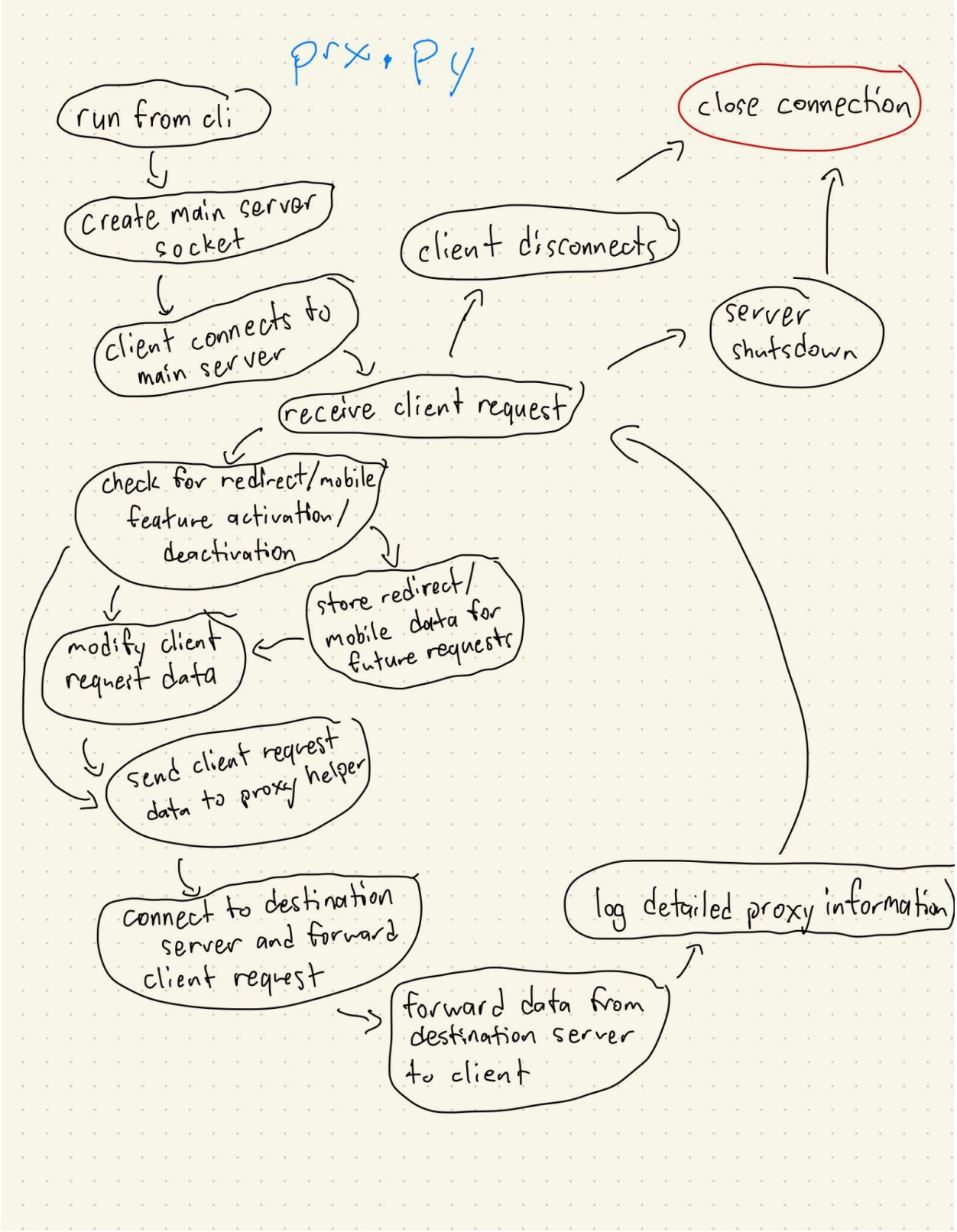
Introduction/Reference

I used Ubuntu 18.04 on a Virtual Machine on VMware Fusion on my Macbook Pro. I coded the project in Python 3.6.

For reference on sockets and select I used the following websites:

- https://docs.python.org/3/library/socket.html
- https://medium.com/pipedrive-engineering/socket-timeout-an-important-but-not-simple-issue-with-python-4bb3c58386b4
- https://docs.python.org/3/library/select.html

Flow Chart:


prx.py
run from cli
create main server socket
client connects to main server
receive client request
check for redirect/mobile feature activation/ deactivation
store redirect/ mobile data for future requests
modify client request data
send client request data to proxy helper
connect to destination server and forward client request
forward data from destination server to client
log detailed proxy information
client disconnects
close connection
server shutsdown

Black by Block Explanations:

```
def log(num, red, mob, cli_ip, cli_prt, req_f_cli, ua_f_cli, dst_dmn,
        dst_prt, req_t_dst, ua_t_dst, stat_dst, mime_type, mime_size):
    line1 = str(num) + " [" + red + "] Redirection [" + mob + "] Mobile\n"
    line2 = "[CLI connected to " + str(cli_ip) + ":" + str(cli_prt) + "]\n"
    line3 = "[CLI ==> PRX --- SRV]\n"
    line4 = "> " + req_f_cli + "\n"
    line5 = "> " + ua_f_cli + "\n"
    line6 = "[SRV connected to " + dst_dmn + ":" + str(dst_prt) + "]\n"
    line7 = "[CLI --- PRX ==> SRV]\n"
    line8 = "> " + req_t_dst + "\n"
    line9 = "> " + ua_t_dst + "\n"
    line10 = "[CLI --- PRX <== SRV]\n"
    line11 = "> " + stat_dst + "\n"
    line12 = "> " + mime_type + " " + mime_size + "bytes\n"
    line13 = "[CLI <== PRX --- SRV]\n"
    line14 = "> " + stat_dst + "\n"
    line15 = "> " + mime_type + " " + mime_size + "bytes\n"
    line16 = "[CLI disconnected]\n"
    line17 = "[SRV disconnected]\n"
    line18 = "-----------------------------------------------------------"
    print(line1 + line2 + line3 + line4 + line5 + line6 + line7 + line8 + line9 +
          line10 + line11 + line12 + line13 + line14 + line15 + line16 + line17 + line18)
```

This code block is a helper function that simply takes in all the required arguments to create the detailed server logging for a proxy request. It follows the format given in the project 3 guide.

```
def process_base_url(base):
    port_idx = base.find(":")
    url_res_idx = base.find("/")

    if url_res_idx == -1:
        url_res_idx = len(base)

    if port_idx == -1 or url_res_idx < port_idx:
        url_port = 80
        url = base[:url_res_idx]
    else:
        url_port = int((base[(port_idx + 1):])[:url_res_idx - port_idx - 1])
        url = base[:port_idx]
    return [url, url_port]
```

This code block is a helper function that takes the entire base url, for example danawa.com/?start_mobile, and trims it down to just the url and port. This is very useful for urls with ports in the url and other such edge cases.

```
def proxy(url, port, conn, data):
    # print("Proxy Start: ", "URL: ", url, " Port: ", port, " Data: ", data)
    prx = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    prx.settimeout(100)
    try:
        prx.connect((url, port))
        prx.send(data)

        first_reply = None
        while 1:
            try:
                reply = prx.recv(1024)
                # print(" REPLY PART: ", reply)
                if len(reply) > 0:
                    if not first_reply:
                        first_reply = reply
                    conn.send(reply)
                    chk_hdr = reply.decode(errors='ignore')
                    if chk_hdr.find("404") > 0 or chk_hdr.find("400") > 0:
                        break
                else:
                    break
            except Exception as tm:
                print("Inner Proxy: ", tm, " Data: ", data)
                break
        prx.close()
```

This code block is the first half of a helper function that does all the proxy requests. It takes the url we want to connect to, the port of the server we want to connect to (which is 80 by default for http), the client connection, and the client request data. It then creates a new socket for the proxied requests and sets a timeout of 100. It will try to connect to the destination server and forward all replies from it back to the client or if an exception occurs it will close the proxy and exit to the main program, or if an error occurs with receiving/sending data it will continue to print a detailed server log. It stores the first chunk of data received from the destination server so that we can process and log it later on. It also checks to see if there are any 404 or 400 status codes to quickly exit from the loop rather than waiting for a timeout, which was an annoying edge case I occasionally ran into.

```
    if not first_reply:
        raise Exception("Reply was empty.")

    # Print logs
    decode_reply = first_reply.decode(encoding='utf-8', errors='ignore')

    req_frm_cli = "GET " + orig_url
    ua_frm_cli = orig_ua

    if redirect:
        req_to_dst = "GET " + redirect_url
    else:
        req_to_dst = req_frm_cli

    if mobile:
        ua_to_dst = "Mozilla/5.0 (Android 7.0; Mobile; rv:54.0) Gecko/54.0 Firefox/54.0"
    else:
        ua_to_dst = ua_frm_cli

    stat_code_dst_start = decode_reply.find(" ") + 1
    stat_code_dst_end = decode_reply.find("\r\n", stat_code_dst_start)
    stat_code_dst = decode_reply[stat_code_dst_start:stat_code_dst_end]

    if decode_reply.find("Content-Type:") > 0:
        mime_typ_start = decode_reply.find("Content-Type: ") + 14
        mime_typ_end = decode_reply.find("\r\n", mime_typ_start)
        mime_typ = decode_reply[mime_typ_start:mime_typ_end].split(";")[0]
    else:
        mime_typ = "None"

    if decode_reply.find("Content-Length:") > 0:
        mime_sz_start = decode_reply.find("Content-Length: ") + 16
        mime_sz_end = decode_reply.find("\r\n", mime_sz_start)
        mime_sz = decode_reply[mime_sz_start:mime_sz_end]
    else:
        mime_sz = "0"

    log(request_counter, "O" if redirect else "X", "O" if mobile else "X", address[0], address[1],
        req_frm_cli, ua_frm_cli, url, port, req_to_dst, ua_to_dst, stat_code_dst, mime_typ, mime_sz)

except Exception as prx_error:
    print("Proxy: ", prx_error)
    prx.close()
```

This code block is the second part of the proxy helper function and it contains all the processing for the detailed server logs and calls the log helper function with all the processed data. For example, it decodes the first reply from the destination server and extracts important log information like the response status code, mime type, and mime size and stores them in variables to send to the log helper function to print out. The log function is called after every proxy operation where a critical exception doesn't occur. Some of the vars not defined here are defined at the high level in the main program.

```
# Take Port # from cli arguments
if len(sys.argv) != 2:
    raise Exception("ERROR! Usage: script, Port #")
cli_args = sys.argv

# Create srv socket
host, port = '127.0.0.1', int(cli_args[1])
srv_addr = (host, port)
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
srv.setblocking(0)
srv.bind(srv_addr)
srv.listen(1)

print("Starting proxy server on port " + str(port))
print("------------------------------------------------------------")
```

In this code block we take the port number from the command line args and create a socket for the proxy server to receive requests from the client. It then logs the server startup on the command line. We are only listening to one client for this project.

```
# Client vars
connections = [srv]
request_counter = 0
redirect = False
redirect_url = ""
orig_url = ""
mobile = False
orig_ua = ""
```

In this code block we store all our global client vars, which are used for the mobile and redirect features as well as storing our client/srv sockets.

```
while True:
    try:
        sr, sw, se = select.select(connections, [], [])
        for s in sr:
            if s is srv:
                # Accept client connection
                connection, address = s.accept()
                connections.append(connection)
            else:
                raw_data = s.recv(1024)
                data = raw_data.decode(encoding='utf-8', errors='ignore')
                if data:
                    try:
                        req_url_header = data.split('\n')[0]
                        if req_url_header.split(' ')[0] != 'GET':
                            continue
                        req_url = req_url_header.split(' ')[1]
                        http_idx = req_url.find("://")

                        if http_idx == -1:
                            base_url = req_url
                        else:
                            base_url = req_url[(http_idx + 3):]
```

In this code block we enter the main while loop which uses select to determine when a client wants to connect to the server and when it wants to send a request to the server to be proxied. The request is received then decoded so that we can process and find the base_url that contains the destination server domain and possibly extra information such as a specific port or the command to activate/deactivate the mobile and redirect features.

```
# Checks for special features
start_redir_feat = base_url.find("start_redirect")
stop_redir_feat = base_url.find("stop_redirect")
start_mobile_feat = base_url.find("start_mobile")
stop_mobile_feat = base_url.find("stop_mobile")
if start_redir_feat != -1:
    redirect = True
    redirect_url = base_url[start_redir_feat + 15:]
if stop_redir_feat != -1:
    redirect = False
    redirect_url = ""
    orig_url = ""
if start_mobile_feat != -1:
    mobile = True
if stop_mobile_feat != -1:
    mobile = False
    orig_ua = ""
```

In this code block we check if the commands for the mobile and redirect features are present in the base_url. If so, we store some information such as whether or not a feature is activated, the redirect url for future requests, and the original user agent. Some of these we use for our detailed logging.

```
# Check redirect
orig_url = base_url
if redirect:
    old_base_url = base_url
    old_url = process_base_url(base_url)[0]
    base_url = redirect_url
    data = data.replace(old_base_url, base_url)
    data = data.replace(old_url, base_url)

# Check mobile
user_agent_start = data.find("User-Agent:") + 12
user_agent_end = data.find("\r\n", user_agent_start) + 1
orig_ua = data[user_agent_start:user_agent_end - 1]
if mobile:
    data = data.replace(data[user_agent_start:user_agent_end],
                        'Mozilla/5.0 (Android 7.0; Mobile; rv:54.0) Gecko/54.0 Firefox/54.0\r')
    # data = data.replace(base_url, process_base_url(base_url)[0] + '/')
```

In this code block we do some additional work if the redirection or mobile feature are active. If the redirect feature is active we save the original url and then replace the base_url with the redirect_url that is saved when the start_redirect command is first used. We then update the client request data with the new base_url to reflect the redirection. If the mobile feature is active we save the original user agent then update the client request data with spoofed/fake mobile user agent. In this way, when we forward the client request to the destination server the client request data will now contain the modified information for redirection and mobile user agent.

```
# Check if persistent connection
if data.find("Connection:") > 0:
    conn_header_start = data.find("Connection:")
    conn_header_end = data.find("\r\n", conn_header_start)
    data = data.replace(data[conn_header_start: conn_header_end], "Connection: close")
if data.find("Keep-Alive:") > 0:
    ka_header_start = data.find("Keep-Alive:")
    ka_header_end = data.find("\r\n", ka_header_start) + 2
    data = data.replace(data[ka_header_start: ka_header_end], "")

proxy(process_base_url(base_url)[0], process_base_url(base_url)[1], s, data.encode())
request_counter += 1
```

In this code block we check to see if the client request data contains a persistent connection request. If it does, we modify the Connection header to close instead of keep alive. We also remove the entire Keep-Alive header as well. This way, the destination server does not receive a keep alive connection request. Once all the above is done, we send the url, port, client connection, and encoded data to the proxy helper function. Lastly, we initiate the request_counter upon a successful proxy operation.

```
                except Exception as e:
                    print("Server: ", e)
                    pass
            else:

                # Remove the disconnected client
                connections.remove(s)

                # Close the socket
                s.close()
except KeyboardInterrupt:
    srv.close()
    sys.exit()
```

In this code block, we log a critical exception if it occurs, which is very rare. We also remove the client from our connections list and close the client socket when the client no longer has any requests. In the case of KeyboardInterrupt, which signifies a manual shutdown of the server, we gracefully close the server socket and exit the program.