**Unidad 3. Protocolos de comunicación**

1. **Introducción:**

* **Protocolo de comunicación:** Conjunto de reglas que permiten que dos o más partes de un sistema distribuido puedan comunicarse y transmitir información. Existen protocolos de comunicación en todas las capas de TCP/IP con sus propias reglas y objetivos. Ejemplos:
* **Capa de Aplicación:** HTTP (navegación web), FTP (envío de archivos), SMTP (correo electrónico), DNS (nombres de dominio), DHCP (asignación dinámica de IP), etc.
* **Capa de Transporte:** TCP (confirmación de recepción) o UDP (sin confirmación).
* **Capa de Internet:** IP (enrutamiento y direccionamiento).
* **Acceso a Red:** Ethernet, WiFi, ARP, etc.

1. **Protocolos de la capa de Aplicación:**

* **Protocolos de comunicación de la capa de Aplicación:** Son los protocolos que utilizan las aplicaciones, que pueden ser ampliamente utilizados o específicos de ciertas aplicaciones. En cualquier caso, estos protocolos utilizan sockets que se encargan de la gestión de envío y recepción de la información, que se implementan en la capa inferior, la de Transporte. Los más protocolos estándar más utilizados en esta capa son:
* **HTTP (HyperText Transfer Protocol):** Utilizado por los navegadores web para realizar peticiones a los servidores web e interpretar su respuesta, así como por los servidores para tramitar las peticiones. Elementos de la comunicación vía HTTP:
* **Petición (request):** Al escribir una dirección web en el navegador se genera una petición que se transmite al servidor vía socket TCP con el siguiente formato:

**Dirección de búsqueda:** tutorialspoint.com/html/index.htm

A screenshot of a computer code

Description automatically generated

* **Línea de petición:** Incluye la siguiente información:
* **Método:** Tipo de petición, que puede tomar varios valores, entre ellos:
* **GET:** Para solicitar un recurso.
* **POST:** Para enviar datos, normalmente, a través de formularios.
* **Ruta:** La URL del recurso que se solicita.
* **Versión del protocolo:** Indica qué protocolo se está utilizando (HTTP/1.0, 1.1, 1.2, 2 o 3)
* **Cabeceras:** Son parámetros de la petición que aportan información adicional. Por ejemplo:
* **User-Agent:** Navegador utilizado, sistema operativo, etc. de origen.
* **Host:** Nombre de dominio o URL de destino.
* **Accept-Language:** Lista de idiomas aceptados.
* **Content-Type:** Indica el formato en que se envía la información del Cuerpo.
* **Cuerpo:** Contiene los datos a transmitir. En las peticiones de tipo GET suele estar vacío, mientras que en las peticiones POST incluye la información en texto plano, XML, JSON o binario, según lo indicado con Content-Type en la cabecera correspondiente.
* **Respuesta (response):** La respuesta del servidor tendría el siguiente formato:

**A screenshot of a computer

Description automatically generated**

* **Línea de respuesta:** Incluye la siguiente información:
* **Versión del protocolo:** Igual que en la petición.
* **Estado:** Código numérico de 3 dígitos que indica el estado de la operación, se clasifican según el primer número:
* **1xx:** Respuestas informativas.
* **2xx:** Respuestas satisfactorias. Por ejemplo: **200 OK** (petición correcta).
* **3xx:** Indica al navegador que debe realizar una acción, por ejemplo, redireccionar la petición.
* **4xx:** Errores de parte del cliente (navegador). Por ejemplo: **400 Bad Request** (la sintaxis de la petición no es correcta), **401 Unauthorized** (la petición necesita autorización) o **404 Not Found** (el recurso no existe).
* **5xx:** Errores de parte del servidor. Por ejemplo: **500 Internal Server Error** (error en el servidor).
* **Cabeceras:** Aportan información adicional a la respuesta:
* **Date:** Fecha y hora de envío.
* **Server:** Información del servidor.
* **Last-Modified:** Fecha de última modificación del recurso.
* **Content-Length:** Tamaño del cuerpo.
* **Content-Type:** Formato del cuerpo.
* **Cuerpo:** Contenido del mensaje. Puede ser HTML, CSS, JavaScript, imágenes en binario, etc. según lo especificado en la cabecera Content-Type.
* **FTP (File Transfer Protocol):** Utilizado para la transferencia de archivos entre cliente y servidor. Permite navegar a través de la estructura de directorios del servidor, ver los archivos disponibles y transmitir archivos de forma bilateral.
* **Tipos de peticiones:**
* Validación de usuarios.
* Información sobre el sistema de archivos.
* Modificación del sistema de archivos remoto.
* Transferencia de archivo en ambas direcciones.
* **Comandos:** Las peticiones se realizan a través de comandos que indican las operaciones a realizar. De forma similar a HTTP, el servidor contestará con un código de 3 dígitos que indican el resultado de la petición. Por ejemplo: 220 (conectado), 230 (autenticación correcta), 530 (autenticación incorrecta), etc. Algunos comandos usuales:
* **bye/quit/close:** Cierra la conexión.
* **cd:** Cambia el directorio.
* **dir/ls:** Lista los archivos del directorio cargados o disponibles para descarga.
* **get/mget:** Descarga un archivo o múltiples archivos
* **put/mput:** Sube uno o varios archivos.
* **open:** Abre una conexión FTP.
* **pwd:** Indica el directorio actual (working directory).
* **Canales de comunicación:** El protocolo establece canales diferentes según la operación que se va a ejecutar:
* **Canal de control:** Para realizar peticiones, la conexión permanece abierta mientras haya un cliente conectado, el servidor suele utilizar el puerto 21.
* **Canal de datos:** Se crea cuando es necesaria una transferencia y se cierra al terminar. Puede trabajar en dos modos:
* **Modo activo:** Prevé que el cliente indique al servidor la dirección IP y puerto al que realizar la transferencia y es el servidor el que se encarga de la conexión.
* **Modo pasivo:** El cliente solicita al servidor los datos de conexión y gestiona la conexión.
* **Seguridad:** Es un protocolo poco seguro porque no utiliza medios de encriptación de datos. Alternativas:
* **FTPS:** Extiende FTP con encriptación mediante SSL/TLS.
* **SFTP:** Protocolo diferente de FTP, pues se creó de cero con el protocolo SSH (Secure Shell, protocolo para el acceso remoto seguro, que utiliza el puerto 22) y ciertas características de FTP.
* **Tipo de Acceso:**
* **Anónimo:** Usuario especial (Anonymous o ftp) que solo tienen permiso para descargar archivos en un único directorio del servidor.
* **Autorizado:** Mediante usuario y contraseña, puede acceder a varios directorios y realizar operaciones.
* **Clientes y servidores:**
* **Servidores:** Filezilla Server
* **Clientes:** Línea de comandos, Filezilla Client, Navegadores, Explorador de archivos.
* **SMTP (Simple Mail Transfer Protocol):** Utilizado para el envío de correos electrónicos mediante comandos que permiten identificar origen y destino mediante una infraestructura de servidores.
* **Componentes:**
* **Servidores de correo:**
* **MTA (Mail Transfer Agent):** Reciben los correos y lo transmiten a otros MTA, funcionan como clientes y servidores, y utilizan el protocolo SMTP.
* **MDA (Mail Delivery Agent):** Reciben los correos de un MTA y lo depositan en el buzón destinatario. Utilizan los protocolos POP/IMAP como servidores.
* **Clientes de correo:**
* **MUA (Mail User Agent):** Permiten al usuario escribir, enviar, consultar o manipular sus buzones. Utilizan como clientes el protocolo SMTP para enviar correos y también como clientes los protocolos POP/IMAP para acceder a los buzones del servidor POP/IMAP (un MDA).
* **Protocolos:**
* **SMTP:** Envío de correo.
* **POP/IMAP:** Entrega de correo.
* **Esquema de funcionamiento:**

**A diagram of a computer system

Description automatically generated**

* **Protocolo SMTP:** Orientado a conexión que se basa en la transmisión de texto. Elementos:
* **Comandos del cliente:**
* **HELO:** El cliente se identifica a sí mismo.
* **MAIL FROM:** Identifica al remitente.
* **RCPT TO:** Establece el receptor.
* **DATA:** Indica el inicio del contenido del mensaje.
* **QUIT:** Termina la sesión.
* **Respuestas del servidor:**
* **220:** Servidor listo.
* **250:** Solicitud completada.
* **354:** Comienza el envío de datos.
* **221:** Cierre de comunicación.
* **Protocolo POP3 (Post Office Protocol):** Protocolo para la recepción de mensajes, que permite la descarga y borrado. Se diseñó para poder ver y manipular correos electrónicos de forma offline, pues se conecta al servidor, descarga localmente los archivos y los elimina del servidor. Elementos:
* **Comandos del cliente:**
* **STAT:** Devuelve el número de mensaje y tamaño total.
* **LIST:** Lista con identificación de mensajes disponibles.
* **RETR [índice]:** Devuelve el mensaje indicado por el índice.
* **DELE [índice]:** Elimina el mensaje indicado por el índice.
* **QUIT:** Termina la sesión y guarda cambios.
* **Respuestas del servidor:**
* **+OK**.
* **-ERR**
* **Protocolo IMAP (Internet Message Access Protocol):** Protocolo más avanzado que POP3 que permite:
* **Acceso remoto:** Permite acceder al correo desde cualquier dispositivo ya que los mensajes permanecen en un servidor en lugar de descargarse localmente.
* **Sincronización:** Los cambios en el cliente se aplican en el servidor (marcar como leído, eliminarlo, etc.), así como los nuevos correos se notifican inmediatamente porque IMAP está permanentemente conectado.
* **Múltiples carpetas:** Permite organizar los correos en carpetas.
* **Acceso selectivo:** Permite un acceso más rápido a los correos.
* **Acceso simultáneo:** Varias aplicaciones y dispositivos pueden acceder simultáneamente a los correos.
* **Capacidad de búsqueda:** Dispone de herramientas avanzadas de búsqueda de correos.
* **Agilidad:** Se transmite solo la cabecera para que el usuario decida si lo lee, y solo lo descarga si el usuario lo lee.

1. **Node.js:**

* **Concepto:** Entorno de ejecución de JavaScript para servidores. Creado en 2009, se popularizó porque permite trabajar en backend y frontend con JavaScript, así como permite manejar muchas conexiones bloqueantes de forma simultánea (chats, juegos online, streaming, etc.). Es clave para el desarrollo actual de aplicaciones web.
* **Características:**
* **API basada en eventos:** Sus operaciones son manejadas por eventos.
* **Hilo único:** Opera con un único hilo principal para todas las operaciones.
* **Operaciones E/S eficientes:** Utiliza un modelo asincrónico y no bloqueante.
* **Escalabilidad:** Permite muchas conexiones simultáneas con pocos recursos.
* **Ecosistema robusto:** Muchas librerías con npm (Node Package Manager).
* **Desarrollo rápido:** Usar un único lenguaje agiliza el desarrollo frontend y backend.
* **Tecnologías modernas:** GrahpQL, WebSockets, Express.js, etc.
* **Uso recomendado:**
* **Aplicaciones en tiempo real:** Chats, juegos, streaming, notificaciones, etc.
* **Aplicaciones basadas en eventos:** IoT o monitorización.
* **Aplicaciones web:** Para construir APIs RESTful y servicios web, con solicitudes concurrentes.
* **Desarrollo de prototipos:** El uso de JavaScript para frontend y backend agiliza el proceso.
* **Uso no recomendado:**
* **Uso intensivo de CPU:** Al utilizar un único hilo, solamente emplea un núcleo que estresaría al usarlo de forma intensiva.
* **Aplicaciones heredadas:** Las librerías de Node entran en conflicto con las heredadas.
* **Elementos para trabajar con Node:**
* **Node.js:** Programa open source, se descarga la versión 20.10 LTS. Comprobación de instalación: Abrir la consola (cmd) y comprobar la versión instalada: node -v.
* **Edición de archivo y ejecución en consola:** Crear archivo de texto plano de nombre hola.js (en consola: type nul > hola.js). Editar el archivo (notepad hola.js) y ponerle de contenido console.log(“Hola, mundo”). Guardar (Ctrl + G). Ejecutar el archivo (**node hola.js)** y mostrará el mensaje en pantalla
* **IDE para desarrollar con Node:** Se utilizará VS Code. Para abrir el terminal de ejecución de VS Code: Ctrl + ñ.
* **Módulos de Node.js:**
* **Concepto:** Bibliotecas, permiten reutilizar código en diferentes aplicaciones.
* **Módulos útiles:**
* **fs:** Trabajo con archivos y directorios.
* **path:** Trabajar con rutas.
* **os:** Obtener información del sistema operativo.
* **url:** Permite parsear direcciones URL.
* **http/https:** Crear servidores HTTPS/HTTPS.
* **net:** Permite trabajar con sockets TCP.
* **útil:** Utilidades de formato de textos, logging, etc.
* **Importar módulos:** Con el método require(<módulo>) se almacena la referencia en una variable (constante). Por ejemplo:

const fs = require(“fs”)

* **Crear módulos personalizados:** Del mismo modo, se pueden crear módulos e importarlos con require() para usar sus funciones. Sin embargo, para poder exportar las funciones se debe incluir en un objeto.Las funciones que no se incluyan en dicho objeto son privadas. Por ejemplo: las funciones suma y resta se incluyen en el objeto modules.exports, con los alias add y substract, pero la función multiplica no, por lo que será privada:

function suma (n1, n2){

    return n1 + n2;

}

function resta (n1, n2){

    return nº - n2;

}

function multiplica (n1, n2){

    return n1 \* n2;

}

module.exports = {

    add: suma,

    substact: resta

}

Luego, se importa el módulo recién creado indicando su path:

const mm = require(“./mi\_modulo.js”);

let res = mm.add(4, 5);

console.log(res);

* **Crear un servidor con Node:** Se importa el módulo “http” que dispone de la función **createServer()** que permite crear el servidor (devuelve un objeto de tipo Server). Esta función recibe dos parámetros:
* **options:** De tipo Object, indica las opciones de configuración del servidor.
* **requestListener:** De tipo Function, es la función que se ejecuta cuando se produce una petición (Evento “request”). El Evento “request” tiene, a su vez, dos objetos asociados:
* **request:** Instancia de http.ServerRequest que contiene toda la información de la petición que recibe.
* **response:** Instancia de http.ServerResponse que permite responder a la petición.

Para crear el requestListener, por tanto, se debe crear una función con dos parámetros (request y response), y almacenar esa función en una variable que se pasará a createServer().

Ejemplo:

const http = require("http");

const requestListener = function (request, response){

    console.log("Se ha producido una petición")

};

const server = http.createServer(requestListener);

server.listen(80);

**→ Servidor a la escucha:** La función listen(<puerto>) del objeto Server permite que el servidor quede a la escucha de nuevas conexiones. Ahora, al ejecutar el programa en la consola de VS Code y al abrir “localhost” desde cualquier navegador se producirá el evento y se imprimirá en consola que se ha producido una petición.

* **Obtener el valor de las cabeceras:** El parámetro request contiene toda la información de las cabeceras, por lo que es posible acceder a ella al recibir la petición:

const requestListener = function (request, response){

    console.log("Se ha producido una petición")

    console.log("URL: " + request.url)

    console.log("Método: " + request.method)

    console.log("Algunos campos de la cabecera:")

    console.log("Host: " + request.headers['host'])

    console.log("User-Agent: " + request.headers['user-agent'])

    console.log("Accept-Language: " + request.headers['accept-language'])

};

* **Responder a la petición:** El objeto response es el que permite responder a la petición, manipulando el contenido de la respuesta:

const requestListener = function (request, response){

    response.setHeader("Content-Type", "text/html")

    response.writeHead(200)

    response.write("<h1>Hola, mundo!</h1>")

    response.end()

};

**→ Más sencillo:** Es mejor unificar la respuesta utilizando solamente dos líneas:

response.writeHead(200, {'Content-Type' : 'text/html'})

response.end(("<h1>Hola, mundo!</h1>")

* **Lectura de ficheros:** Node dispone del módulo “fs” para acceder a directorios y ficheros, y de la función readFile() que permite leer ficheros. Esta función recibe tres parámetros:
* **path:** La ruta al fichero.
* **options:** Opciones de la lectura, como la codificación del archivo.
* **callback:** De tipo Function, se llama al finalizar la lectura del archivo y recibe, a su vez, dos parámetros:
* **err:** De tipo Error, incluye información si se produce un error.
* **data:** Contiene los datos leídos.

const fs = require("fs")

const lectura = function(err, data){

    if(err){

        console.log("Error: " + err);

        return;

    }

    console.log("Datos leídos: " + data);

}

fs.readFile('./prueba.txt', lectura);

* **Redireccionamiento:** Se puede indicar al navegador que un recurso ha cambiado de ubicación y se le indica el nuevo destino:

response.writeHead(301, {'Location' : '/estilos.css'});

response.end();

* **Query string:** Es una forma de enviar datos del cliente al servidor a través de la URL para transmitir parámetros, realizar consultas o personalizar la experiencia del usuario. Es la parte que viene en una URL después del signo ‘?’. Por ejemplo:

[http://localhost/index.html**?lang=es&color=black**](http://localhost/index.html?lang=es&color=black)

En concreto, los query string son pares clave – valor, en el caso anterior, para el idioma (lang=en) y para el color (color=black).

// Variable que almacena la url de la petición dividida en partes lógicas

    var url\_partes = url\_module.parse(request.url, true);

    var url\_path = url\_partes.pathname;

    var busqueda = url\_partes.query;

 // Si la url contiene un query string lang=en devuelve la página en inglés

        if (busqueda['lang'] == 'en'){

            // Separa la url para quitarle el /

            var recursos = url\_path.split('/');

            // Construye la url con ./en delante del nombre del recurso

            url\_path = path.join(\_\_dirname, `./en${recursos[1]}`);

        }

        // De lo contrario, la devuelve en español

        else {

            url\_path = path.join(\_\_dirname, url\_path);

        }