NOVA SCHOOL OF
SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

YATFS – Yet Another Transfer File Service
(TPC2 – Redes de Computadores)

1. Introduction

In this assignment we will implement the client and server programs of a File Transfer Read Service using UDP sockets; the server handles only one request at a given time. The transfer is done by sending *chunks* of the *file*. Considering a file with L bytes as an array of bytes with elements from *file[0]* to *file[L-1]*, a file chunk is defined by:

- *offset B:* the distance in bytes to the beginning of the file
- *size S:* number of bytes contained in the chunk
- chunk (B,S) contains the *file[S]* to *file[B+S-1] bytes*

The server waits for requests in the UDP port *portSP*. There is only one type of request. The request is a datagram containing a Python tuple with:

- the name of the file – ***file_name***
- the offset – ***B***
- the number of bytes of the chunk – ***S***

The server replies with a datagram that contains a tuple with

- the status of the reply:
    - 0 - OK
    - 1 - file does not exist
    - 2 - invalid offset
- number of bytes transferred (can be less than the requested if B+S is greater than the file size)
- the bytes transferred

In this homework, the possibility of UDP datagram loss is considered. We will assume that requests are never lost but that replies can be lost.

2. Datagram loss and select

As said before, the possibility of UDP datagram loss is considered. As in our setting, this almost never happens, we will simulate this situation by imposing that the server uses the following code to send datagrams:

```
import random

…

def serverReply (msg, sock, address):
	# msg is a byte array ready to be sent
	# Generate random number in the range of 0 to 10
	rand = random.randint(0, 10)
	# If rand is less is than 3, do not respond
	if rand >= 3:
		sock.sendto(msg, address)
	return
```

As you can see, 20% of the times the server will not reply, simulating a datagram loss. We will assume that requests sent by the client will not be lost.

This situation requires that the client must handle a situation where a reply to the request does not arrive. Waiting for a reply that may not arrive can be done by the following code that uses the ***select*** function.

The POSIX ***select()*** system call enables a system to keep track of several input/output descriptors. So, the *select* system call waits for one of the descriptors or a whole to turn out to be "ready" for a particular type of I/O activity (for example, input possible). In ***select()*** one can specify a:

- list of channels to be tested for input availability,
- list of channels to be tested for output readiness,
- list of channels to be tested of some exceptional event occurrence (for example, error)
- timeout

The ***select()*** system call finishes if one of the following conditions occurs:

- one or more of the input channels has data to be read,
- one or more of the output channels are ready to send data,
- an error occurs in one or more of the channels mentioned in the third list,
- timeout expires without any event in the above three sets of channels

More information about the ***select()*** system call can be found in several places in the internet, namely:

https://www.tutorialspoint.com/unix_system_calls/_newselect.htm

Use of ***select*** in Python is much simpler than in C. Detailed information can be found in:

https://docs.python.org/3/howto/sockets.html#socket-programming-howto

https://docs.python.org/3/library/select.html

The following code returns *True* if data is available for reading in ***uSocket*** or False if, after 1 second wait, no reply was received.

```
import select
...
def waitForReply( uSocket ):
      rx, tx, er = select.select( [uSocket], [], [], 1)
      # waits for data or timeout after 1 second
      if rx==[]:
            return False
      else:
            return True
```

Please note that, if ***waitForReply*** returns *True*, one must still call ***recvfrom(…)*** for receiving the datagram.

3. Serializing and deserializing data

The datagrams exchanged between client and server contain arrays of bytes. How to put Python data structures like tuples in a datagram? And how to retrieve a tuple from a datagram?

The conversion from an object declared in a programming language to a byte array is called *serializing*; the conversion from a byte array to an object to be used in a programming language is called deserializing (more about this in the Distributed Systems course of next semester).

In Python, the methods available in the ***pickle*** package allow the serialization and deserialization of data structures like tuples. To serialize we use the method ***dumps*** and to deserialize the method used is ***loads***. The following example illustrates the use of ***pickle***:

```
import pickle
...

request = (fileName, offset, blockSize)
req = pickle.dumps(request)
UDPSocket.sendto(req, endpoint)

...
```

```
import pickle
...

message, address = sock.recvfrom(1024)
request=pickle.loads(message)
fileName = request[0]
offset = request[1]
noBytes = request[2]
print(f'file= {fileName},offset={offset},
       noBytes={noBytes}')
...
```

4. Work to do

1) Server

The server should be implemented in Python and be invoked with the command:

python server.py portSP

Its actions should correspond to the following pseudo-code. For replying to the client only the function *serverReply* can be used.

```
create socket ss and bind it to portSP
while TRUE:
      receive datagram with request and deserialize it using pickle.

      open file for reading, if open fails reply
        with a datagram with status 1; the other fields must be filled

      verify if the offset is valid using os.path.filesize(…) method
        if open fails reply with a datagram with status 2

      if both previous tests succeed use seek to position the file pointer
        in the required position and try to read S bytes from the file
        reply with a tuple (0, no_of_bytes_read, file_chunk)

      serialize the reply using pickle and call serverReply
```

2) Client that tests the server by getting the full contents of a file in 1K byte chunks
The client should be implemented in Python and be invoked with the command:

python client.py host_of_server portSP fileName chunkSize

Its actions should correspond to the following pseudo-code. Client should handle the situation where the reply to a request does not arrive.

```
create socket sc and bind it to some UDP port
open local file for writing
offset = 0
while TRUE:
      prepare request(fileName,offset,size) and serialize it with pickle
      send the request to (host_of_server, portSP)
      wait for reply; if reply does not arrive, repeat request
      write byte chunk received to file
      if EOF
            break
      else
            offset = offset + size
```

3) Study the performance of the of the client (Optional - 2 points)

Consider a file with 20M bytes. Obtain the transfer time for chunk sizes 64 bytes, 1024 bytes, 4K bytes and 32K bytes. Repeat with drop rates of 10%, 25% and 40%. What can you infer of the results obtained?

Obs: It will be important to experimentally study the behavior of your implementation, when the block size changes (use the chunk sizes indicated above or other additional sizes), checking the impact on file transfer performance (transfer rate). You can measure the time (t0 ms) before starting the transfer file and again when the transfer is completed (t1 ms). You can calculate the transfer performance Tr (transfer rate) as follows: Tr = F_size / (t1-t0). Where F_size is the file size in bytes. This allows you to obtain the transfer rate, for example in Kbytes/sec, once the respective conversions have been made.

5. Delivery

Details about the delivery will be sent later. You must build a zip file with the code of client and the server. The delivery should be done before **10 AM of October 17, 2023**.