FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Electrical Engineering Department

Telecommunication Systems

2012/2013

Laboratory Work 2:
Data link layer protocols with sliding window

Integrated Master in Electrical Engineering and Computers

http://tele1.dee.fct.unl.pt

Index

1. Objective

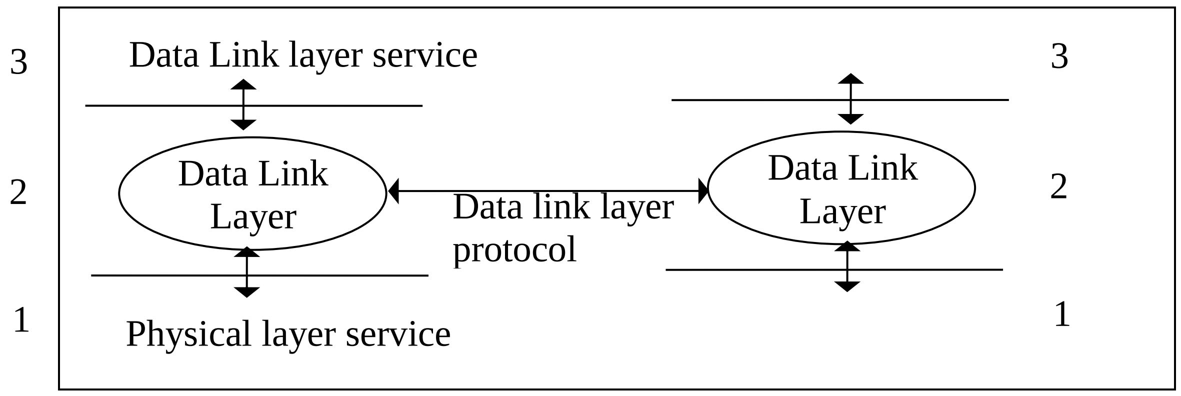
Familiarization with the sliding window data-link layer protocols, of type Stop & Wait, Go-Back-N.

The work consists in development of a data-link layer protocol based on sliding window, of type Go-Back-N with Nack, in a phased manner. For this, it is provided a system simulator developed in the course using TCP sockets, which simulates the operation of the network protocol level and physical level.

> **Suggestions**: In certain parts of this document appears some text formatted differently that begins with the word "Suggestion". It is not mandatory to follow what is written there, but may be important for students or groups where there is not yet at ease with programming, data structures and algorithms.

2. Specifications

The aim is to develop a data-link layer protocol for a point-to-point physical connection, which interacts with the Network layer and Physical layer protocols, respectively through the interfaces of the services of the Logical and Physical layers. Thus, the system represented below will be simulated through the set of primitives of the two services.

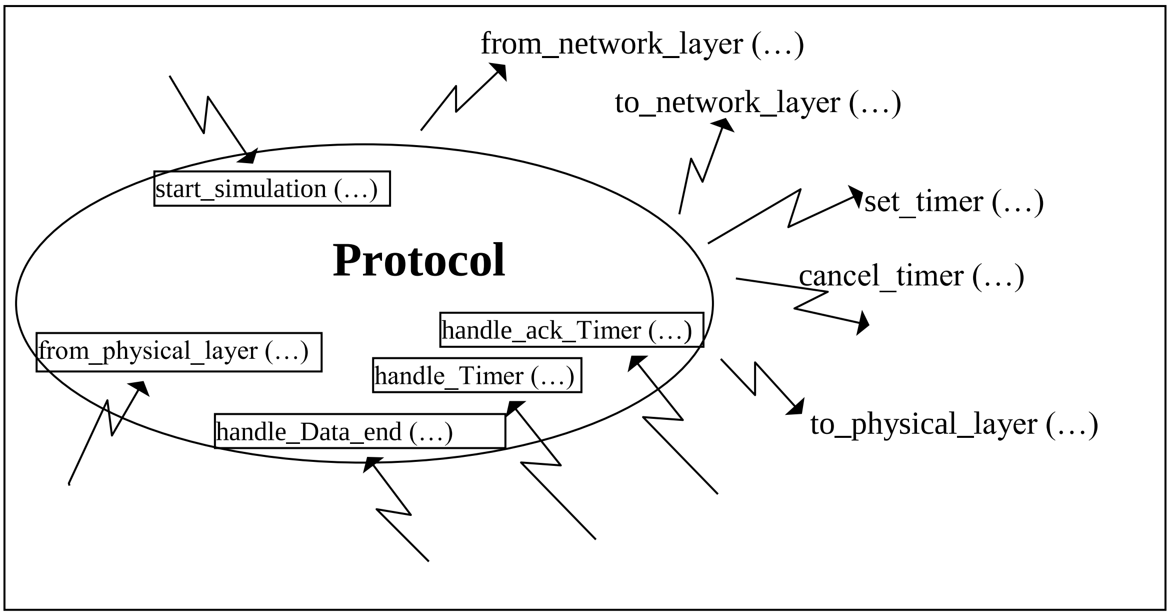


The proposed approach was inspired in the simulator described in sections 3.3 and 3.4 of the recommended book (Computer Networks 5th edition), although it makes the simulation a bit closer to reality (e.g. it considers the transmission time of data frames). The data-link layer protocol will react to events and can invoke commands.

The Data Link layer begins with the event that is served by the method `start_simulation()`. To receive packets (strings) from the Network layer, the data-link layer uses the method `from_network_layer()` and sends packets (strings) to the Network layer invoking the method `to_network_layer()`. The data must be delivered in the same order they were received, recovering from channel errors at the Physical layer.

The protocol can send frames to the Physical layer using `to_physical_layer()` and it can receive frames from the Physical layer in its method `from_physical_layer()`, invoked by the Physical layer protocol.

Finally, you can use a set of support features to manage timers. Using the methods `set_timer` and `cancel_timer`, you can set or cancel a timer identified by a number greater than or equal to zero (called *key*). When the time expires, the method `handle_timer()` is called. The figure below illustrates what was just described. Students should implement the balloon called "Protocol".



2.1. Frames types

The protocol can send or receive three kinds of frames: DATA, ACK or NACK.

<u>Data frames (DATA)</u>

The data frames have the following fields:

- Sequence number (`seq`)
- Acknowledgment (`ack`)
- Information (`info`)

The frames are numbered with a `seq` number between 0 and a maximum number specified in a window (`sim.get_max_sequence()`). The `ack` field contains the sequence number of the last data frame received.

The data frames have a non-null transmission time, so its transmission is carried out in two phases:

1) It starts sending the frame using the method `to_physical_layer`;

2) It is received an event `handle_Data_end`, stating that ended the sending of the data frame.

Other frames can not be sent between the beginning of the sending of a data frame and the reception of the end event.

<u>Acknowledgment frames (ACK)</u>

The acknowledgement frames (ACK) are generated after the reception of data frames, indicating the sequence number of the last data frame successfully received. The ACK frame is considered instantaneous, being sent on a single method invocation to `to_physical_layer`. The ACK frame contains a single field:

- Acknowledgment (`ack`)

As it is more efficient to send this information via data frames (for piggybacking), an auxiliary timer was set (*ack_timer*) that can be used to wait for a data frame, only sending ACK after this time. Thus, after receiving a data frame you should:

1) Start the ACK timer using the method start_ack_timer();
2a) If a data frame to transmit arrives, the timer can be canceled using the method `cancel_ack_timer();`
2b) If the timer expires, the event `handle_ack_Timer` is generated, which should send the ACK frame.

Negative acknowledgement frames (NACK)

The negative acknowledgment frames (NACK) can be generated in sliding window protocols due to incoming data frames out of order (e.g. it is waiting for the frame 0 and receives the 1) as a result of having jumped some sequence number. It is also considered an instantaneous frame and only has one field:

- Acknowledgement (`ack`), with the sequence number of a data frame to be retransmitted.

The receiver of this frame should retransmit the data frame requested as soon as possible. To prevent the send from continuously retransmiting the same frame, the NACK may be generated only once for each sequence number.

2.2. Data Link layer protocols

In sections 3.3 and 3.4, the recommended book describes the three basic types of data link layer protocols that will be made in this work. This section briefly reproduces the fundamental aspects of each, but we recommend a careful reading of the book for proper completion of the work.

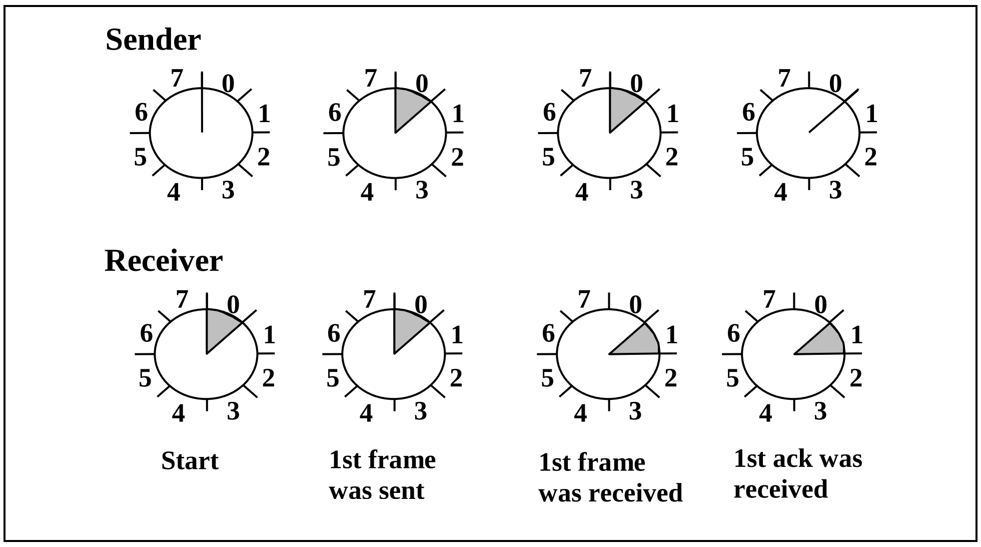
Utopian protocol

The Utopian protocol is described in section 3.3.1 of the book, corresponding to realize the transmission of frames sequentially without any error recovery mechanism. The receiver is limited to receive the frames and send them to the network level.

It is provided the complete code of a bidirectional version of this protocol with the work assignment.

Stop & Wait protocol

The Stop & Wait protocol is described in section 3.4.1 of the book and corresponds to a sliding window protocol with unitary transmitting and receiving windows. The sender holds the next sequence number transmitted and the receiver the next expected sequence number, in accordance with the diagram to the right.

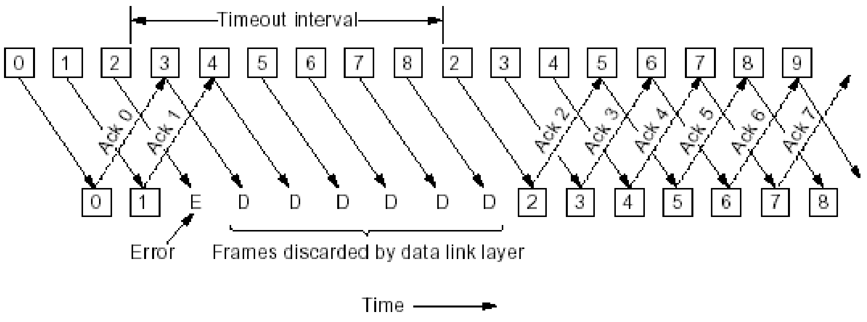


The sender shall set a timer each time it sends a data frame. If it expires, you should resubmit the plot. When an ACK (or data frame) confirm the frame, you should cancel the timer and send the next frame.

<u>Go-Back-N protocol</u>

The Go-Back-N protocol is described in section 3.4.2 of the book. It corresponds to a sliding window protocol with a unitary receiving window, which uses pipelining to improve performance when the bandwidth * delay product is high.

In this case, the sender will need an array to maintain the transmit buffers, and may transmit up to `sim.get_send_window()` frames without receiving acknowledgments (i.e. transmission window). When an error occurs, it has to retransmit all data frames from the one that has not been confirmed, as shown in the figure below.



The management of timers is somewhat more complex in this protocol. The ideal is to maintain a timer for each individual pending frame. You can also use a single timer on the older unconfirmed frame at a given time. In the latter case, the timer would be reset each time it is confirmed a new data frame.

Sugestions: As the management of various timers can make the program slightly more complex, it is suggested that the implementation is carried out in two phases: i) the first can use only one timer, ii) the second should use a timer for each pending data frame.

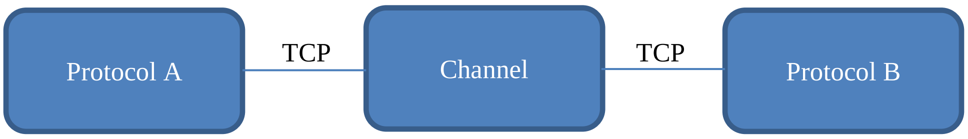
<u>Go-Back-N with NACK protocol</u>

The Go-Back-N with NACK protocol is not directly described in the book. It adds the NACK frame to the Go-Back-N protocol, as is described in section 3.4.3 of the book, introduced in the context of a selective repeat protocol. In Go-Back-N is somewhat simpler, because it will be necessary to transmit everything starting from `seq`. So it accelerates the response to the loss of frames compared to waiting for the timer expiration.

2.3. Simulation Scenario

In this work, a network with variable delay time and a constant average frame loss rate is simulated. Two programs are used:

- *Protocol* – implements the data link layer protocol and emulates the network level, controlling the sending and receiving of data packets numbered sequentially. The data link layer part will be made by students;
- *Channel* – connects two instances of *Protocol*, emulating the propagation time and missing frames with a certain probability of loss.

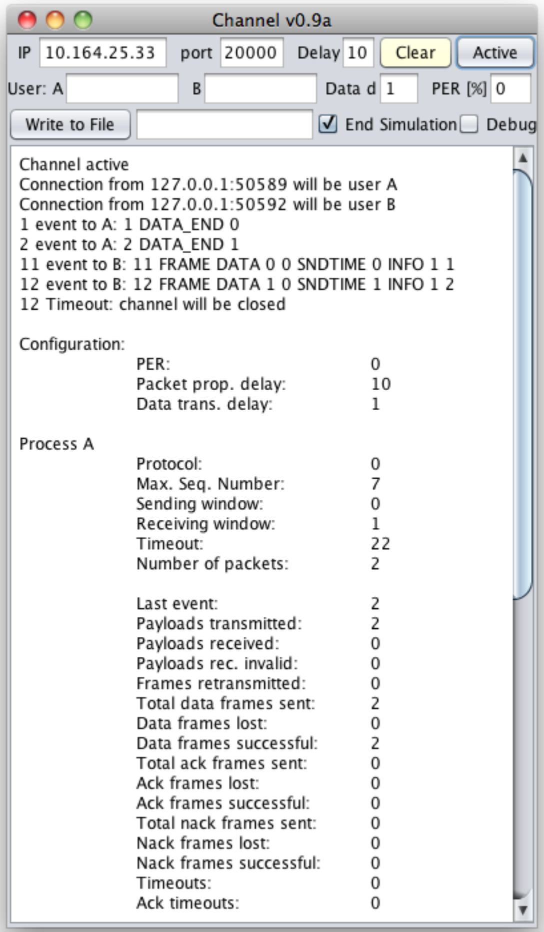


After booting, the *channel* accepts two TCP connections from two *Protocol* programs, starting a new simulation. The channel implements a discrete event scheduler, receiving the commands generated by the protocols and generating the events related to the physical level and the timers shown previously, ordered according to the simulation time. The simulation time is measured in units of virtual time, called *tics*.

Both the *protocol* and the *channel* provided with the statement write briefly (or exhaustively, in *debug* mode) the events and commands that are generated, and the contents of the queue with events waiting to be fired.

3. Program Development

3.1. Channel Application



```
Channel active
Connection from 127.0.0.1:50589 will be user A
Connection from 127.0.0.1:50592 will be user B
1 event to A: 1 DATA_END 0
2 event to A: 2 DATA_END 1
11 event to B: 11 FRAME DATA 0 0 SNDTIME 0 INFO 1 1
12 event to B: 12 FRAME DATA 1 0 SNDTIME 1 INFO 1 2
12 Timeout: channel will be closed

Configuration:
                PER:                        0
                Packet prop. delay:         10
                Data trans. delay:          1

Process A
                Protocol:                   0
                Max. Seq. Number:           7
                Sending window:             0
                Receiving window:           1
                Timeout:                    22
                Number of packets:          2

                Last event:                 2
                Payloads transmitted:       2
                Payloads received:          0
                Payloads rec. invalid:      0
                Frames retransmitted:       0
                Total data frames sent:     2
                Data frames lost:           0
                Data frames successful:     2
                Total ack frames sent:      0
                Ack frames lost:            0
                Ack frames successful:      0
                Total nack frames sent:     0
                Nack frames lost:           0
                Nack frames successful:     0
                Timeouts:                   0
                Ack timeouts:               0
```

The *channel* application is provided fully implemented. It consists of a Java application with the GUI shown on the right. Pressing the button "*Active*", the channel starts a *ServerSocket* on the IP and port shown, getting ready to receive connections from *protocols*, to be designated respectively the *protocol* (*User*) A and B.

The application allows you to perform various configurations of the simulation scenario:

- "*Delay*" – propagation time of frames in the channel;
- "*Data d*" – duration of a data frame (time between sending the frame and the event *Data_end*);
- "*PER [%]*" – average *Packet Error Rate* in the channel, which can affect all frames transmitted with equal probability;
- "*End Simulation*" – controls whether the channel automatically terminates a simulation when it does not receive events for a second, or if the decision is left to the user by pressing the "*Active*" button;
- "*Write to File*" – controls whether messages written to the screen are echoed to a file.

At the end of the simulation, it is a written report with the value of the various settings used in the channel and protocols, and several measures of system performance for protocols A and B. The measures are:

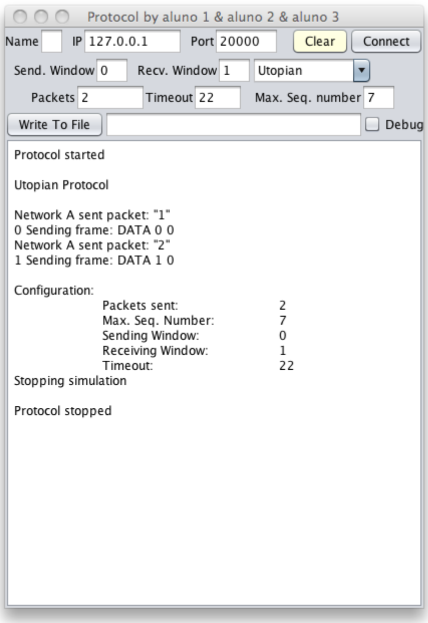
- *Last event* – time of the last event that the *protocol* received / originated, returning the total delay that occurred to send and receive all the data;
- *Payloads transmitted* – number of packets transmitted by the network level;
- *Payloads received* – number of packets received by the network level;
- *Payloads rec. invalid* – number of packets received out of order by the network level;
- *Frames retransmitted* – number of retransmitted data frames (the programmer should explicitly update this counter);
- *Total data frames sent* – total number of data frames sent;
- *Data frames lost* – number of data frames lost due to channel errors;
- *Data frames successful* – number of data frames received successfully;
- *Total ack frames sent* – total number of ACK frames sent;
- *Ack frames lost* – number of ACK frames lost due to channel errors;
- *Ack frames successful* – number of ACK frames received successfully;
- *Total nack frames sent* – total number of NACK frames sent;
- *Nack frames lost* – number of NACK frames lost due to channel errors;
- *Nack frames successful* – number of NACK frames received successfully;
- *Timeouts* – number of Timeout events received;
- *Ack timeouts* – number of ACK_Timeout events received.

The performance of the developed protocols will be measured using these metrics, highlighting the delay and the ratio of the total number of packets transmitted per data packet. It is suggested that the performance of the protocols is tested for: i) a scenario without error and with adjusted timeout (PER = 0 and Timeout = 22 tics), ii) scenario with errors and adjusted timeout (PER = 50% and Timeout = 22 tics), and iii) scenario with errors and a long timeout (PER = 50% and Timeout = 44 tics).

You can run the *channel* from the terminal with the command: `java -jar Channel.jar`

3.2. Protocol Application

The work consists solely in implementing the data link layer protocols. Everything else is supplied fully realized. The two figures below represent the *protocol* nodes A and B with protocol messages generated according to channel figure previously shown.



Name IP 127.0.0.1 Port 20000
Send. Window 0 Recv. Window 1 Utopian
Packets 2 Timeout 22 Max. Seq. number 7

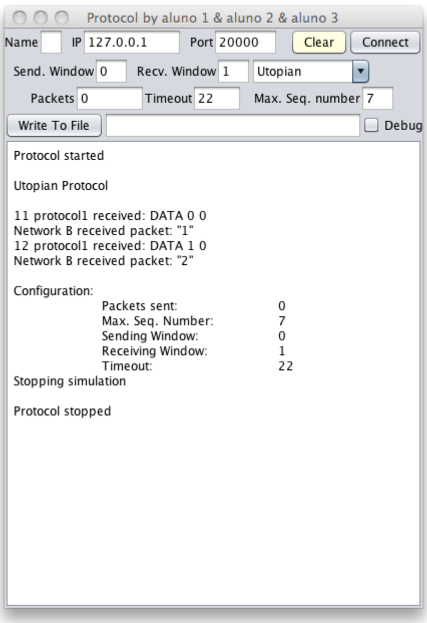
```
Protocol started

Utopian Protocol

Network A sent packet: "1"
0 Sending frame: DATA 0 0
Network A sent packet: "2"
1 Sending frame: DATA 1 0

Configuration:
                Packets sent:           2
                Max. Seq. Number:       7
                Sending Window:         0
                Receiving Window:       1
                Timeout:                22
Stopping simulation

Protocol stopped
```



Name IP 127.0.0.1 Port 20000
Send. Window 0 Recv. Window 1 Utopian
Packets 0 Timeout 22 Max. Seq. number 7

```
Protocol started

Utopian Protocol

11 protocol1 received: DATA 0 0
Network B received packet: "1"
12 protocol1 received: DATA 1 0
Network B received packet: "2"

Configuration:
                Packets sent:           0
                Max. Seq. Number:       7
                Sending Window:         0
                Receiving Window:       1
                Timeout:                22
Stopping simulation

Protocol stopped
```

The graphical interface allows you to define which channel is connected, selecting the IP and port. The simulation starts when you press the "*Connect*" button and the channel generates a start event simulation.

Through the graphical interface, you can set:

- *Packets* – the number of packets that will be sent during the simulation;
- *Max. Seq. number*– the maximum sequence number (generaly given by 2^n-1);
- *Send Window* – the sending window size;
- *Recv Window* – the receiving window size (it is always 1 in this work);
- *Timeout* – time waiting for an acknowledgment before resending a data frame.

There is also a combo box that allows you to choose the data link layer protocol: Utopian; Stop & Wait, Go-Back-N, and Go-Back-N with Nack. The objective is to develop the code for the last three protocols.

The given program consists of three packages, each having the following classes:

- Package `terminal`:

- *Terminal,java* (completed) – Main class with graphical interface that manages the timing of various objects used;
- *Connection.java* (completed) – Thread that habndles the TCP connection to the channel;
- *NetworkLayer.java* (completed) – Class that implements the network layer interface;

- Package `simulator`:

- *Frame.java* (completed) – Class that saves and serialises frames;
- *Event.java* (completed) – Class that saves and serialises events;
- *Log.java* (completed) – Interface that defines the *Log* function;

- Package `protocol`:

- *Simulator.java* (completed) – Interface that defines all the commands that can be used to implemente a daat link layer protocol;
- *Callbacks.java* (completed) – Interface that defines all the methods that must be implemented while performing a data link layer protocol;
- *Protocol1.java* (completed) – Implementation of the Utopian data link layer protocol;
- *Protocol2.java* (**to be completed**) – Implementation of the Stop & Wait protocol;
- *Protocol3.java* (**to be completed**) – Implementation of the Go-Back-N protocol;
- *Protocol4.java* (**to be completed**) – Implementation of the Go-Back-N protocol with NACKs;

Students will only modify the last three files, to implemente the desired protocols, mainly using the methods defined in the interfaces `Callbacks` and `Simulator`, described in the next section.

3.2.1. Commands

In the *Protocol* class code that is provided to students, you can invoke on the `sim` object the following methods (which were defined in the `Simulator interface`). The purpose of each is explained below:

- Get the size of the sending window:

```
        int get_send_window();
```

- Get the maximum sequence number:

```
•        int get_max_sequence();
```

- Get the timeout value:

```
•        long get_timeout();
```

- Get the current simulation time:

```
•       long get_time();
```

- Send the frame `frame` to the physical layer (i.e. to the channel):

```
•       void to_physical_layer(simulator.Frame frame);
```

- Start a timer associated to the key `key`. If it is started twice with the same key, the first timer is cancelled:

```
•       void set_timer(long delay, int key);
```

- Cancel the timer associated with the key `key`:

```
•       void cancel_timer(int key);
```

- Start the ACK timer:

```
•       void set_ack_timer();
```

- Cancel the ACK timer:

```
•       void cancel_ack_timer();
```

- Stop the simulation:

```
•       void stop();
```

To allow a correct counting of the number of retransmitted data frames, the method `count_retransmission` should be called each time a frame is retransmitted:

```
void count_retransmission();
```

3.2.2. Events

Think of the event as something that causes the invocation of the *callback* methods. These methods will be implemented by the students in a *Protocol* class, and were defined in the `Callbacks interface`:

- Start of simulation:

```
void start_simulation(long time);
```

- End of transmission of the data frame with the sequence number `seq`:

```
void handle_Data_end(long time, int seq);
```

- Firing of the timer associated with the key `key`:

```
void handle_Timer(long time, int key);
```

- Firing of the ACK timer:

```
void handle_ack_Timer(long time);
```

- Reception of the frame `frame` from the physical layer:

```
void from_physical_layer(long time, simulator.Frame frame);
```

- End of simulation:

```
void end_simulation(long time);
```

In all events, it is received the current simulation time in `time`.

3.2.3. Network layer

Class `terminal.NetworkLayer` defines the methods to exchange packets with the network layer, and is instanciated in `net` object:

- Get a new packet from network layer (if there is no longer any more to send, returns `null`):

```
String from_network_layer();
```

- Send a new packet to the network layer (in case of error returns `false`):

```
        boolean to_network_layer(String packet);
```

3.2.4. Creation and Reading of frames

The frames are objects of class `simulator.Frame (import Simulator.Frame)`. This class contains the fields as explained previously (seq, ack, info, etc.) plus another one called `kind`. In addition, it has static methods for creating new instances of objects, and methods for accessing various fields. The `kind` field defines the type of frames, having three valid values for a valid frame: `Frame.DATA_FRAME`, `Frame.ACK_FRAME` or `Frame.NACK_FRAME`; and the value `Frame.UNDEFINED_FRAME` when it is not initialized.

The method `kind()` can be used to get the value of `kind`.

To create a new frame of each of the three types it is possible to use the methods:

```
    public static Frame new_Data_Frame(int seq, int ack, String info);
    public static Frame new_Ack_Frame(int ack);
    public static Frame new_Nack_Frame(int nack);
```

To access the fields you can use the methods:

```
    public int seq();       // for DATA_FRAME
    public String info();   // for DATA_FRAME
    public int ack();       // for DATA_FRAME, ACK_FRAME, NACK_FRAME
    public long snd_time(); // time when it was sent
```

It is also possible to get a textual description of the contentes of a frame:

```
    public String kindString();   // return the kind of the frame
    public String toString();     // return the kind and the content summary
```

Two methods were defined to enable the transportation of frames through TCP sockets, which convert the contentes to and from strings (i.e. do object serialization). These functions will not be used by students, but generate the representation that is shown in the applications log.

```
    public String frame_to_str();
    public boolean str_to_frame(String line, Log log);
```

3.2.5. Utopian Protocol

The class *Protocol1* was developed to serve as an example for students. It performs the utopian protocol described above. The class adds to the interface `Callbacks` two state variables, two methods for sequence numbers and a method to centralize the sending of frames to the physical layer.

The state variables are used to control the sequence numbers used in the frames:

```
    private int next_frame_to_send; // Number of the next data frame to be sent
    private int frame_expected;     // Expected number for the next data frame received
```

The two methods are used to incremente/decrement a sequence number, which considers the maximum sequence number used in the simulation:

```
    int incr_seq(int n);
    int decr_seq(int n);
```

The method to send frames is private and is the following:

```
    private boolean send_next_data_packet() {
      String packet= net.from_network_layer();   // Get a packet from the network layer
      if (packet != null) {
        next_frame_to_send= incr_seq(next_frame_to_send);  // Increment the seq. number
        // Create a new Frame object
        Frame frame = Frame.new_Data_Frame(next_frame_to_send, frame_expected, packet);
        sim.to_physical_layer(frame);    // Send the frame to the physical layer
        return true;
      }
      return false;  // Failed; no more packets to send
    }
```

This method is called:

- In the beginning of the simulation:

```
public void start_simulation(long time) {
  sim.Log("\nUtopian Protocol\n\n");
  send_next_data_packet();    // Start sending the first data frame
}
```

- When the sending of a data frame ends:

```
public void handle_Data_end(long time, int seq) {
  send_next_data_packet();    // Send the next data frame
}
```

Frames are received in method `from_physical_layer`, which does a little more than the original version of the book – it verifies if it is the expected sequence number, and advances it to the next frame.

```
public void from_physical_layer(long time, Frame frame) {
  sim.Log(time + " protocol1 received: " + frame.toString() +"\n");
  if (frame.kind() == Frame.DATA_FRAME) {     // Check the frame kind
    if (frame.seq() == frame_expected) {    // Check the sequence number
      net.to_network_layer(frame.info()); // Send the frame to the network layer
      frame_expected = incr_seq(frame_expected);
    }
  }
}
```

The remaining methods of interface `Callbacks` are not used in this protocol. They simply write that were invoked.

3.3. GOALS

A sequence for the development of the work can be:

1. Program the *Stop&Wait* protocol in class `Protocol2`. Start by studying the implementation of Utopian protocol, in class `Protocol1`, to see what can be reused;
2. Program the *Go-Back-N* protocol with a single timer in class `Protocol3`. Start by looking into your implementation of the *Stop&Wait* protocol, in class `Protocol2`, to realize what you can reuse;
3. Complete the programming of *Go-Back-N* protocol, introducing the use of multiple parallel timers for the different frames and the ACK timer in `Protocol3` class (do not forget to backup the first version);
4. Program the *Go-Back-N* protocol with NACKs in class `Protocol4`, starting from class `Protocol3`.

ALL students must try to **complete phase 2**. In the first week of the work a general introduction to the work is made and phase 1 should be finished. At the end of the second week the phase 2 must be almost completed. At the end of the third week you should have ended phase 3. At the end of the fourth and final week you should try to achieve the maximum number of phases, taking into account that it is preferable to do less well (running and no errors), than everything and nothing works.

STUDENT POSTURE

Each group should consider the following:

- Do not waste time with the aesthetics of input and output data;

- Program in accordance with the general principles of good coding (using indentation for comments, using variables with names conform to its functions ...) and;
- Proceed so that the work is equally distributed to the two members of the group.