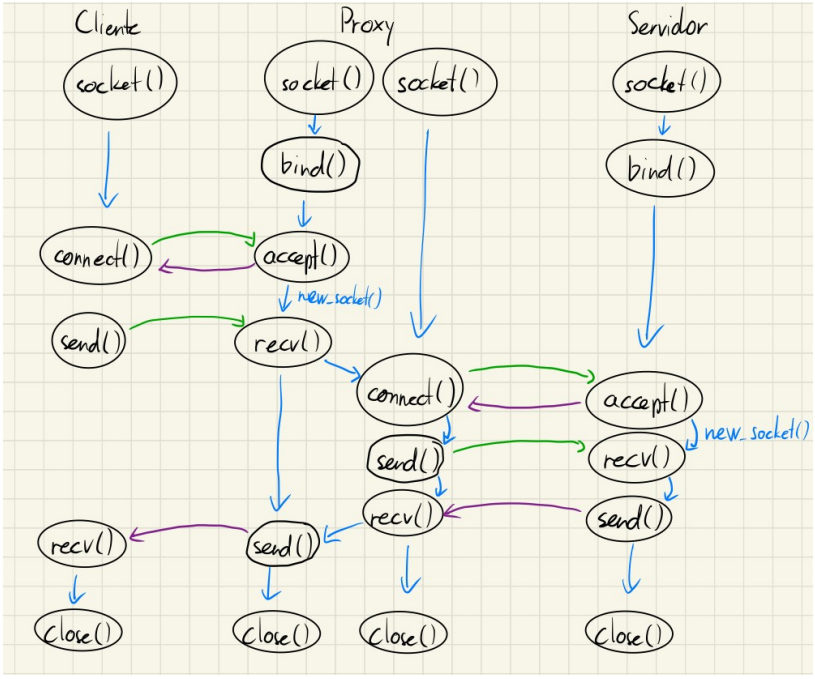
Informe Actividad 1:

Construir un proxy

Nombre: Vicente Olivares Gómez

Diagrama de funcionamiento de un proxy

Es necesario partir explicando que un proxy es un dispositivo intermedio entre un cliente y el servidor. Por lo que si el cliente quiere enviar un mensaje al servidor, el mensaje primero pasa por el proxy, al igual que cualquier respuesta de parte del servidor hacia el cliente. Como se trata de mensajes HTTP, que necesitan que el mensaje llegue en orden, se trabaja con sockets orientados a conexión. En la siguiente imagen se muestra un diagrama del funcionamiento de un proxy en el escenario de un cliente que envía una request HTTP al servidor, luego el servidor le envía una response HTTP y después cierran el canal de comunicación.



Siguiendo el orden del diagrama, se parte con la creación de 4 sockets: uno en el cliente que lo comunicará con el proxy, otro en el servidor que servirá para establecer la conexión con el proxy y en el proxy se crean dos, uno para establecer la conexión con el cliente y otro para la comunicación con el servidor. Luego, cuando el proxy invoca el método accept para aceptar la conexión solicitada por el cliente, se crea un nuevo socket. Este se usará para recibir mensajes del cliente y enviarle respuestas. De manera análoga se crea un nuevo socket cuando el servidor acepta la conexión solicitada por el proxy. Por lo tanto en todo el proceso indicado en el diagrama de la imagen se utilizan un total de 6 sockets.

Código de la Actividad

El código de Python de esta actividad está dividido en cuatro archivos: *constants.py*, *Message_HTTP.py*, *utilities.py* y *server.py.* Este último es el archivo que se debe ejecutar para hacer funcionar el proxy.

Archivo constants.py

Dentro de *constants.py* se encuentran definidas cuatro constantes utilizadas en el resto de archivos de código. Estas son PROXY_ADDRESS, BUFF_SIZE, HEAD_END_SEQ y HEAD_ENDLINE.

- PROXY_ADDRESS es la tupla con la dirección IP y el puerto utilizados como parámetros para invocar al método bind desde el socket del proxy que establece la conexión con el cliente.
- BUFF_SIZE un entero con el tamaño en bytes del buffer de recepción del socket utilizado para intercambiar mensajes con el cliente.
- HEAD_END_SEQ es el string “\r\n\r\n” que indica el término del head de un mensaje HTTP.
- HEAD_ENDLINE es el string “\r\n” que indica el final de un header de un mensaje HTTP.

Se creó este archivo con dos propósitos. En primer lugar, para tener fácil y rápido acceso a PROXY_ADDRESS y a BUFF_SIZE para modificarlos, puesto que a la hora de testear sobre el proxy fue necesario cambiar el puerto utilizado varias veces, pues este seguía tomado por una ejecución anterior del código. También, porque dentro de las pruebas a realizar sobre el código era necesario cambiar el tamaño del buffer de recepción del proxy. En segundo lugar, para tener guardadas las secuencias de término del head y de final de un header en un mensaje HTTP, puesto que estas secuencias son utilizadas con frecuencia dentro del código.

Archivo Message_HTTP.py

En este archivo se encuentra definida la clase Message_HTTP utilizada para representar a un mensaje HTTP. Tiene dos atributos: head, un diccionario que contiene todos los headers del mensaje, y body, un string con el contenido del mensaje. La clase tiene definidos los siguientes métodos:

- *__str__():* Retorna la representación en string del mensaje HTTP.
- *add_header(header_name : str, header_content : str):* Agrega el header recibido al head del mensaje, o lo modifica en caso de que ya exista un header con el mismo nombre recibido.
- *get_request_destiny_address():* Retorna la dirección a la que va dirigida el mensaje.
- *get_URI():* Retorna la URI de destino del mensaje.
- *replace_forbidden_words(forbidden_words_list : list):* Reemplaza las palabras indicadas en *forbidden_words_list* dentro del body del mensaje.

Se decidió crear esta clase para poder manipular y procesar los mensajes HTTP con mayor facilidad.

Archivo utilities.py

El archivo utilities.py contiene las definiciones de 12 funciones auxiliares utilizadas para facilitar el funcionamiento del proxy y mantener el código más limpio. El proposito de estas funciones será explicado a continuación.

Para garantizar la recepción completa de los mensajes HTTP enviados y recibidos por el proxy se definieron las siguientes cuatro funciones:

- *send_full_message(sender_socket : socket.socket, message : str) :* Envía el mensaje *message* por partes hasta que haya sido recibido completamente. Se realiza esto para cubrir los casos en que el tamaño del buffer del receptor sea menor que el tamaño del mensaje enviado.
- *receive_HTTP_message(receiver_socket : socket.socket, buff_size : int) :* Recibe un mensaje HTTP mediante el socket *receiver_socket*, retornando el mensaje como un string. La función se asegura de recibir completamente el head mediante la función *receive_HTTP_head* y luego recibe todo el body, utilizando el largo del body, obteniéndolo desde el header Content-Length, y la función *receive_HTTP_body.* Esto para cubrir el caso en que el mensaje HTTP no quepa completamente en el buffer del receptor del socket.
- *receive_HTTP_head(receiver_socket : socket.socket, buff_size : int) :* Recibe el head completo de un mensaje HTTP y retorna dos strings: un string con el head completo y otro string con la primera parte del body que puede haber sido recibida junto a la última parte del head. Para asegurarse de recibir el head completo se invoca el método recv de *receiver_socket* hasta encontrar la secuencia de doble salto de línea “\r\n\r\n” que marca el final del head en un mensaje HTTP.
- *receive_HTTP_body(receiver_socket : socket.socket, buff_size : int, body_size_bytes : int, body_first_part : str) :* Recibe el body completo de un mensaje HTTP y lo retorna como un string. Para asegurarse de recibir el body entero, se toma la primera parte del body *body_first_part*, que puede haber sido recibida junto a la última parte del head, y luego se invoca el método recv de *receiver_socket* hasta obtener la cantidad de bytes indicada en *body_size_bytes.*

Para que el código pueda ser ejecutado por la consola independiente sin importar el directorio en que esta se encuentre, se definió la función *getPath(fileName : str)* que retorna un string con la ruta absoluta del archivo *fileName.* La función está limitada para ser utilizada solo para los archivos ubicados en la misma carpeta que *utilities.py*.

Para que el proxy pueda procesar los mensajes HTTP recibidos, se definieron dos funciones, *parse_HTTP_message(http_message : str)* y *parse_HTTP_head(http_head : str).* La primera recibe un string con un mensaje HTTP y lo separa en su head y su body. Luego, entrega el head a la función *parse_HTTP_head*, que retorna un diccionario con los headers del head. Por último, se usa el diccionario de headers y el body para crear una instancia de la clase Message_HTTP y se retorna el objeto creado.

La función *create_HTTP_message(message : Message_HTTP)* retorna la representación en string del mensaje HTTP *message*.

La función *create_HTTP_response(request : Message_HTTP, body_html_name, response_code)* retorna una response HTTP como objeto de la clase Message_HTTP con el código de respuesta *response_code y el* archivo HTML *body_html_name* como body.

Para que el proxy pueda filtrar las páginas web a las que el cliente puede acceder, se definió *check_request(request : Message_HTTP, bloqued_URIs : list)* que verifica si la URI a la cual va dirigida la request del cliente está dentro de la lista de URIs bloqueadas. Para esto retorna un booleano indicando si la URI está permitida o no. Para los casos en que la URI no esté permitida, se definió la función *send_403_error(request : Message_HTTP, client_comm_socket : socket.socket),* que envía una respose HTTP al cliente con el error 403.

Como última función de este archivo, se definió *load_json(json_file_path : str)* que carga y retorna el contenido del archivo JSON ubicado en la ruta *json_file_path.*

Archivo server.py

Este es el principal archivo de código, ya que para hacer funcionar el servidor proxy, se debe correr el archivo *server.py*. Al ejecutarlo parte cargando un archivo JSON con las páginas bloqueadas, las palabras prohibidas y el nombre del usuario, utilizando la función *load_json* definida en *utilities.py*. Por defecto se utiliza el archivo *json_actividad_http.json*, sin embargo, si se desea utilizar otro archivo JSON, se debe ejecutar *server.py* desde la consola, entregando como primer parámetro el nombre del archivo JSON y su ruta como segundo parámetro. Además, es necesario que el nuevo archivo JSON tenga la misma estructura que *json_actividad_http.json*.

Luego de cargar el contenido desde el archivo JSON, se crea el socket *client_connection_socket* que servirá para establecer la conexión con el cliente y se le enlaza con la dirección IP y puerto *PROXY_ADDRESS* de *constants.py.* A partir de este punto se ingresa en un ciclo while infinito para aceptar conexiones de clientes.

Cada iteración del ciclo while comienza aceptando una conexión de un cliente, creando el socket *client_comm_socket* que será utilizado para intercambiar mensajes con el cliente. Luego, se recibe un mensaje HTTP completo mediante la función *receive_HTTP_message,* definida en *utilities.py.* Se recibe el mensaje con esta función en vez de hacerlo directamente con el método *recv* del socket *client_comm_socket,* para cubrir los casos en que el buffer de recepción del socket sea más pequeño que el mensaje HTTP completo, ya que si esto sucede, la parte del mensaje que no cupo en el buffer simplemente no es recibida, por lo que es necesario asegurarse de que se recibió completamente el mensaje. Después de recibir la request HTTP del cliente, esta se procesa y guarda como un objeto de la clase Message_HTTP mediante la función *parse_HTTP_message.*

Una vez el mensaje está guardado en la variable *client_http_msg*, el proxy revisa si la página a la cual está intentando acceder el cliente está permitida, usando la función *check_request.* Si la página no está permitida, se envía una response HTTP con el código de error 403, con la función *send_403_error.* En

caso contrario, se crea el socket *server_comm_socket* y se solicita establecer un canal con el servidor al que va dirigida la request del cliente. Cuando el canal ya está establecido, se agrega el header “X-ElQuePregunta” a la request con el nombre de usuario indicado en el archivo JSON cargado al inicio de la ejecución, se crea un string con la nueva request HTTP, se envía al servidor mediante la función *send_full_message* y se espera una respuesta del servidor. No se usa directamente el método send por razones análogas por las que se utilizó *receive_HTTP_message* en vez del método recv.

Al recibir la respuesta del servidor con *receive_HTTP_message*, se traduce la respuesta a un objeto Message_HTTP, se reemplazan las palabras del body de la response indicadas en el archivo JSON cargado al inicio por palabras indicadas en el mismo JSON. Después de cambiar las palabras prohibidas, se crea un string con la nueva response HTTP y se envía esta al cliente con la función *send_full_message*.

Por último, se cierran los sockets *client_comm_socket* y *server_comm_socket* para cortar los canales de comunicación del proxy con el cliente y con el servidor, y se comienza una nueva iteración del ciclo while, esperando una nueva solicitud de conexión de algún cliente.

Preguntas de la Actividad

¿Cómo sé si llegó el mensaje completo?

Para saber que llegó completamente un mensaje, se debe asegurar que llegaron todas las partes de este. En el caso de los mensajes HTTP, estos se componen de dos partes, la primera es el HEAD y la segunda el BODY. Por lo que hay que asegurarse de que el HEAD y el BODY llegaron completos para saber que el mensaje llegó completo.

¿Cómo sé que el HEAD llegó completo?

El HEAD tiene como secuencia de término un doble salto de línea “\r\n\r\n”, por lo que si se identifica esta secuencia dentro del mensaje total recibido, se sabrá que el HEAD llegó completo.

¿Cómo sé que el BODY llegó completo?

El BODY no tiene una secuencia de término como el HEAD, pero dentro del HEAD existe el header “Content-Length” que indica el largo del BODY en bytes. Entonces, al recibir los bytes indicados en este header, se habrá recibido el body completo.

¿Qué pasa si los headers no caben en mi buffer?

Si para recibir un mensaje HTTP se utiliza directamente el método *recv* del socket correspondiente, existe la posibilidad de que el buffer de recepción puede que no tenga el tamaño suficiente para contener todo el mensaje, todo un HEAD o incluso un header. Por lo tanto, al momento de recibir un mensaje es importante recibirlo por partes invocando varias veces el método recv, verificando que llegó completo guiándose por la secuencia de término del HEAD y por el largo del BODY indicado en el header “Content-Length”. Además es importante buscar la secuencia de término del head dentro del

mensaje total recibido y no por lo recibido dentro del buffer, pues la secuencia puede llegar en partes diferentes del mensaje.

Pruebas de la Actividad

Al testear el proxy con las pruebas indicadas en la actividad con el navegador Firefox, se verificó que no se puede acceder al sitio http://cc4303.bachmann.cl/secret y se recibe un error 403 desplegando el contenido de blocked_URI.html. También, se verificó que al acceder al sitio http://cc4303.bachmann.cl/ el contenido se modifica según los cambios introducidos en el header “X-ElQuePregunta”. Además, el proxy permite ingresar a los sitios http://cc4303.bachmann.cl/ y http://cc4303.bachmann.cl/replace, cambiando las palabras prohibidas.

Luego, al probar el proxy con buffer de tamaño menor a un mensaje pero mayor a la de un HEAD, y con un buffer de tamaño menor a la de un HEAD pero mayor a la de la start line, no se vieron cambios en el mensaje recibido.

Cabe destacar que luego de un tiempo corriendo, el proxy establece una conexión sin realizar alguna búsqueda en el navegador y al recibir este mensaje se arroja un error del tipo UnicodeDecodeError, indicando que hay un byte que no es posible decodificar en UTF-8.