An L4 entity that we create for this home exam is very restricted: it can only communicate between two processes. When a different process (IPv4 address or port) interferes, the behaviour of the L4 entity is undefined. The main feature of this very simple transport layer is that it implements the stop-and-wait protocol between two peer entities.

On top of this, at L5, we implement a labyrinth generation and solver application. The L5 client, which you write, requests a maze from the given L5 server, as well as a start and end coordinate at the top center and bottom center of the labyrinth. Your client finds a path through this labyrinth and returns the solution to the server. Having done this, the client terminates the communication with the server and terminates.

Steps

In this assignment, the higher layers depend on each other. That means that it is important to make L2 work before L4 and L4 before L5. We provide separate binary servers for testing L2 (datalink-text-server), L4 (transport-test-server) and L5 (maze-server), as well as source code for the appropriate clients (datalink-text-client, transport-test-client, maze-client).

Importantly, you don't have to implement each of the layers perfectly before working on the next layer above it. By starting the server with the parameter "-p 0.0", you prevent the server from inserting errors into the checksums of L2 frames on sending or receiving. This makes it possible for you to start with a very simple implementation (the L2 frame headers and L4 packet headers must be correct, but packets will not be dropped or retransmitted).

Sensible steps may be:

(1) Implement frame sending and receiving with correct headers and correct checksum computation in L2. This means implementing initial versions of l2sap_create, l2sap_destroy, l2sap_sendto and l2sap_recvfrom_timeout. If you choose "-p 0.0" for the server, no frames get lost, and you don't have to implement a timeout for l2sap_recvfrom_timeout.

(2) Implement packet sending and receiving of packets in L4. You must implement the types, sequence number and acknowledgement number in the packet headers according to stop-and-wait. If you choose "-p 0.0", you don't have to handle any timeout and retransmission, and you don't have to do any work that enables full-duplex behaviour. Without frame drops, sends and receives will not overlap, and there won't be timeouts. You implement l4sap_create, l4sap_send, l4sap_recv and l4sap_destroy.

At this point, you can either introduce frame loss at L2 and a stop-and-wait implementation with timeout and retransmission at L4; or you can implement the labyrinth solution next over L2 and L4 implementations that do not support frame loss.

Finally, you should work on the missing part.

The L2 layer

The L2 entity that you implement should emulate the link layer by sending UDP packets to the L2 peer entity. Such a UDP packet emulates your L2 frame. The frame (the 2-PDU) should not be longer than 1024 bytes including the frame header.
The frame header has the following structure:

```
struct L2Header
{
    uint32_t dst_addr;
    uint16_t len;
    uint8_t  checksum;
    uint8_t  mbz;
};
```

The details of these fields can be found in l2sap.h. It is important to keep in mind that all of these fields are sent over the network in network byte order. Since the L2Header has a length of 8 bytes, it is also important to remember that the payload (the 2-SDU) cannot be longer than 1016 bytes.

If a higher layer attempts to send payloads larger than the payload size, L2 must discard them.

If an L2 entity receives a frame (emulated UDP packet) that contains the wrong checksum, the frame must be discarded and not passed to the higher layer. The checksum is computed by a byte-wise XOR operation of the `len` bytes of the frame.

The `dst_addr` field must contain the same IPv4 address (in network byte order) that you use in the `sockaddr_in` structure that you use for sending the UDP packet that emulates the L2 frame.

`mbz` means “must be zero”.

The L3 layer

We don’t implement the L3 layer. We have not covered routing in the lecture at this point, meaning that an L3 implementation makes no sense.

Actually, if you wish, you may add an L3 layer for style that does nothing but pass through from L4 to L3. Its header must be empty and so on.

The L4 layer

The L4 layer implements flow control and error control using the classical stop-and-wait protocol. It does not implement congestion control, it does not segment packets, it provides no security, it does not establish and tear down connections between peer L4 entities.

Each L4 entity is capable of communicating with exactly one peer L4 entity, and has to terminate when these 2 entities are no longer communicating with each other. It is not possible to reset the state of an L4 entity, which would allow it to start communicating with another peer L4 entity. If another L4 entity interferes with the communication between two peer entities, the result is unpredictable.

Its header is very simple:

```
struct L4Header
{
    uint8_t type;
    uint8_t seqno;
    uint8_t ackno;
    uint8_t mbz;
};
```

It is thus 4 bytes long, which means that the L4 payload (4-SDU) can only have 1012 bytes.

The `type` of the L4 packet can be either `L4_DATA`, `L4_ACK` or `L4_RESET`. The values are defined in l4sap.h. The meaning of the first two is obvious, but `L4_RESET` tells the peer entity that an entity is shutting down and that the peer entity should be the same. This message can be sent several times.

`seqno` is either 0 or 1 and indicates the sequence number of a data packet according to stop-and-wait. The initial value is 0.

`ackno` is either 0 or 1 and indicates that acknowledgment number sent in response to a data packet that has been received, according to the stop-and-wait protocol.

`mbz` must be zero.

The L5 layer: the labyrinth

The application performs exactly 4 steps before shutting down:

- The client sends a text message containing "`MAZE <random number>`".
- The client waits for a response. This response is a serialized maze data structure as described below.
- After solving the maze, the client sends the maze back to the server.
- The client sends the text message `QUIT` to the server and shuts down the L4 entity.

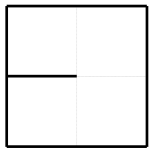
The maze packet has the following format:

```
struct Maze
{
    uint32_t edgeLen;
    uint32_t size;
    uint32_t startX;
    uint32_t startY;
    uint32_t endX;
    uint32_t endY;
    char     maze[];
}
```

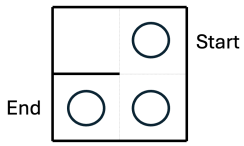
All header values are in network byte order. `edgeLen` indicates how wide and high a maze is. `size` is `edgeLen*edgeLen` for convenience. `startX` and `startY` are the maze coordinates where the client must start searching for a path through the labyrinth, `endX` and `endY` is the position that the search must reach. `maze` is a sequence of bytes that indicate in which direction each cell of a labyrinth is open. The possible values are `left`, `right`, `up`, `down`, and every combination of these. These are bit values that can be found in `maze.h`.

The bytes in the maze are stored as a sequence of rows without separators, so if the maze is a 2x2 maze, the variable maze would contain 4 bytes, the first byte is the top left, the second top right, the third bottom left, the last bottom right. The maze that we use are all going to be smaller than 1012, so will fit into a single L4 packet.

A 2x2 maze with the values
`{ right, left | down, right, left | up }`
would represent the following maze:



The client that receives this maze must solve the path from startX, startY to endX, endY. If `startX==1, startY==0` and `endX==0, endY==1`, the solution would be
`{ right, left | down | mark, right | mark, left | up | mark }`
and look like this:



The client serializes the maze after marking the solution to `mark` bits and sends it back to the server.

The Precode

The 3 main files where you implement the L2, L4 and maze-solving functionality:

- l2sap.c
- l4sap.c
- maze.c

You should implement the dummy functions in these files. We encourage you to add support functions. That will make the task of implementing the required functionality very much easier.

You can also create additional header and code files, but you must of course edit CMakeLists.txt to include them.

Their header files are the next 3. You should not change maze.h. You can extend the L2 entity information in the `struct L2SAP` in l2sap.h, but it is probably not required. You must extend the L4 entity information in the `struct L4SAP` in l4sap.h. You should not modify the frame header `struct L2Header` or the packet header `struct L4Header`.

You must not change the parameters or return values of the L2 or L4 functions, because our test code requires that they are exactly as handed out.

- l2sap.h
- l4sap.h
- maze.h

The three client files are meant to test the L2 functionality against our server datalink-test-server, the L4 functionality against our server transport-test-server, and maze-client.c is the client that requests the maze from our maze-server. The files maze-plot.c provides plot of a maze, where 'X's walls and 'o's are marks.

- datalink-test-client.c
- transport-test-client.c
- maze-client.c
- Maze-plot.c

Finally, there is the CMake file that helps creates Makefiles for compiling your code.

- CMakeLists.txt

Advice

Updates

This is a new assignment not similar to any earlier one. It is therefore likely that some bug reports are going to come in and the precode will require improvements. Take regular looks at the Messages on the course page https://www.uio.no/studier/emner/matnat/ifi/IN2140/v25/ .

The up-to-date version of the precode, these documents and the binary servers are available from Github here: https://github.uio.no/IN2140v2/in2140-v25-he

Using valgrind

To check the program for memory leaks, we recommend that you run Valgrind with the following flag:

```
valgrind \
    --leak-check=full --track-origins=yes \
    DITT_PROGRAM
```

Submission

To submit your code, you must use Inspera as mentioned above.

To create your zip files, use the command “make package_source” in your build directory. That creates a file HomeExam-1.0-Source.zip containing your source code, CMakeLists.txt and .txt files.

Make sure that the directory with your source code includes GROUP.txt and also README.txt if you explain parts of your solution.

About the Evaluation

The home exam will be evaluated on the computers of the login.ifi.uio.no pool. The programs must compile and run on these computers. There will be ambiguities in the assignment text. You may point them out in comments in the code and in your README.txt. Write about your choices and assumptions in your README.txt that is delivered alongside the code.

It is possible to pass the exam without solving the task entirely. An A will require a mostly complete and very good, but not a perfect solution. In fact, we consider it very hard to solve the task perfectly. Especially an implementation of a full-duplex solution at L4 and testing it with transport-test-server and a packet loss value>0 will be a challenge (and don’t even try loss probabilities higher than 0.1).

We will test your solution with the programs that we hand out, but it is very important for you to know that we will also test the implementations of your functions *separately* using small test programs. It is essential that you do not change the name, parameter or return value of the functions, because each change will cause some tests to fail.

We expect **at minimum** that the L2 tests work without any flaws (no memory leaks etc), and that the function mazeSolve() creates a perfect path from start to end (no detours, no dead ends, connecting start and end).

To clarify, the minimum does not require mazeSolve() to work in the program maze-client.c. We will link it with a non-networked test program and it is sufficient if it solves every solvable 31x31 maze that we use to call it.