SCTP AND DCCP SERVER-CLIENT IMPLEMENTATION

Ioannis Mitro
AEM: 2210

Giorgos Fragkias
AEM: 2118

Chrysovalantis Gkagkas
AEM: 1980

January 29, 2020

1 Introduction

The project was based in the implementation of SCTP and DCCP protocol used into a server-client connection. The two concepts are implemented using Python in the same mindset, with the client asking for a .html page over HTTP with a GET request, and the server sending the file over the respective protocol. We will discuss the code mainly, and parts of the implementations that are the same on both SCTP and DCCP - mainly the HTTP handling part - will be discussed only in the SCTP section.

2 Stream Control Transmission Protocol (SCTP)

2.1 Python library

The python library we used for this section is **pysctp**, the official python module for the socket API implementation of the SCTP protocol stack and library. Inside the code, it is imported using *import sctp*, and under the hood it uses functions written in C language, used in the libsctp package that can be installed using the usual apt-get install command in any SCTP aware linux kernel.
Using the pysctp module we described above, the socket class is called sctpsocket, and is the root class of the socket we would use for the implementation. The package offers two "styles" of this class, TCP and UDP style. We decided to use the sctpsocket_tcp as base for our connection.

2.2 The client

The client python script we created is actually a simple one, and consists of a SCTP_client class and three functions that initialise the class, request the file from the server and finally show the file in a new firefox tab.
The class, other than the constructor, provides two methods. One that makes the request and gets the results and one that closes the connection.
In the constructor method (__init__) we are specifying the socket family AF_INET to create an SCTP socket using the sctpsocket_tcp class constructor.
After that, we are actively connecting to the socket opened in the server using the IP and port. Since we are performing an HTTP connection in a localhost server the two values are 127.0.0.1 and 80.
The make request method, that is the most functional method of the class sends a request specified by an argument using the socket and receives from the socket two data units, one for the headers of the HTTP reply and one for the body. If the reply is not a 404 "not found" one, it uses firefox to open the html file it got.

```python
class SCTPclient:
  def __init__(self,serv_ip,serv_port):

    # Initialize the socket
    sock = sctp.sctpsocket_tcp(socket.AF_INET)
    sock.connect((serv_ip,serv_port))
    self.socket = sock
    self.dataSize = 8000

  def makeRequest(self,req):
    self.socket.sctp_send(req)
    print(req)
    headers = self.socket.recv(self.dataSize)
    body = self.socket.recv(self.dataSize)
    body = body.decode("utf-8")
    print(body)
    if "Not Found" not in body:
      with open("out.html","w") as f:
        f.write(body)
      os.system("firefox out.html")
  def close_connection(self):
    self.socket.close()
```

Figure 1: The sctp client class

The main code running when we run the script (using python client.py) simply performs the construction of the SCTP socket, the connection, the request, the reply evaluation the appearance of the result html page in a firefox tab and finally the closing of the connection.

```python
client = SCTPclient("127.0.0.1",8080)
client.makeRequest("GET / HTTP 1.0")
client.close_connection()
```

Figure 2: The sctp client running commands

2.3 The server

The server class we implemented provides a constructor method that firstly creates the socket using sctpsocket_tcp and clears any events. Then it binds on our ip and port (127.0.0.1, 80) and listens for any connections.

```python
class SCTPserver:
  def __init__(self,serv_ip,serv_port,conns):

    # Initialize the server parameters
    self.ip = serv_ip
    self.port = serv_port
    self.connections = conns

    # Create the server
    sock = sctp.sctpsocket_tcp(socket.AF_INET)
    sock.events.clear()
    sock.bind((self.ip,self.port))
    sock.listen(self.connections)

    self.socket = sock
    self.dataSize = 8000
```

Figure 3: The sctp server constructor

It also provides a run method that simply accepts incoming connections on a loop and uses the method fetch to answer to the HTTP requests.

```
23    def run(self):
24
25      while True:
26
27        # Waiting for client to connect
28        client,addr = self.socket.accept()
29
30        # Fetch data with socket
31        self.fetch(client)
```

Figure 4: The run method

The fetch method uses recv to receive an incoming request, decodes it, and if it is a GET request, it tries to fetch the file the user asks for. Then it calls the method create_response.

```
45    def fetch(self,client):
46
47      # Retrieve the client's request
48      req = client.recv(self.dataSize)
49
50      # Check the format
51      if(self.checkFormat(req)):
52
53        # Tokenize the request
54        tokens = req.decode('utf-8').split()
55
56        # Retrieve the requested object
57        if tokens[1] == '/':
58          filename = "index.html"
59        else:
60          filename = tokens[1].replace("/","")
61        self.create_response(self.checkIfFileExists(filename),client,filename)
62      else:
63        print(req)
```

Figure 5: The fetch method

Create_response is simply the HTTP part handler that sends the http headers (with 200 if the file was found and 404 if not) and the body if the file exists.

```
33    def create_response(self, found, c, filename):
34      if found == True:
35        with open('./'+filename, 'r') as content_file:
36          content = content_file.read()
37          c.send(b'HTTP/1.0 200 OK\nContent-Type: text/html\n\n')
38          print(content)
39          c.send(content.encode())
40          c.close()
41      else:
42        c.send(b'HTTP/1.0 404 Not Found\nContent-Type: text/html\n\n')
43        c.close()
```

Figure 6: The create_response method

When we finally run the server python script, the script simply creates a new SCTPserver object using the constructor and uses the run method to wait for new connections by clients and answer them appropriately when they arrive.

2.4 HTTP Connection

For testing the code we wrote we opened two tabs in our terminal, one to run the server and one for the client. below is the two tabs discussion:

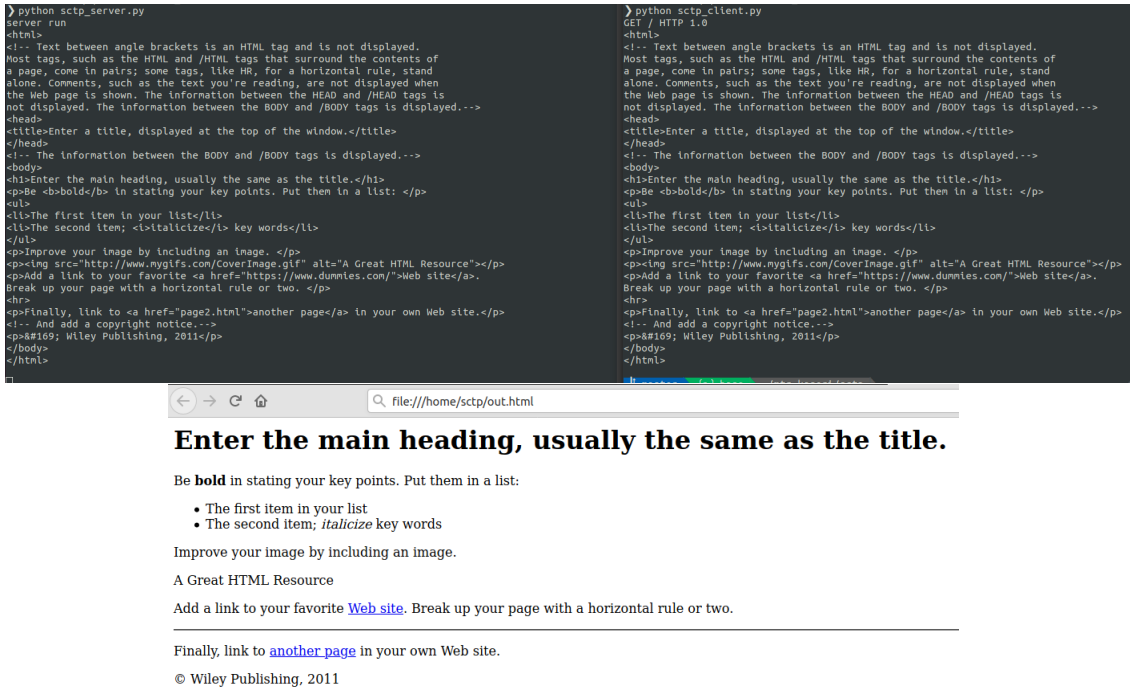

Figure 7: The HTTP GET request and reply

We can see that the server prints *server run* when we run him that indicates waiting for connections. When we run the client the request is made (we can see the GET / HTTP 1.0 header in the stdout), the server fetches the html file (a sample index.html file we created) prints the html's content in the stdout for us to see and proceeds to create an HTTP reply.
The client receives the HTTP reply containing the headers, extracts the html file in the case that it exists and opens a new firefox tab to preview the html file.

2.5 Wireshark Logs

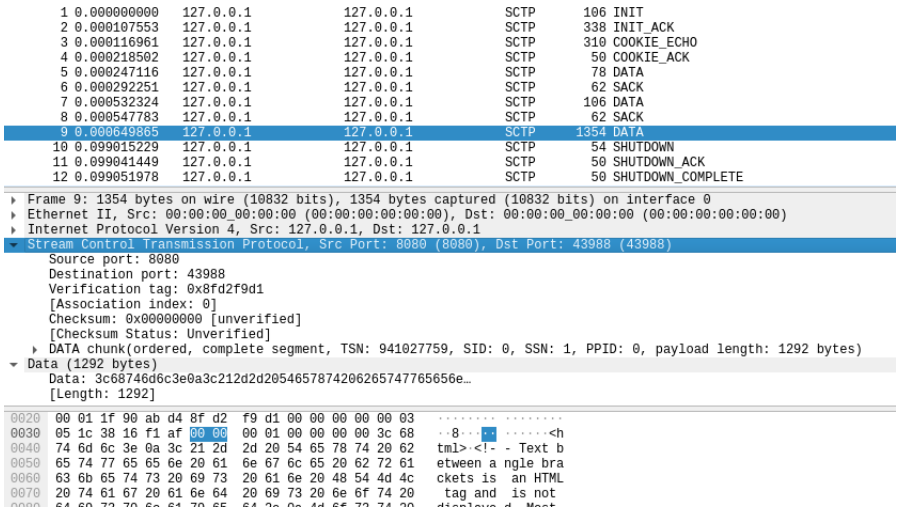

Figure 8: Wireshark Logs regarding the session

3 Datagram Congestion Control Protocol (DCCP)

3.1 Python library

The python library we used for this section is the classic **socket** library. Inside the code, it is imported using *import socket*, and it implements the classic socket functionality. In order to use the DCCP protocol, we had to use certain options inside the socket that would make the socket function under DCCP. To do that we used mainly the method *setsockopt* of socket that did most of the difference in the code. The rest of the code is the same implementation as SCTP regarding the part of the HTTP connection.

3.2 The client

The client python script we created consists of a DCCP_client class and three functions that initialise the class, request the file from the server and finally show the file in a new firefox tab.
The class, other than the constructor, provides two methods. One that makes the request and gets the results and one that closes the connection like the SCTP client class.

After that, we are actively connecting to the socket opened in the server using the IP and port. Since we are performing an HTTP connection in a localhost server the two values are 127.0.0.1 and 80.
The make request method, is the same method as the sctp client we discussed above.//

```
18  class DCCPclient:
19      def __init__(self, servIp, servPort):
20
21          # Initialize the socket
22          self.socket = socket.socket(
23              socket.AF_INET, socket.SOCK_DCCP, socket.IPROTO_DCCP)
24          self.socket.setsockopt(
25              socket.SOL_DCCP, socket.DCCP_SOCKOPT_PACKET_SIZE, packet_size)
26          self.socket.setsockopt(
27              socket.SOL_DCCP, socket.DCCP_SOCKOPT_SERVICE, True)
28          self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
29
30          # Connect
31          self.dataSize = 8000
32          self.ip = servIp
33          self.port = servPort
34          self.socket.connect((self.ip, self.port))
```

Figure 9: The dccp client class

The parameters (constants of socket) defined from us so that the socket is using the DCCP Protocol are the following:

```
4   socket.DCCP_SOCKOPT_PACKET_SIZE = 1
5   socket.DCCP_SOCKOPT_SERVICE = 2
6   socket.SOCK_DCCP = 6
7   socket.IPROTO_DCCP = 33
8   socket.SOL_DCCP = 269
9   packet_size = 512
10  socket.DCCP_SOCKOPT_AVAILABLE_CCIDS = 12
11  socket.DCCP_SOCKOPT_CCID = 13
12  socket.DCCP_SOCKOPT_TX_CCID = 14
13  socket.DCCP_SOCKOPT_RX_CCID = 15
```

Figure 10: The dccp opts

The rest of the code of the client works in the same way as the sctp client. Opening a connection, sending the request, getting the html, viewing it and closing the connection.

3.3 The server

The server class we implemented has the same constructor as the DCCP_client class, except that it binds and listens to the interface instead of actively connecting. The parameters used in the socket.setsockopt method are the same with

Figure10.
When we finally run the server python script, the script simply creates a new DCCPserver object using the constructor and uses the run method to wait for new connections by clients and answer them appropriately when they arrive. At the end, it closes the connection.

```
93  server = DCCPserver("127.0.0.1", 8080, 3)
94  print("server run")
95  server.run()
96  server.connection_close()
```

Figure 11: The HTTP GET request and reply

3.4 HTTP Connection

For testing the code we wrote we opened two tabs in our terminal, one to run the server and one for the client. below is the two tabs discussion:

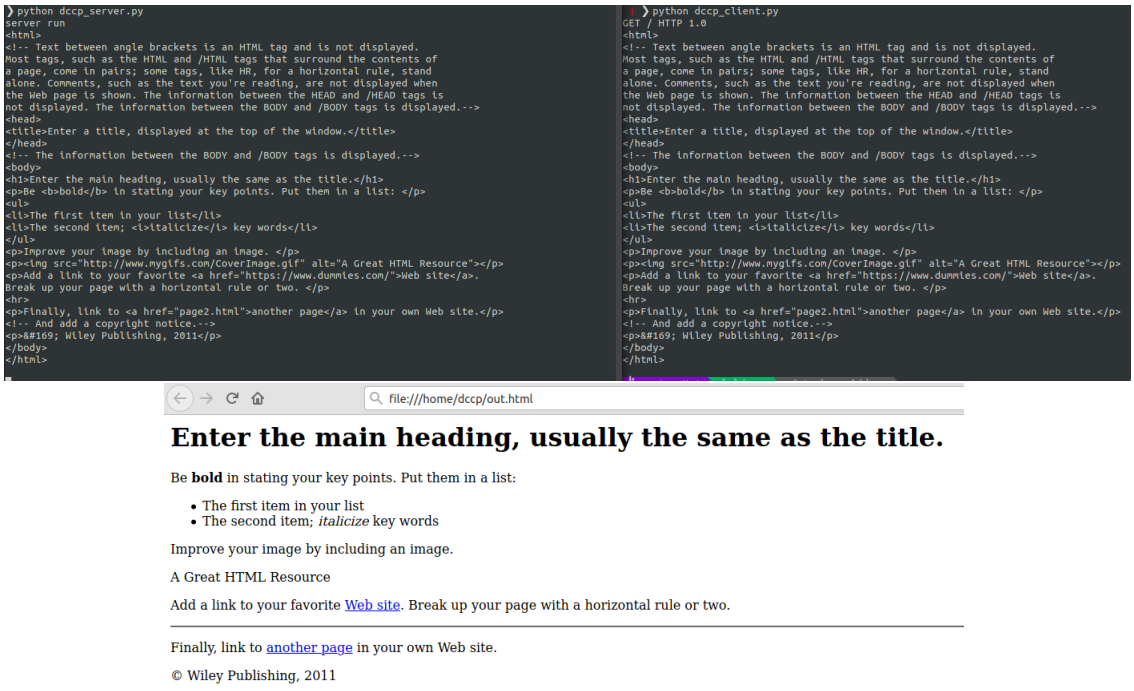

Figure 12: The HTTP GET request and reply

We can see that the server prints *server run* when we run him that indicates waiting for connections. When we run the client the request is made (we can see the GET / HTTP 1.0 header in the stdout), the server fetches the html file (a sample index.html file we created) prints the html's content in the stdout for us to see and proceeds to create an HTTP reply.
The client receives the HTTP reply containing the headers, extracts the html file in the case that it exists and opens a new firefox tab to preview the html file.

3.5 Wireshark Logs

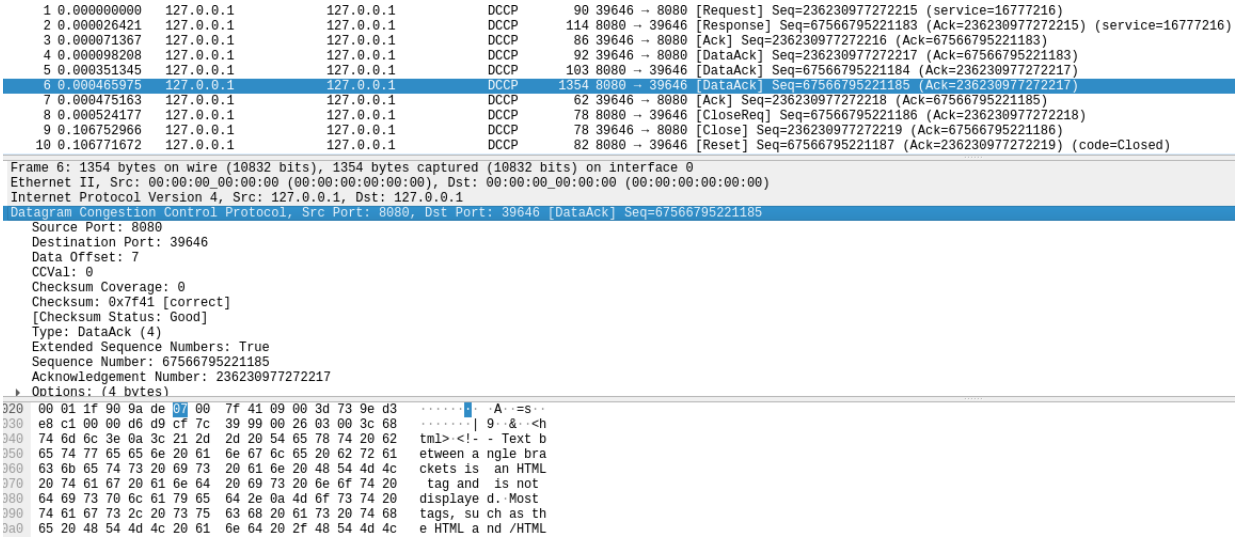

Figure 13: Wireshark Logs regarding the session

4 GET Request over DCCP/SCTP for binary files

Doing an HTTP GET request and getting back an html webpage is all good, but we wanted to implement getting binary files as well, such as images/videos/audio etc. .
In order to do that, we added some code for the case of binary files, and basically did not encode/decode our content with UTF-8 and we read/wrote in binary from and to files.
So, let's take a cute cat image and try to get it with an HTTP GET request over SCTP (The same implementation is in DCCP as well) .

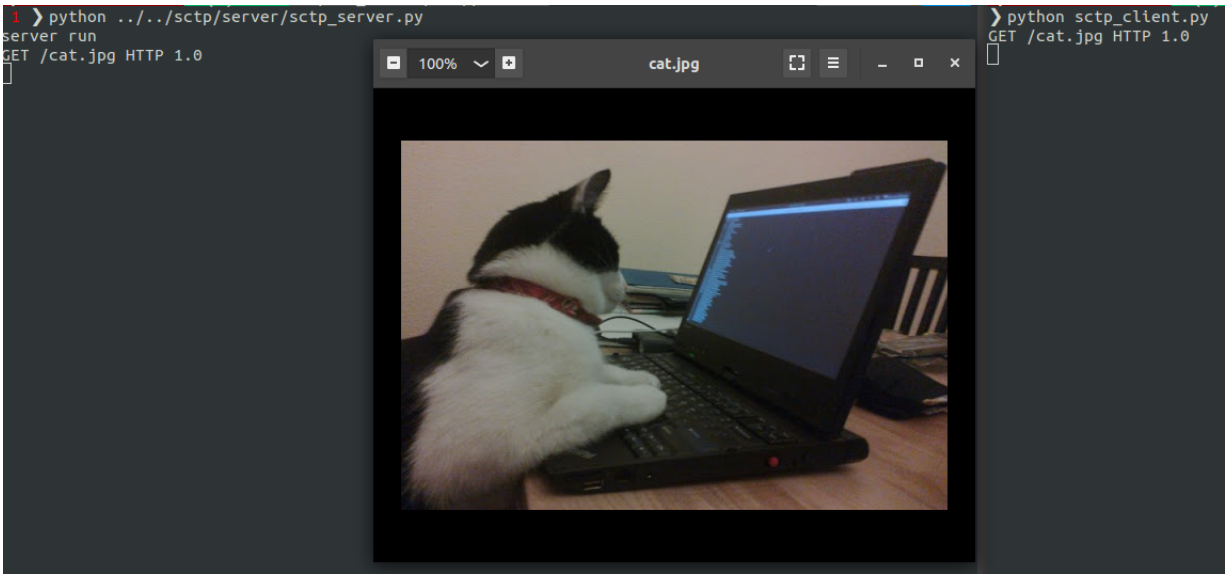

Figure 14: Getting an image on SCTP connection

Now we have received an image of a cute hacking cat. The preview of an image is done using the lightweight tool **eog**.// In the same way, we can also send videos, audio and even executable files. Of course in a commercial web server this

has to be specified in the ACCEPT header of the request but we believe that this implementation is out of the scope of this project. This connection was done using the SCTP client and server but in the same way it would work using the DCCP client and server pair.