Lab #2 - Transport Protocol

Karl-Johan Grinnemo
karl-johan.grinnemo@kau.se

Karlstad University — January 10, 2024

Introduction

This lab introduces students to the basic features of reliable transport protocols, such as the concept of a Finite State Machine (FSM) and recovery of lost packets.

Examination

The lab is graded as *pass* or *failed*. To pass, the student or students should demonstrate the lab to one of the lab assistants in the course and submit the state machine and source code for their transport protocol according to instructions on Canvas.

Preparations

- Review Section 3.4, *Principles of Reliable Data Transfer* (especially Section 3.4.1, *Building a Reliable Data Transfer Protocol*) in the textbook.[1]
- Download the source code to the network emulator from Canvas (`Lab_2_Source_Code.zip`) and uncompress the zip file in the folder where you intend to develop your transport protocol.
- Familiarize yourself with the code and make a test compilation and a test run:

Command Line

```
>  make
> ./sim 20 0.2 0.2 10 2
-----  Stop and Wait Network Simulator Version 1.1 --------

The number of messages to simulate: 20
Packet loss probability: 0.200000
...
Event time: 18.705740,  Type: 1, From layer 5 , Entity: 0
...
Event time: 180.325134,  Type: 1, From layer 5 , Entity: 0

Simulator terminated at time 189.818970 after sending ...
```

[1] J. F. Kurose och K. W. Ross. *Computer Networking - A Top-Down Approach*, 7th edition.

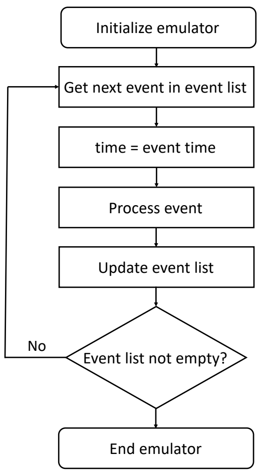

Figure 1: The event loop of the discrete-event network emulator.

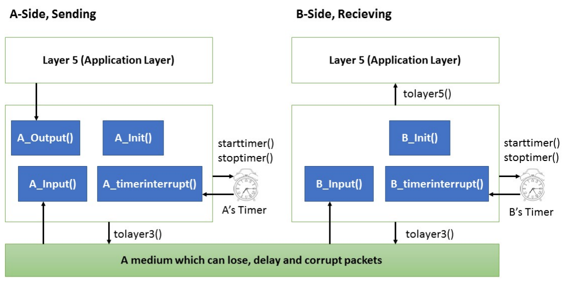

Figure 2: The architecture of the discrete-event network emulator.

In the shown test run, host A sends 20 packets (first argument) with an intensity of 1 packet every 10 time units (fourth argument) to host B with a packet-loss probability of 20% (second argument), a packet-corruption probability of 20% (third argument). The trace level is two (the fifth and last argument). Note that since we only can have one outstanding packet at host A, there has to be a time interval longer than the round-trip delay between packets being sent.

Description

In this lab, you are asked to implement a *stop-and-wait* (*alternating bit*) transport protocol (layer-4 protocol) that works similarly to `rdt3.0` in the textbook. The provided source code in `Lab_2_Source_Code.zip` implements a so-called discrete-event emulator that emulates a scenario in which a host A sends messages to a host B, which acknowledges them and prints out successfully received messages on the terminal. The emulator emulates a unidirectional, reliable data transfer from host A to host B.

Figure 1 illustrates the main principle behind the discrete-event network emulator – the event loop[2]. The emulator models the operation of an emulated network scenario in discrete events of the time. Each event, for example, the transmission of a packet, a timeout, or a reception of a packet, is scheduled to be executed at a specific time in the future and is inserted in time order in the so-called *event list*. During an emulation run, packets sent by a host are generated, and events are scheduled for transmitting these packets at specific points in time, that is, inserted in the event list. An emulation run continues for as long as events are in the event list and does not terminate until the event list is empty. The architecture of the network emulator is depicted in Figure 2 and found in the source file, `Sim_Engine.c`. Before the event loop starts, the function `init()` is called, and the emulator is initialized, which, among other

[2]You find the event loop in source file `Sim_Engine.c`.

things, involves the generation and scheduling of the transmission of the first message[3]. Also, the functions `A_init()` and `B_init()` are invoked before the start of the event loop. The event loop handles three types of events: generation of packets by the application (`FROM_LAYER5`), reception of packets in layer 3 (network layer) from the underlying medium (`FROM_LAYER3`), and timer interupts (`TIMER_INTERRUPT`). The event `FROM_LAYER5` results in either of the functions `A_output()` or `B_output()` being called; if the event is generated by host A then `A_output()` is called, and if it is generated by host B then `B_output()` is called. The event `FROM_LAYER3` results in either of the functions `A_input()` or `B_input()` being called; packets received by host A results in function `A_input()` being called, while packets received by host B results in function `B_input()` being called. The emulator supports two retransmission timers: one on host A and one on host B. A retransmission timer is started by invoking the function `starttimer()` and stopped by invoking the function `stoptimer()`. When a retransmission timer expires, either of the functions `A_timerinterrupt()` or `B_timerinterrupt()` is called, depending on whether the timer was set by host A or host B.

Although all core parts of the emulator have already been implemented, it remains for you to implement six functions:

1. `A_init()` where all initialization of the transport protocol on host A is carried out.
2. `A_output(struct msg message)` where `message` is a structure containing data to be sent to host B. It is the job of your protocol to ensure that the data in such a message is delivered in order and correctly to the application layer of host B. Sometimes, packets generated by the application are received in a state that prohibits immediate transmission. It is up to you to decide how to handle this situation. A superior approach would be to have a send queue and enqueue these packets, but we also allow for the less elegant solution of dropping them altogether.
3. `A_input(struct pkt packet)` where `packet` is a packet received from layer 3. Remember that received packets may be corrupted and have the wrong checksum.
4. `A_timerinterrupt()` which, as previously mentioned, is invoked when the retransmission timer on host A goes off.
5. `B_input(struct pkt packet)` which is the same function as `A_input()` but on host B.
6. `B_init()` where all initialization of the transport protocol on host B is carried out.

Note that you may obtain more debug information by setting the macro `TRACE` to a value greater than 1 (the fifth argument to the simulator).

End of Lab

[3]Messages are the same as packets generated at layer 5, the application layer.