

Assignment 3

(Due: Sunday, Nov 27 midnight)

Assignment Setup:

Create a project in Eclipse called A3. Extract the assignment files into the A3 project, excluding the PDF instructions. You will notice that there are the following files:

- 1- `A3_template.py`: This is your solution file. You need to rename the file to `A3.py`. Also, you need to enter your credentials on top of the file.
- 2- `A3_test_server.py` and `A3_test_client.py`: These are the two testing files. You need to run the server file first, then the client.
- 3- `A3_ouptut_server.py` and `A3_output_client.py`: contain expected output at the server and client sides
- 4- There are four text files called: `"plaintext0.txt"`, `"plaintext1.txt"`, `"plaintext2.txt"`, `"plaintext3.txt"`.

Make sure that all your files are on the same directory level within the project.

Objectives:

- Build UDP server and client using UDP sockets
- Build socket communication using IPv6
- Use Object Oriented approach to building server/clients
- Implement application-level communication protocol over UDP
- Provide mechanisms for sequencing and fragmentation in UDP

Overview:

Your main objective in this assignment is to design a UDP server/client application called: UDP File Sharing Protocol (UFSP). This is a simple protocol, in which a client requests a file from the server. If the file is available, the client downloads the file.

General Configurations:

- The server and client use UDP IPv6 sockets.
- Assume that both the server and client will run on the same local host.
- The server and client use the default encoding of 'utf-8' in their message exchange. However, assume that only ASCII characters will be used.

- The server and client always use strings in the communication. For example, to send the number 9, it should be converted to a string then to bytes.
- Assume that the server and client buffers are synchronized. The buffer, which is a property in both classes, refer to the maximum number of bytes to expect in the datagram.
- Assume that there is no multithreading. Therefore, the server can only serve one client at a time.

Protocol:

- 1- The server is launched and listens on a specific port which is known to the clients. The server maintains a timer for inactivity. If the timer expires, the server closes
- 2- The client requests a file using the command: `"get <filename>"`
- 3- The server either sends: `"available"` or `"unavailable"`
 - a. If the client receives `"available"` it starts downloading the file
 - b. If the client receives `"unavailable"` it can either requests another file or closes.
- 4- If a file is available at the server, and after sending `"available"`, the server sends to the client the number of lines in the file in the following format: `"lines:<num_of_lines>"`¹.
- 5- Next, the server sends the file, each line in a separate datagram. The line is to be formatted as the following: `"<line_num>_<text>"`². The `line_num`, is the line number, starting from 0, and the `<text>` is the actual line text.
- 6- If a line is too long, the line should be fragmented into multiple datagrams. Each segment will be formatted as: `"<line_num>_<frag_num>_<text>"`
- 7- Once the server sends all lines, it sends `'EOF'`³.
 - a. If the client receives `'EOF'`. It triggers the reliability check.
 - b. If all lines were successfully received, the client either sends another file request, or closes.

¹ The client is not going to send an acknowledgement. We are not going to worry about the scenario if this specific message was lost.

² Because we are using the underscore character to separate the line number from the actual line, we are going to assume that the line does not have any underscores.

³ This is the string `'EOF'`, not the ASCII character EOF.

Client Configuration:

The `UFSP_Client` class has the following properties:

- `__socket` (socket): a private property that stores a UDP IPv6 socket.
- `__port` (int): a private property, passed as an optional keyword argument with default value 5000. Despite being a client socket, the socket should be bound to the given port.
- `name` (str): a public property, that is set by default to 'UFSP(Client:<port>)'.
- `buffer` (int): a public property, passed as an optional keyword argument with default value of 64
- `timeout` (float): a public property, passed as keyword argument with a default value of 5. The socket timeout should be configured to use this value.

The class defines the following methods:

`__str__(self):`

This method overloads the built-in `__str__` method and produces a string representation of the `UFSP_Client` objects in the following format:

```
<UFSP_Client name>:
    port = <port>
    buffer = <buffer>
    timeout = <timeout>
```

`get_file(self, filename, server):`

This method allows the client to request a specific file from the given server. The method sends the "get <filename>" command to the server.

- 1- If the file is available, the client waits for the number of lines and starts downloading the file
- 2- If the file is unavailable, the client exits the method

`__get_num_of_lines(self):`

This private methods should be invoked only, if a file is available. The method blocks awaiting for the number of lines. It discards any datagrams sent by the server other than the one that has the format: "lines:<num_lines>". The function returns the number of lines as an integer. If the client timeouts, the number of lines is -1.

```
download_file(self, filename, num_of_lines):
```

This method downloads a file from the server. It receives the number of lines, so it has a previous knowledge of how many lines to expect to receive.

The received file will be saved into another local file called: "filename_copy". For instance, if the file is: "test.txt", it will be saved as: "text_copy.txt".

The file lines are sent using the format: "<linenum>_<linetext>". The client should write to the output file the received lines in their respective order. If a datagram carrying a specific line ended up missing, an empty string should be stored.

Since the client is using UDP, the datagram may arrive out of order. Therefore, the client should maintain a mechanism to sequence the coming lines. Since the datagrams already have the line number, this should be trivial.

Note that a line could be fragmented. A fragmented line will have the following format: "<linenum>_<fragnum>_<linetext>". Assume that any fragments of the same line will arrive in order.

Since the client's buffer is synchronized with the server, the client is assured that any received datagram is not going to exceed its buffer value.

When the client receives the string 'EOF', it marks the end of download. The buffered lines will be written to the output file. As noted above, any missing datagrams will be replaced by an empty string.

```
close(self):
```

Since the `socket` is a private property, a mechanism is needed to close the client, which is provided by this method. Note that using the given implementation, once a client socket is closed, it cannot be started again. If we want to restart the socket, then another client object needs to be created.

Server Configuration:

The `UFSP_Server` class has the following properties:

- `port` (int): a public property, passed as a keyword argument with default value 4000
- `buffer` (int): a public property, passed as keyword argument with default value 64

- `timeout` (float): a public property, passed as keyword argument with a default value of 5
- `name` (str): a public property, that is set to `'UFSP(Server:<port>')`.
- `socket` (socket): a private property
- `files` (list): a private property that maintains the list of files available at the server. This is initialized to an empty list.

`__create_socket(self):`

This private method creates and configures the server socket. Since the socket is private and this method is private, this method should be invoked by the constructor. The socket should be a UDP IPv6. The socket should be bound to the given port. Make sure to configure the timeout parameter and the `SO_REUSEADDR` socket option.

`add_file(self, filename):`

This public method adds a given filename to the list of files available to the server.

`available_file(self, filename):`

This public method returns `True` if the given filename is available at the server and returns `False` otherwise.

`start(self):`

This public method allows the server to listen for incoming UDP datagrams, more specifically, the `"get <filename>"` requests. If the file is available, the server notifies the client and then invokes the `send_file` method. If it is unavailable, the server notifies the client that it is unavailable and then waits for other requests. The server maintains a timeout. If it expires, it closes the socket and exits.

`send_file(self, client, filename):`

This public method sends a given filename to the given server. It starts by sending the number of lines using the message: `"lines:<num>"`, followed by the file content.

The file content is broken into lines. Each line is sent in the following format: `"<line_num>_<line_text>"`. The line numbers reflect their order in the original file, starting from 0.

If a line is too long, then it needs to be fragmented. The fragmented line will appear as: "<line_num>_<frag_num>_<partial_line_text>". Again, the fragments will be numbered starting from 0.

Watch for the fact that the buffer size reflects the datagram's length including the prefixes, such as the line number and the fragment number and the in-between underscores. Therefore, the method has to pay special attention to when to initiate fragmentation. This should be handled by the next method.

When all lines have been sent, the server sends the string 'EOF' and exits the method.

```
fragment(self, line, counter):
```

This public method receives a line (extracted from a file), and a counter (representing the line number).

The method examines both parameters and decides if fragmentation is needed.

The method returns a list of fragments. If no fragmentation is needed, then the list will contain a single item, which is the given line, with the prefix <line_num>_.

If fragmentation is needed, then the list will contain all fragments with proper prefixes.

```
close(self):
```

Since the `socket` is a private property, a mechanism is needed to close the client, which is provided by this method. Note that using the given implementation, once a client socket is closed, it cannot be started again. If we want to allow such operation, then we need to make the `create_socket` method public, which is something we do not want to do at this assignment.