# William Coulter z5113817 – COMP3331 Assignment Report

## Background – Python 3.7.2

Writing this assignment introduced a couple of new tools being *threading* and the Python module *Pickle*. Demo found at ***https://youtu.be/1PNjxK1L3Xw.***

### Threading

Threading allows my program to have multiple processes running in the background that do not stop the execution of the main thread. This is done by creating a *class* inherited from *threading.Thread* which can be passed variables and have its *run* function overwritten. When this class is instantiated and its *.start()* property is called in the main thread, the overwritten *run* function in the instantiated thread will execute. The main thread can either continue executing or wait until the newly spawned thread has finished its execution before continuing if specified.

```python
import socket
import threading
import pickle

from supporting.Messages import newSuccessorMessage

# sends information about new successor to target over TCP
class newSuccessor(threading.Thread):
    def __init__(self, target_id, new_successors):
        threading.Thread.__init__(self, daemon=True)
        self._target_id = target_id
        self._new_successors = new_successors

    def run(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            information = newSuccessorMessage(self._new_successors)
            msg = pickle.dumps(information)
            s.connect(('127.0.0.1', 50000 + self._target_id))
            s.sendall(msg)
```

*Figure 1 - example of a thread class. This thread sends a message over TCP to another peer, updating its target's successors*

What this means is, each peer can have its own UDP Server running, its own TCP Server and ping its successors in the background while it waits for terminal input. Figure 2 shows the structure of how each peer executes using threads and visually displays the structure of each peer's code. Note that "Task Thread" is a thread that executes from command-line input, for example "request X" or "quit."

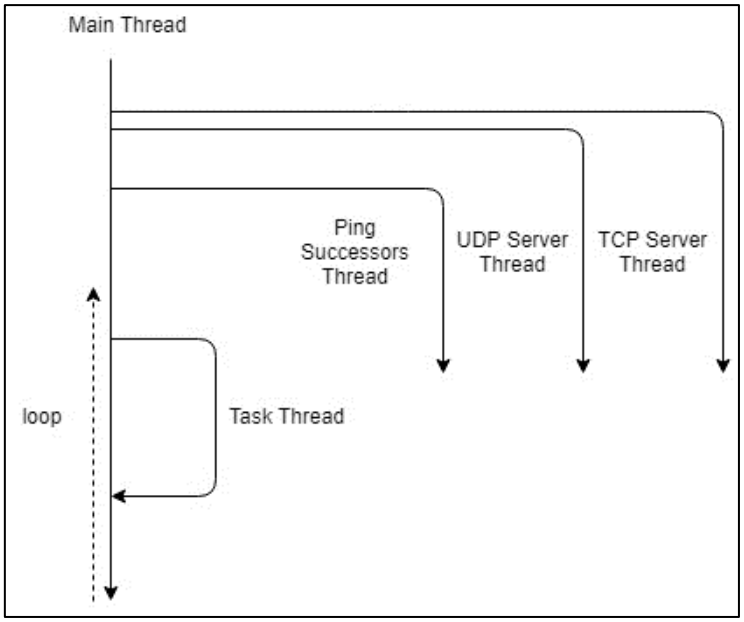


*Figure 2 - Code structure of each peer's execution using threads*

**Pickle**

Pickle allowed me to attach a lot of meaning to the messages that each of my peers sent each other. Messages over sockets have to be sent in *bytes* format. As a result, it can be very tedious trying extract information out of a straight bytes packet, especially when the packet needs to contain information on segment size, sequence number, ack number, data, new successors etc. The *pickle.dumps* function converts a meaningful (to people) python object into bytes that can be transmitted over sockets. From the receiver's end, the *pickle.loads* function converts the received string of bytes back into its meaningful state that can then be interpreted.

My *Messages.py* file contains all the different message types that I used for this assignment. Note that these "messages" are just different classes with different attributes which I can instantiate with given variable before sending them off to another peer. Figure 2 shows how the instantiated object *request* which contains valuable information on the peer who is requesting a file, the file number being requested, and its own peer id is easily transferred into bytes and sent to its successor.

```
def run(self):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        # create pickle message
        request = requestFileMessage(self._requestor_id, self._file_no, self._id)
        msg = pickle.dumps(request)

        # send request to successor
        s.connect(("127.0.0.1", 50000 + self._successor))
        s.sendall(msg)
```

*Figure 3 - Example of how the Pickle module is used to easily construct a bytes message*

The successor's TCP Server can then easily unpack (or *depickle*) this message using *pickle.loads* and extract meaning from the original byte string.

Threading and Pickle were two crucial tools in writing the functionality for this assignment, which I will go over in the next section.

## Ping Successors

As shown in Figure 2, each peer pings its successors in the *Ping.py* thread. This thread calls the *PingRequest.py* thread in a loop to contact its successors over UDP. Response are then received and printed to the terminal. While the peer is pinging, it is also *listening* for messages from its predecessors in its *UDPServer.py thread.* Each peer then keeps track of its predecessors through this.

In terms of design choices, Peers ping their successors every 1 second. This interval was decided because it was regular enough such that the peer system can know reasonably soon when a peer has disconnected, but not too frequent such that the UDP Servers of each peer are flooded.

## Requesting a File

When a peer receives "request X" in its terminal, it will call on the *FileRequest.py* thread which sends a message over TCP to its successor, asking if it is responsible for the file it is requesting. If a peer is responsible, it's *TCPServer.py* thread will call the *FileTransfer.py* thread, to transfer the file over UDP.

The *FileTransfer.py* thread first sends a response message to ensure the requestor is still alive. The requesting peer's UDP server will confirm that it can receive the file and send an ACK. Once the ACK is received by the sender, the file transfer begins.

The sender sends packets of the maximum segment size and employs a *stop and go* protocol. These packets contain information on the file number, the actual data read from the file, the ack number, the sequence number, the maximum segment size and the drop probability. All of this is nicely packaged in the *fileTransferMessage* class and pickled into bytes before sent as shown in Figure 4.

```
63      data = fileTransferMessage(self._file_no, bytes_read, ack_no, seq_no, self._MSS, self._drop_prob)
64      msg = pickle.dumps(data)
65      s.sendto(msg, ('127.0.0.1', 50000 + self._requestor_id))
```

*Figure 4 - Epitome of how useful pickling a class can be when many data fields are used*

Before this happens, the sender decides if the packet was to be loss, according to the drop probability. If this is the case, the sender log file is logged, and the sender waits 1 second (to simulate timeout) before resending.

If no packet loss occurs, the receiver receives the message, writes to its new file, writes to its receiver log and sends an ACK back. Once the receiver receives the ACK, the next section of the file is sent. This repeats until the entire file has been transferred.

## Peer Departure

The challenge with this section was transferring information on an update of variable value between threads. When a peer "quits," it sends a message to its successor and predecessors' *UPDServer.py* threads, but we want this information to transfer into its *Main Thread*. This was a challenge because not only does calling a thread mean that you instantiate it with its own copy of its variables and therefore changing these variables in the child thread doesn't alter the variables in its parent, but Python as a language *doesn't have pointers!* Objects are passed to functions *by reference* so I can't just simply give a pointer variable to all my threads for them to update when I instantiate them.

I overcame this problem by researching[1] and discovering that *immutable* object in Python, such as lists, do bind multiple variables to the same instance. So, if I passed my *UDPServer.py* and *TCPServer.py* a list containing its successors, altering this list in the child thread would translate into the main thread!

With this new understanding of Python variables, when a peer wants to "quit," it calls on *NewSuccessor.py* and *NewPredecessor.py* which informs other peers' *TCPServer's* of their new successors, and each peers' Main thread is, as such, reflective of this change.

## Kill a Peer

Ping messages that peers send each other contain a sequence number. If a sequence number that a peer is sending is more than 3 past the most recent sequence number received, the peer is considered no longer alive, meaning a peer can be dead for seconds before the system is aware. This is a reasonably small amount of time, however is also reliable since at least 4 pings have to be consecutively missed, and is why the interval was chosen. The peer that has noticed one of its successors is no longer alive, will call the *RequestNewSuccessor.py* thread and ask its alive successor for its new successor. Note that this is done in the *PingRequest.py* Thread.

Although this isn't required from the assignment's marking criteria, peers then update their predecessors through their *UDPServer.py* thread, as they become aware that they are receiving pings from a different peer.

[1] http://qa.geeksforgeeks.org/1540/qa.geeksforgeeks.org/1540/does-python-support-pointers-like-c.html