Redes de Comunicaciones y Cómputo Distribuido

1er Cuatrimestre 2026

Taller - Protocolos HTTP y WebSocket

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Objetivo

Este taller tiene como objetivo comprender cómo funcionan los protocolos de nivel de aplicación. Se enfoca en entender la construcción de requests y responses, el intercambio de información entre cliente y servidor y la diferencia entre protocolos stateless (HTTP) y stateful (WebSocket).

Metodología de Trabajo

Para completar cada ejercicio o subconsigna, el alumno debe: primero crear un archivo con el request HTTP/WebSocket dentro de la carpeta `requests/`, luego implementar el código correspondiente en los archivos indicados, y finalmente ejecutar el request con netcat para validar que la implementación funciona correctamente. Sin crear el archivo de request y ejecutarlo con netcat, el ejercicio no se considera completo.

Herramientas

Netcat (comando `nc`) es una herramienta que permite enviar datos raw por TCP. Se utiliza para enviar requests HTTP manualmente y observar las respuestas. La sintaxis básica es:

```
cat archivo.http | nc localhost 8080
```

Este comando lee el archivo y lo envía byte por byte al servidor en localhost puerto 8080, mostrando la respuesta exacta que el servidor devuelve.

1 Ejercicio 1: Protocolo HTTP

1.1 Ejercicio 1.1

Completar la función `parse_request_line()` en el archivo `http_parser.py`. Esta función debe separar la primera línea de un HTTP request en sus tres componentes: método, path y versión del protocolo. La función recibe como entrada un string como `"GET /index.html HTTP/1.1"` y debe retornar un diccionario con las keys `"method"`, `"path"` y `"version"`.

Para completar este ejercicio, crear un archivo `requests/01_parse_test.http` con un request HTTP simple y ejecutarlo con netcat para verificar que el parser funciona correctamente.

1.2 Ejercicio 1.2

Completar la función `build_response()` en el archivo `http_parser.py`. Esta función formatea respuestas HTTP correctamente según el estándar. Recibe como parámetros el código de estado (200, 404, etc.), el contenido de la respuesta en bytes, y el tipo MIME del contenido. Debe construir un response HTTP con el siguiente formato:

```
HTTP/1.1 CODIGO MENSAJE\r\n
Content-Type: tipo\r\n
Content-Length: tamanio\r\n
\r\n
body
```

Utilizar el diccionario `STATUS_MESSAGES` para obtener el mensaje correspondiente al código de estado.

Para completar este ejercicio, crear un archivo `requests/01_response_test.http` con un request HTTP y ejecutarlo con netcat para verificar que las respuestas se formatean correctamente.

1.3 Ejercicio 1.3

Para completar este ejercicio, primero crear el archivo `requests/01_get.http` que solicite el recurso `/index.html`. Un GET request tiene la siguiente estructura:

```
METODO PATH VERSION
Host: valor
[linea vacia]
```

Luego, en el archivo `http_server.py`, completar las funciones `handle_client()` y `handle_get()` para manejar requests GET. Es necesario implementar ruteo según el método del request.

Finalmente, para validar que el ejercicio está completo, levantar el servidor con `python http_server.py` y en otra terminal ejecutar el request creado:

```
cat requests/01_get.http | nc localhost 8080
```

1.4 Ejercicio 1.4

Para completar este ejercicio, primero crear el archivo `requests/02_get_json.http` que solicite el recurso `/users.json`. Luego, en el archivo `http_server.py`, completar la función `get_content_type()` para determinar el Content-Type según la extensión del archivo. Modificar la función `handle_get()` para servir archivos del filesystem desde la carpeta `resources/`, retornando código 200 si el archivo existe y 404 si no existe.

Para validar que el ejercicio está completo, ejecutar el request creado con netcat:

```
cat requests/02_get_json.http | nc localhost 8080
```

1.5 Ejercicio 1.5

Completar la función `parse_request()` en el archivo `http_parser.py`. Esta función debe parsear un request HTTP completo incluyendo el body. La línea vacía separa los headers del body. La función debe retornar un diccionario con las keys `method`, `path`, `version`, y `body`.

Para completar este ejercicio, crear un archivo `requests/02_parse_complete.http` con un request que incluya headers y body, y ejecutarlo con netcat para verificar que el parser funciona correctamente.

1.6 Ejercicio 1.6

Para completar este ejercicio, primero crear el archivo `requests/03_post.http` con el siguiente contenido:

```
POST /users HTTP/1.1
Host: localhost:8080
Content-Length: [calcular bytes exactos]

{"name": "Charlie"}
```

El header `Content-Length` debe contener el número exacto de bytes del body. Luego, en el archivo `http_server.py`, completar la función `handle_post()` para procesar requests POST y retornar código 201. Actualizar el ruteo para manejar el método POST.

Finalmente, para validar que el ejercicio está completo, ejecutar el request creado con netcat:

```
cat requests/03_post.http | nc localhost 8080
```

1.7 Ejercicio 1.7

Para completar este ejercicio, primero crear el archivo `requests/04_head.http` igual a un GET request pero utilizando el método HEAD. Luego, en el archivo `http_server.py`, completar la función `handle_head()` para manejar requests HEAD, retornando solo headers sin body. Actualizar el ruteo para manejar el método HEAD.

Finalmente, para validar que el ejercicio está completo, ejecutar el request creado con netcat:

```
cat requests/04_head.http | nc localhost 8080
```

Se observarán los headers de la respuesta pero no el contenido del recurso.

1.8 Ejercicio 1.8

Para completar este ejercicio, primero crear el archivo `requests/05_headers.http` con múltiples headers como Host, User-Agent, Accept y Connection. Luego, en el archivo `http_parser.py`, completar la función `parse_headers()` y modificar `parse_request()` para incluir los headers en el diccionario retornado.

Finalmente, para validar que el ejercicio está completo, ejecutar el request creado con netcat:

```
cat requests/05_headers.http | nc localhost 8080
```

1.9 Ejercicio 1.9

Para completar este ejercicio, primero crear el archivo `requests/06_not_found.http` solicitando un recurso inexistente como `/noexiste.html`. La implementación del Ejercicio 1.4 ya debe manejar correctamente el error 404.

Finalmente, para validar que el ejercicio está completo, ejecutar el request creado con netcat:

```
cat requests/06_not_found.http | nc localhost 8080
```

2 Ejercicio 2: Protocolo WebSocket

2.1 Ejercicio 2.1

Para completar este ejercicio, primero crear el archivo `requests/handshake.http` con un HTTP request de upgrade WebSocket. El protocolo WebSocket comienza como un HTTP request normal con headers especiales: `Upgrade: websocket`, `Connection: Upgrade`, `Sec-WebSocket-Key` (valor aleatorio en Base64), y `Sec-WebSocket-Version: 13`. El servidor debe responder calculando el valor `Sec-WebSocket-Accept` según el algoritmo del RFC 6455.

En el archivo `websocket_frame.py`, completar la función `calculate_accept_key()` según el algoritmo del RFC 6455: concatenar el `Sec-WebSocket-Key` con el string mágico `"258EAFA5-E914-47DA-95CA-C` calcular SHA-1, y encodear en Base64.

En el archivo `websocket_server.py`, completar la función `handle_handshake()` para procesar el request de upgrade y responder con código 101. El response debe tener el siguiente formato:

```
HTTP/1.1 101 Switching Protocols\r\n
Upgrade: websocket\r\n
Connection: Upgrade\r\n
Sec-WebSocket-Accept: [accept key calculado]\r\n
\r\n
```

Finalmente, para validar que el ejercicio está completo, ejecutar el request creado con netcat:

```
cat requests/handshake.http | nc localhost 8080
```

2.2 Ejercicio 2.2

A diferencia de HTTP que utiliza texto plano, WebSocket empaqueta mensajes en frames binarios. La estructura de un frame es: Byte 0 contiene FIN (bit 7) y Opcode (bits 0-3), Byte 1 contiene MASK (bit 7) y Length (bits 0-6), los bytes 2-5 contienen la Masking key si MASK=1, y el resto contiene el Payload. Los opcodes importantes son `0x1` para mensaje de texto, `0x8` para cerrar conexión, `0x9` para ping, y `0xA` para pong.

En el archivo `websocket_frame.py`, completar las funciones `unmask_payload()`, `parse_frame()` y `build_frame()` para manejar frames WebSocket. Recordar que el cliente enmascara los frames pero el servidor no.

Para completar este ejercicio, probar la implementación usando el cliente WebSocket proporcionado en `client.html` o creando requests manuales con frames binarios.

2.3 Ejercicio 2.3

Para completar este ejercicio, primero asegurarse de que el handshake funciona correctamente ejecutando el request creado en el Ejercicio 2.1. Luego, en el archivo `websocket_server.py`, completar las funciones `handle_messages()` y `handle_client()` para implementar un echo server que reciba mensajes y los devuelva al cliente.

Finalmente, para validar que el ejercicio está completo, iniciar el servidor con `python websocket_server.py` abrir `client.html` en el navegador, escribir y enviar mensajes, y verificar que el servidor hace eco de los mensajes. El archivo `client.html` proporciona un cliente WebSocket JavaScript completo.