

---

# 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 libscdp 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.

```

5 class SCTPclient:
6     def __init__(self,serv_ip,serv_port):
7
8         # Initialize the socket
9         sock = sctp.sctpsocket_tcp(socket.AF_INET)
10        sock.connect((serv_ip,serv_port))
11        self.socket = sock
12        self.dataSize = 8000
13
14    def makeRequest(self,req):
15        self.socket.sctp_send(req)
16        print(req)
17        headers = self.socket.recv(self.dataSize)
18        body = self.socket.recv(self.dataSize)
19        body = body.decode("utf-8")
20        print(body)
21        if "Not Found" not in body:
22            with open("out.html","w") as f:
23                f.write(body)
24            os.system("firefox out.html")
25    def close_connection(self):
26        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.

```

28 client = SCTPclient("127.0.0.1",8080)
29 client.makeRequest("GET / HTTP 1.0")
30 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.

```

6 class SCTPserver:
7     def __init__(self,serv_ip,serv_port,conns):
8
9         # Initialize the server parameters
10        self.ip = serv_ip
11        self.port = serv_port
12        self.connections = conns
13
14        # Create the server
15        sock = sctp.sctpsocket_tcp(socket.AF_INET)
16        sock.events.clear()
17        sock.bind((self.ip,self.port))
18        sock.listen(self.connections)
19
20        self.socket = sock
21        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:

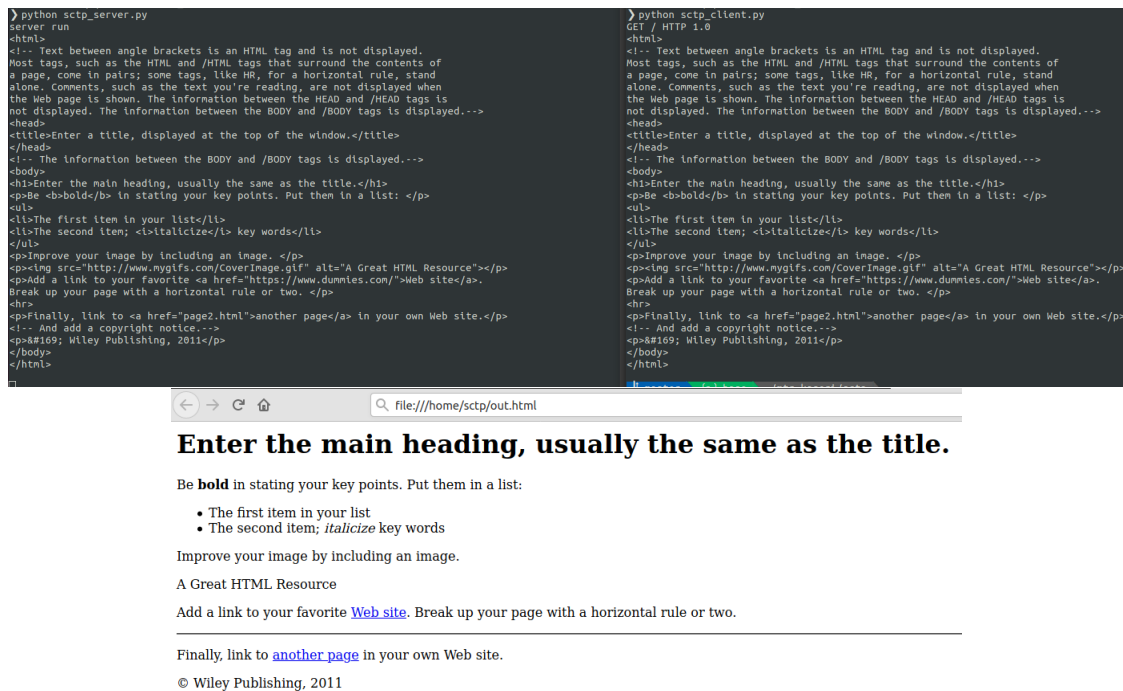


Figure 7: The HTTP GET request and reply

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

1	0.000000000	127.0.0.1	127.0.0.1	SCTP	106	INIT
2	0.000107553	127.0.0.1	127.0.0.1	SCTP	338	INIT_ACK
3	0.000116961	127.0.0.1	127.0.0.1	SCTP	310	COOKIE_ECHO
4	0.000218502	127.0.0.1	127.0.0.1	SCTP	50	COOKIE_ACK
5	0.000247116	127.0.0.1	127.0.0.1	SCTP	78	DATA
6	0.000292251	127.0.0.1	127.0.0.1	SCTP	62	SACK
7	0.000532324	127.0.0.1	127.0.0.1	SCTP	106	DATA
8	0.000547783	127.0.0.1	127.0.0.1	SCTP	62	SACK
9	0.000649865	127.0.0.1	127.0.0.1	SCTP	1354	DATA
10	0.099015229	127.0.0.1	127.0.0.1	SCTP	54	SHUTDOWN
11	0.099041449	127.0.0.1	127.0.0.1	SCTP	50	SHUTDOWN_ACK
12	0.099051978	127.0.0.1	127.0.0.1	SCTP	50	SHUTDOWN_COMPLETE

▶ Frame 9: 1354 bytes on wire (10832 bits), 1354 bytes captured (10832 bits) on interface 0	
▶ Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)	
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	
▼ Stream Control Transmission Protocol, Src Port: 8080 (8080), Dst Port: 43988 (43988)	
Source port: 8080	
Destination port: 43988	
Verification tag: 0x8fd2f9d1	
[Association index: 0]	
Checksum: 0x00000000 [unverified]	
[Checksum Status: Unverified]	
▶ DATA chunk (ordered, complete segment, TSN: 941027759, SID: 0, SSN: 1, PPID: 0, payload length: 1292 bytes)	
▼ Data (1292 bytes)	
Data: 3c68746d6c3e0a3c212d2d2054657874206265747765656e...	
[Length: 1292]	

0020	00 01 1f 90 ab d4 8f d2	f9 d1 00 00 00 00 00 03	.....<h
0030	05 1c 38 16 f1 af 00 00	00 01 00 00 00 00 3c 68	..8.....<h
0040	74 6d 6c 3e 0a 3c 21 2d	2d 20 54 65 78 74 20 62	tml><!-- Text b
0050	65 74 77 65 65 6e 20 61	6e 67 6c 65 20 62 72 61	etween a ngle bra
0060	63 6b 65 74 73 20 69 73	20 61 6e 20 48 54 4d 4c	ckets is an HTML
0070	20 74 61 67 20 61 6e 64	20 69 73 20 6e 6f 74 20	tag and is not
0080	64 60 73 70 64 70 65	64 70 64 70 65 74 20	displayed. Most

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

Figure 10.

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:

```

python dccp_server.py
server run
<!-- Text between angle brackets is an HTML tag and is not displayed.
Most tags, such as the HTML and /HTML tags that surround the contents of
a page, come in pairs; some tags, like HR, for a horizontal rule, stand
alone. Comments, such as the text you're reading, are not displayed when
the Web page is shown. The information between the HEAD and /HEAD tags is
not displayed. The information between the BODY and /BODY tags is displayed.-->
<head>
<title>Enter a title, displayed at the top of the window.</title>
</head>
<!-- The information between the BODY and /BODY tags is displayed.-->
<body>
<h1>Enter the main heading, usually the same as the title.</h1>
<p>Be <b>bold</b> in stating your key points. Put them in a list: </p>
<ul>
<li>The first item in your list</li>
<li>The second item; <i>italicize</i> key words</li>
</ul>
<p>Improve your image by including an image. </p>
<p></p>
<p>Add a link to your favorite <a href="https://www.dummies.com/">Web site</a>.
Break up your page with a horizontal rule or two. </p>
<hr>
<p>Finally, link to <a href="page2.html">another page</a> in your own Web site.</p>
<!-- And add a copyright notice.-->
<p>#169; Wiley Publishing, 2011</p>
</body>
</html>

```

```

python dccp_client.py
GET / HTTP 1.0
<!-- Text between angle brackets is an HTML tag and is not displayed.
Most tags, such as the HTML and /HTML tags that surround the contents of
a page, come in pairs; some tags, like HR, for a horizontal rule, stand
alone. Comments, such as the text you're reading, are not displayed when
the Web page is shown. The information between the HEAD and /HEAD tags is
not displayed. The information between the BODY and /BODY tags is displayed.-->
<head>
<title>Enter a title, displayed at the top of the window.</title>
</head>
<!-- The information between the BODY and /BODY tags is displayed.-->
<body>
<h1>Enter the main heading, usually the same as the title.</h1>
<p>Be <b>bold</b> in stating your key points. Put them in a list: </p>
<ul>
<li>The first item in your list</li>
<li>The second item; <i>italicize</i> key words</li>
</ul>
<p>Improve your image by including an image. </p>
<p></p>
<p>Add a link to your favorite <a href="https://www.dummies.com/">Web site</a>.
Break up your page with a horizontal rule or two. </p>
<hr>
<p>Finally, link to <a href="page2.html">another page</a> in your own Web site.</p>
<!-- And add a copyright notice.-->
<p>#169; Wiley Publishing, 2011</p>
</body>
</html>

```

file:///home/dccp/out.html

**Enter the main heading, usually the same as the title.**

Be **bold** in stating your key points. Put them in a list:

- The first item in your list
- The second item; *italicize* key words

Improve your image by including an image.

A Great HTML Resource

Add a link to your favorite [Web site](https://www.dummies.com/). Break up your page with a horizontal rule or two.

---

Finally, link to [another page](#) in your own Web site.

© Wiley Publishing, 2011

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

```

1 0.000000000 127.0.0.1 127.0.0.1 DCCP 90 39646 → 8080 [Request] Seq=236230977272215 (service=16777216)
2 0.000026421 127.0.0.1 127.0.0.1 DCCP 114 8080 → 39646 [Response] Seq=67566795221183 (Ack=236230977272215) (service=16777216)
3 0.000071367 127.0.0.1 127.0.0.1 DCCP 86 39646 → 8080 [Ack] Seq=236230977272216 (Ack=67566795221183)
4 0.000098208 127.0.0.1 127.0.0.1 DCCP 92 39646 → 8080 [DataAck] Seq=236230977272217 (Ack=67566795221183)
5 0.000351345 127.0.0.1 127.0.0.1 DCCP 183 8080 → 39646 [DataAck] Seq=67566795221184 (Ack=236230977272217)
6 0.000465975 127.0.0.1 127.0.0.1 DCCP 1354 8080 → 39646 [DataAck] Seq=67566795221185 (Ack=236230977272217)
7 0.000475163 127.0.0.1 127.0.0.1 DCCP 62 39646 → 8080 [Ack] Seq=236230977272218 (Ack=67566795221185)
8 0.000524177 127.0.0.1 127.0.0.1 DCCP 78 8080 → 39646 [CloseReq] Seq=67566795221186 (Ack=236230977272218)
9 0.106752966 127.0.0.1 127.0.0.1 DCCP 78 39646 → 8080 [Close] Seq=236230977272219 (Ack=67566795221186)
10 0.106771672 127.0.0.1 127.0.0.1 DCCP 82 8080 → 39646 [Reset] Seq=67566795221187 (Ack=236230977272219) (code=Closed)

Frame 6: 1354 bytes on wire (10832 bits), 1354 bytes captured (10832 bits) on interface 0
Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Datagram Congestion Control Protocol, Src Port: 8080, Dst Port: 39646 [DataAck] Seq=67566795221185
Source Port: 8080
Destination Port: 39646
Data Offset: 7
CCVal: 0
Checksum Coverage: 0
Checksum: 0x7f41 [correct]
[Checksum Status: Good]
Type: DataAck (4)
Extended Sequence Numbers: True
Sequence Number: 67566795221185
Acknowledgement Number: 236230977272217
Options: (4 bytes)
00 01 1f 90 9a de 07 00 7f 41 09 00 3d 73 9e d3 .....|A:~s~
e8 c1 00 00 d9 cf 7c 39 99 00 26 93 00 3c 08 .....|9~&~<h
74 6d 6c 3e 0a 3c 21 2d 2d 20 54 65 78 74 20 62 tml>~<|~Text b
65 74 77 65 65 6e 20 61 6e 67 6c 65 20 62 72 61 etween a ngle bra
63 6b 65 74 73 20 69 73 20 61 6e 20 48 54 4d 4c ckets is an HTML
20 74 61 67 20 61 6e 64 20 69 73 20 6f 6f 74 20 tag and is not
64 69 73 70 6c 61 79 65 64 2e 0a 4d 6f 73 74 20 displaye d..Most
74 61 67 73 2c 20 73 75 63 68 20 61 73 20 74 68 tags, su ch as th
65 20 48 54 4d 4c 20 61 6e 64 20 2f 48 54 4d 4c e HTML and /HTML

```

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) .

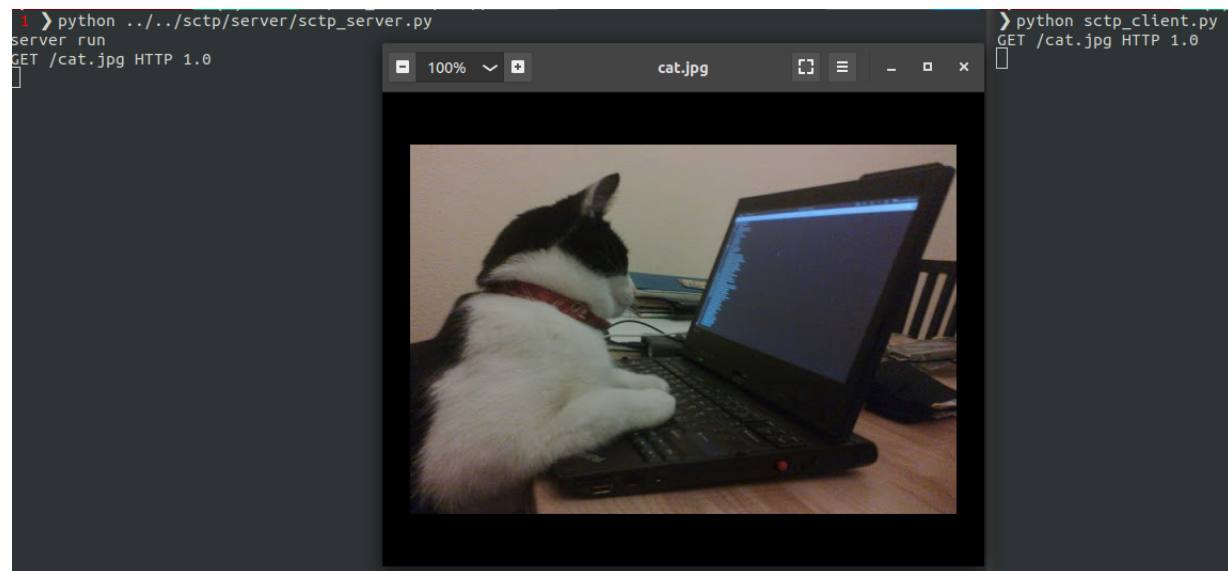


Figure 14: Getting an image on SCTP connection

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.