NOVA SCHOOL OF
SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

Reliable Data Transmission over an Unreliable Network

(TPC2 – Redes de Computadores 2025/2026)

This assignment tackles the problem of reliably delivering information across a network where packet loss occurs. The assignment involves two programs cooperating in the reliable transfer of a file, using UDP datagrams, in an environment where data and acknowledgement datagrams can be (and will be) lost. The scenario comprises two processes:

- **Sender**: a server program that sends the contents of the file in fixed-size blocks to a receiver process located in another machine.
- **Receiver**: a client program that receives the file's blocks, acknowledging each block received.

The sender and receiver handle potential data and acknowledgement loss using a *sliding window* of dimension N and a Go-Back-N (GBN) strategy. The receiver has a receiving window that can only accommodate 1 block. The file transfer must be done in blocks, with the block size defined by the receiver.

The file transfer has two phases:

1. **Start phase** – the receiver sends a request for a file transfer from the sender with the format: `(filename,block_size)`, where *filename* is a string and *block_size* is an integer. If the sender has the file, it will answer with a UDP datagram containing the values `(0,filesize)`, otherwise it will send a UDP datagram containing the values `(-1, 0).`
   Assume that there are no losses in this phase.

2. **File Transfer phase** – the sender transfers the file requested by the receiver. The format and purpose of each UDP datagram at this stage are as follows:

   Data packet from sender to receiver: (*block_number*, *data*)

   representing a block of file data. The *block_number* is an integer sequence number representing a certain block, starting at 1 for the first block of a file. The *data* represents the file's data block encoded as raw bytes. The maximum UDP payload size is the chosen block size plus the number of bytes occupied by the *block_number* codification .

   Acknowledgment packet from receiver to sender: (*block_number*)

   confirming the last correctly received block. The *block_number* represents a cumulative sequence number, that is, a sequence number that acknowledges all the blocks that have already been received.

2. Sender and receiver using Go-Back-N (GBN)

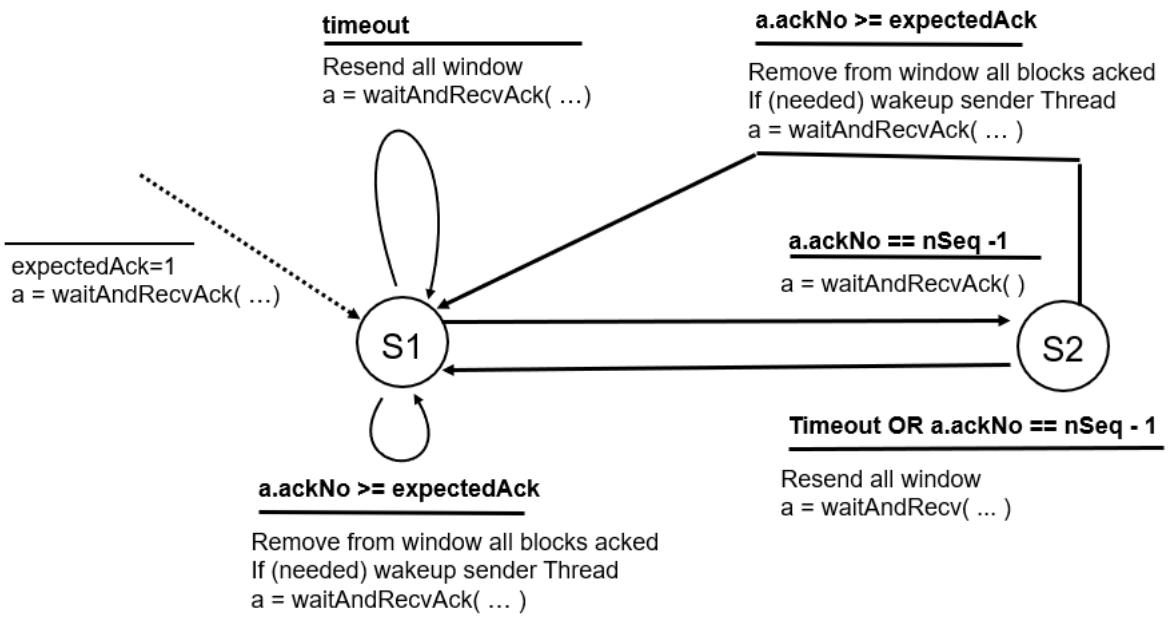
The following code specifies the behaviour of the sender and receiver when transferring a sequence of messages using a Sliding Window mechanism with the GBN protocol; sender has a sliding window with a capacity for N blocks; the size of the receiver window is 1. To accelerate the recovery of lost data packets, the sender also resends all the blocks in its sliding window when it receives two consecutive acknowledgements for a block that has been already acknowledged.

Both the sender and receiver programs have two threads.

Sender program:

- Main thread that reads the blocks from the file and sends the data to the receiver using a sliding window approach.
- Helper thread (`tx_thread`) that manages the sliding window buffer and handles timeouts and retransmissions.

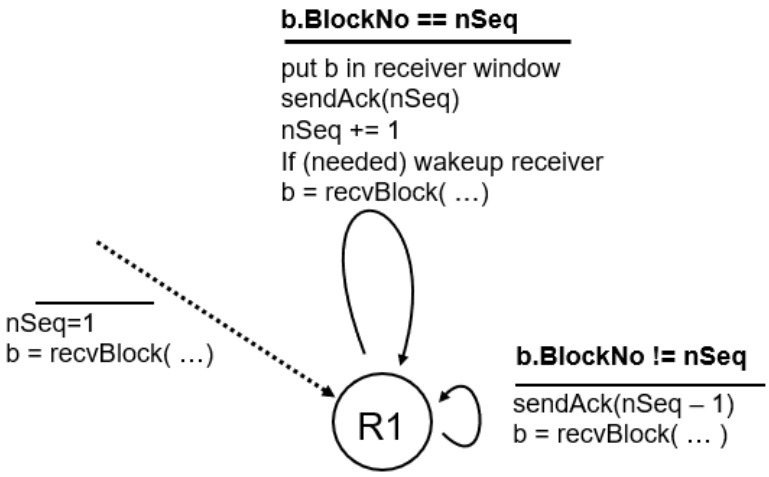
The actions of *tx_thread* are sketched in the following finite state machine:



Receiver program:

- Helper thread (`rx_thread`) that handles the reception of blocks and the sending of acknowledgments.
- Main thread that gets blocks and writes them to a file.

The actions *rx_thread* should follow the finite state machine below:



The two programs transfer one file and then terminate. The end of the file transfer will need to be handled at both the sender and the receiver.

3. Implementation

Your task is to complete the code of two Python programs called *sender.py* and *receiver.*py available on CLIP. Analyse the source code the provided programs:

- *Sender* (server program) is invoked by writing in the command line:

```
python sender.py senderPort windowSize timeoutInSeconds
```

- *Receiver* (client program) is invoked by writing in the command line:

```
python receiver.py senderIP senderPort fileNameInSender fileNameInReceiver blockSize
```

You have to complete the code for:

sender.py – implementing the tx_thread() and sendBlock(...) functions.

receiver.py – implementing the rx_thread(...) function.

Both sender and receiver programs should terminate after guaranteeing that the file was transferred correctly. Your code should work even if the sender and the receiver programs run on different machines. The easiest way to achieve this goal is by using two containers over *Docker*.

Using the pickle Python package to build the payload of your UDP messages is highly recommended (see TPC1 in lab03).

Simulation of datagram loss

To simulate datagram loss, programs should call the following function.

Note that losses (only in the file transfer phase) can occur in both directions of communication, that is, from sender to receiver and from receiver to sender.

```
import random
…
def sendDatagram (msg, sock, address):
	# msg is a byte array ready to be sent
	# Generate random number in the range of 1 to 10
	rand = random.randint(1, 10)
	if rand > 2:
		sock.sendto(msg, address)
```

Waiting for a datagram with timeout

```
import select
…
def waitForReply( uSocket, timeOutInSeconds ):
	rx, tx, er = select.select( [uSocket], [], [], timeOutInSeconds)

	# waits for data or timeout
	if rx == []:
		return False
	else:
		return True
```

3. Delivery

The delivery should be made **before 23:59 on October 20, 2025**. The code developed will be uploaded through Moodle. Further details will be sent in a CLIP message.