Mr. J. System

Sistemes Operatius – Curs 24/25

Alvaro Bello Garrido (alvaro.bello)

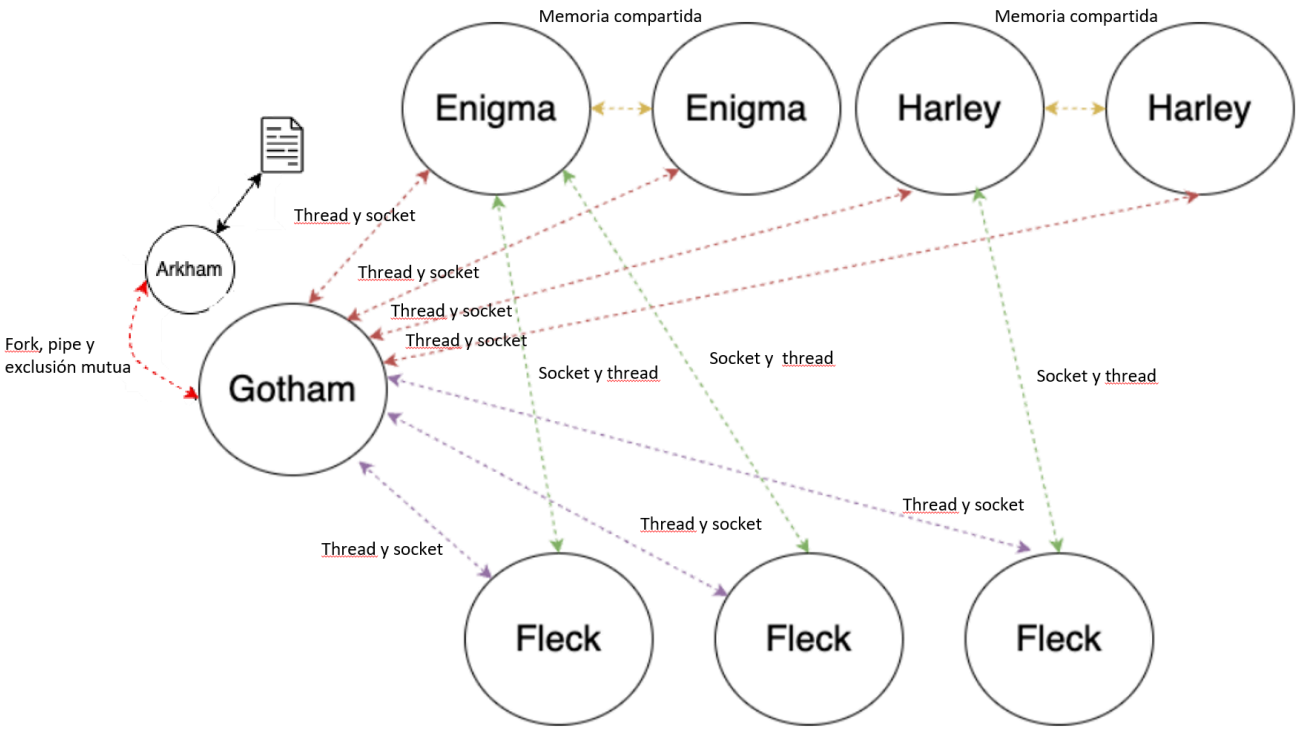
Carla Francos Molina (carla.francos)

24/06/2024

Índice

1. Diseño

1.1 Diagrama de procesos



Primero se inicia el proceso Gotham, que crea dos hilos: uno monta un servidor TCP para atender conexiones de los clientes Fleck y el otro monta otro servidor TCP para gestionar las conexiones de los Workers. Cada vez que llega un Fleck o un Worker, Gotham arranca un thread dedicado que mantiene viva la sesión hasta que la parte remota se desconecte (lo cual detectamos mediante tramas de tipo heartbeat). A continuación, los procesos Harley y Enigma se arrancan y conectan a Gotham, que registra su información (IP, puerto y tipo) en una lista interna de Workers disponibles, estas conexiones se establecerán utilizando sockets, y se utilizarán threads para manejar cada comunicación Gotham-Worker. También cabe mencionar que estas conexiones se mantendrán estables manteniendo a Worker a la espera de que se cierre Gotham y viceversa. Después de esto, el Worker principal se mantendrá a la espera (a través de otro socket Servidor) de que le lleguen peticiones de distorsión por parte de los Fleck.

A continuación Fleck se conectará con Gotham (utilizando también sockets para establecer la conexión y threads para manejar cada conexión Gotham-Fleck desde Gotham) y realizará peticiones

de distorsión. Dependiendo del archivo a distorsionar, Gotham le enviará los datos de un servidor Enigma o Harley a Fleck para que se conecte con él mediante sockets de nuevo y utilizando threads debido a que Fleck podrá realizar una petición de distorsión tipo Texto o una petición de distorsión tipo Media, por lo que se podrá conectar con un servidor Enigma y Harley al mismo tiempo (a su vez, el servidor Worker también administrará las peticiones de distorsión por parte de los Fleck mediante threads ya que pueden recibir peticiones de varios Flecks), y finalmente, Fleck enviará el archivo a distorsionar al Worker especificado para que este lo distorsione y se lo devuelva distorsionado.
La comunicación entre procesos Harley se implementa con memoria compartida para gestionar caidas de estos, de tal forma que si uno cae el siguiente sepa en qué punto de la subida se quedó el anterior.

Los Workers comenzarán conectándose a Gotham, el cual eligirá al primero de cada tipo (Media o Text) como Worler principal, y en caso de que un Worker principal caiga, eligirá a otro worker conectado aleatoriamente para que pase a ser el Worker principal. Después de haber un Worker principal, Fleck se conectará y le enviará una petición de distorsión a Gotham (si no hubiera ningún Worker del tipo de distorsión enviado, se cancelaría la distorsión notificando a Fleck de ello), Gotham le enviará a Fleck la información del Worker principal para la distorsión solicitada, y Fleck le enviará el archivo que desea distorsionar al Worker proporcionado por Gotham (utilizando varias tramas de 256 bytes para enviar el archivo dividido en partes), para que este lo distorsione y se lo envíe de vuelta de la misma forma. Al finalizar cada envío de un archivo entre Fleck-Worker, se deberá indicar mediante un mensaje Check_OK para que el receptor realice una comprobación mediante el MD5SUM de si el archivo ha sido enviado correctamente y lo notifique mediante otro mensaje Check_OK (en caso contrario se notificará mediante un mensaje Check_KO que no se envió correctamente, para que se envíe de nuevo el archivo). A su vez cabe indicar que antes de comenzar a enviar un archivo se indicará tanto el tamaño del archivo como el MD5SUM de este.
En caso de que caiga el Worker principal mientras se le estaba enviando una trama, Gotham elegirá un nuevo Worker principal. Cabe mencionar que el Worker principal que estaba recibiendo el archivo lo estaba almacenando en un directorio compartido. Seguidamente, Fleck solicitará a Gotham los datos de un nuevo Worker principal para seguir enviándole el archivo (ya que los Workers de un mismo tipo estarán en la misma máquina y por ende compartirán directorio). En caso de no haber más Workers del tipo de ‘media’ solicitado, se cancelará la distorsión.

Al caer el hilo principal (Worker principal) durante una distorsión, los demás hilos siguen vivos en el mismo espacio de memoria y pueden acceder inmediatamente a ese mismo buffer de estado para retomar el trabajo.

Para ello creamos un thread que establecerá la conexión con Gotham y que después de esto se mantendrá realizando peticiones HEARTBEAT constantemente, para que cuando se desconecte/caiga Gotham, se de cuenta Worker; y de mientras, paralelamente en el proceso principal de Worker, se cree un servidor para leer las peticiones entrantes de los Flecks mediante Threads.

Por último, para incorporar el sistema de logs (“Arkham”), en el propio Gotham.c se levanta un canal de comunicación adicional entre dos procesos mediante un pipe y un fork. Justo antes de arrancar los hilos de servidor, Gotham crea el pipe y hace fork. El proceso hijo (Arkham) cierra el extremo de escritura, duplica el descriptor de lectura (o lo lee directamente) y arranca su bucle principal de logging. Mientras tanto, el proceso padre (Gotham) cierra el extremo de lectura del pipe y guarda el descriptor de escritura en globalInfo. A partir de ahí, cada vez que ocurre un evento relevante (conexión o desconexión de Fleck/Worker, petición DISTORT, etc.), Gotham invoca log_event(descr), que empaqueta la cadena descriptiva en una trama, y escribe siempre en el pipe.

1.2 Estructuras de datos y justificación

Las diferentes estructuras de datos utilizadas para el desarrollo de la práctica, han sido:

GothamConfig:

```
typedef struct {
    char* ip_fleck;   // Dirección IP para Fleck
    int port_fleck;   // Puerto para Fleck
    char* ip_workers; // Dirección IP para Harley/Enigma
    int port_workers; // Puerto para Harley/Enigma
} GothamConfig;
```

La estructura GothamConfig almacena información que puede variar dependiendo del entorno, como direcciones IP y puertos, para garantizar que Gotham pueda conectarse correctamente tanto con Fleck como con Harley/Enigma.
Las direcciones IP (ip_fleck, ip_workers) son dinámicas, por lo que se utilizan punteros a char (char*). Esto permite gestionar direcciones IP de longitud variable sin desperdiciar memoria y los puertos (port_fleck, port_workers) son enteros (int) porque representan valores numéricos pequeños, lo que optimiza el uso de memoria.

HarleyConfig/EnigmaConfig:

```
typedef struct {
    char *ip_gotham;     // Dirección IP para Gotham (dinámico)
    int port_gotham;     // Puerto para Gotham
    char *ip_fleck;      // Dirección IP para Fleck (dinámico)
    int port_fleck;      // Puerto para Fleck
    char *worker_dir; // Directorio de trabajo para Enigma/Harley (dinámico)
    char *worker_type;   // Tipo de worker ("Media" o "Text") (dinámico)
} Enigma_HarleyConfig;
```

La configuración de HarleyConfig/EnigmaConfig incluye información sobre cómo conectarse con Gotham y Fleck, así como detalles específicos sobre su operación (directorio y tipo de worker). Esta

estructura es fundamental para permitir flexibilidad en la configuración del comportamiento de Harley/Enigma.

Las direcciones IP (ip_gotham, ip_fleck) y el directorio (worker_dir) son cadenas de texto dinámico, por lo que se usa char* y los puertos (port_gotham, port_worker) son valores numéricos pequeños, por lo que se representan con int.

El tipo de worker (worker_type) también es dinámico, ya que puede ser "Media" o "Text", lo que justifica el uso de char* para representar estas cadenas.

FleckConfig:

```
typedef struct {
    char *username;   // Nombre de usuario
    char *user_dir;   // Directorio de usuario
    char *gotham_ip;  // Dirección IP de Gotham
    int gotham_port;  // Puerto de Gotham
} FleckConfig;
```

La estructura FleckConfig está diseñada para almacenar la información necesaria para que el componente Fleck pueda interactuar con el sistema, especialmente con Gotham. Contiene los datos del usuario, el directorio de trabajo y la configuración de conexión a Gotham. La estructura permite que Fleck maneje datos dinámicos como el nombre de usuario y la dirección IP de Gotham de manera eficiente.

Server:

```
typedef struct {
    int server_fd;               // File descriptor del socket del servidor
    struct sockaddr_in address;  // Dirección del servidor
    int port;                    // Puerto del servidor
    int max_connections;         // Número máximo de conexiones en espera
} Server;
```

La estructura Server está diseñada para gestionar las conexiones de red de Gotham de manera eficiente. Su objetivo principal es almacenar toda la información necesaria para crear y administrar

un socket de servidor, permitiendo a Gotham aceptar conexiones de clientes como Fleck y Harley/Enigma.

WorkerFleck:

```
typedef struct {
    char* IP;   // IP de Worker
    char* Port;  // Puerto de Worker
    char* workerType;
    int socket_fd;
    int status; // Estado de la distorsión en marcha [0-100%]
} WorkerFleck;
```

Esta estructura sirve para gestionar y almacenar información sobre los workers conectados a Gotham. Se usa para supervisar múltiples conexiones simultáneas, saber el progreso de las tareas asignadas, y realizar un seguimiento de los Workers conectados a Gotham.

Worker:

```
typedef struct {
    char* workerType;
    char* IP;
    char* Port;// Puerto del servidor de Worker (utilizado para recibir
//conexiones de Flecks)
    int socket_fd;
} Worker;
```

Worker modela un nodo de procesamiento (“worker”) conectado a Gotham. Se utiliza para registrar y diferenciar cada instancia de worker que se conecta (Enigma o Harley), manteniendo la información necesaria para establecer la comunicación de red con él. A partir de este struct, el servidor sabe a qué worker dirigir tareas y por dónde esperar sus resultados.

GlobalInfoGotham:

```
typedef struct {
   GothamConfig* config;    // Global para poder liberarse con SIGINT
    Server* server_fleck;
    Server* server_worker;

    // WORKER
  Worker* workers;// Array donde almacenaremos los Workers conectados a

//Gotham
  int num_workers;
  int enigma_pworker_index; // Indice del worker(Enigma) principal
  int harley_pworker_index; // Indice del worker(Harley) principal

// Mutex para cuando se modifiquen o lean las variables globales relacionadas
con workers
   pthread_mutex_t worker_mutex;

    // FLECK
    int* fleck_sockets;         //Lista de sockets de flecks
    int num_flecks;
    pthread_mutex_t fleck_mutex;

    // Threads de Workers y Fleck
    pthread_t workers_server_thread;
    pthread_t fleck_server_thread;

    // Array de subthreads
    pthread_t* subthreads; // Threads generados por cada conexión Wk o Fleck
    int num_subthreads;
    pthread_mutex_t subthreads_mutex;
} GlobalInfoGotham;
```

Esta estructura agrupa todo el estado compartido del servidor Gotham. Aquí se almacenan las configuraciones, los servidores de escucha (flecks y workers), los arrays de conexiones activas y sus contadores, así como los hilos que atienden nuevas conexiones y subprocesos de trabajo. Además

incluye los mutex necesarios para sincronizar accesos concurrentes. En resumen, es el “centro de mando” que mantiene coherente y accesible toda la información de red y concurrencia.

ThreadArgsGotham:

```
typedef struct {
    int socket_connection;
    GlobalInfoGotham* global_info;
} ThreadArgsGotham;
```

Cuando Gotham crea un nuevo hilo para atender una conexión entrante, empaqueta en ThreadArgsGotham el descriptor de socket y un puntero al estado global. De esta forma, cada hilo dispone tanto de la conexión específica que va a gestionar como de acceso a la información y recursos compartidos, sin necesidad de variables globales dispersas.

TramaResult:

```
typedef struct {
    char type;
    char *timestamp;   // El timestamp en formato de string
    char *data;        // Los datos del mensaje
} TramaResult;
```

La estructura TramaResult sirve para encapsular cualquier “trama” o mensaje que se intercambia entre los componentes del sistema. Su propósito principal es agrupar de forma genérica el tipo de trama (p. ej. solicitud, respuesta, notificación), la marca temporal y el contenido asociado, de modo que cualquier módulo que reciba o envíe datos pueda manejar este único objeto.

DistortInfo:

```
typedef struct {
    char* username;
    char* filename;
    char* distortion_factor;
    WorkerFleck** worker_ptr; // Puntero a WorkerFleck* para poder ponerlo en NULL
} DistortInfo;
```

Esta estructura se usa para describir una solicitud de “distorsión” de un fichero por parte de un usuario. Agrupa los datos de identificación (usuario y archivo), el nivel o factor de distorsión, y un puntero indirecto al worker asignado. Con DistortInfo, la lógica de procesamiento conoce todo lo necesario para lanzar, supervisar y, al finalizar, liberar o resetear el worker responsable.

ClientThread:

```
// Estructura para manejar las conexiones de los Flecks
typedef struct {
    int socket;
    pthread_t thread_id;
    int active;
} ClientThread;
```

La estructura ClientThread está diseñada para registrar y gestionar cada conexión individual de un cliente “Fleck” en Gotham. Contiene el descriptor de socket necesario para intercambiar datos, el identificador del hilo (pthread_t) creado para atender dicha conexión y un flag active que indica si la sesión sigue en curso o debe cerrarse. De esta manera el servidor puede crear, monitorizar y cerrar de forma ordenada cada hilo y su socket asociado, manteniendo un control claro sobre todas las conexiones de Flecks activas y evitando fugas de recursos.

SharedData:

```
typedef struct {
    int transfer_flag;  // 0=recibiendo, 1=enviando, 2=enviando
    long total_bytes_received;
} SharedData;
```

La estructura SharedData centraliza el control y seguimiento de una operación de transferencia de datos compartida entre varios hilos o componentes.

1.3 Recursos del sistema utilizados

Se han utilizado varios recursos para poder realizar la práctica. Cada uno de ellos tiene una razón específica para la que se está utilizando:

¿Por qué threads y no forks para las conexiones de Worker→Gotham, o Fleck→Worker?

Por ligereza, eficiencia y simplicidad, cada nueva conexión (tanto Fleck→Gotham como Worker→Gotham, o Fleck→Worker) se atiende con un hilo. Crear un hilo es mucho más rápido y consume menos memoria que crear un proceso completo.

Todos los hilos dentro de un mismo proceso Gotham comparten la lista de Workers disponibles, las estructuras de datos de clientes conectados, etc., sin necesidad de mecanismos adicionales (memoria compartida, pipes).

Por último, por tema de escalabilidad, con cientos de clientes simultáneos, cientos de hilos es mucho más manejable que cientos de procesos.

¿Por qué sockets y no memoria compartida o pipes para Worker→Gotham, o Fleck→Worker?

Los sockets es uno de los recursos más utilizados en la práctica. La comunicación entre procesos es prácticamente en todo momento, a través de los sockets. Los sockets permiten la comunicación entre procesos que se están ejecutando desde diferentes máquina físicas.

Por lo tanto, como nuestro sistema podría desplegarse con Fleck, Gotham y Workers en máquinas diferentes hacemos uso de sockets.

Además este, ofrece retransmisión automática de paquetes perdidos, control de congestión y segmentación de datos, lo cual simplifica la transferencia fiable de archivos grandes en tramas de 256 B.

Por último, al basarnos en un protocolo de trama fija, nos basta con send()/recv() sobre sockets para serializar mensajes, MD5, tamaños, etc.

Mutex:

Durante la práctica se utilizan varios mutex para garantizar la exclusión mutua y la sincronización de los procesos con threads. Por un lado, se utilizan para hacer que, cuando un proceso (sea el thread que sea), quiera acceder a una variable compartida (o variable global), ningún otro proceso pueda acceder a ella durante ese momento. Por otro lado, también se ha utilizado mutex para bloquear y desbloquear procesos desde otros procesos.

Pipes:

Usamos un pipe para que Gotham y Arkham se comuniquen de manera sencilla y ordenada: Gotham escribe en el extremo de salida cada “trama” de log y Arkham lee del extremo de entrada. De ese modo, todos los mensajes fluyen en secuencia y no hay posibilidad de que dos escritores intenten acceder al mismo tiempo a la memoria intermedia.

Además, al diseñar Arkham como único proceso encargado de abrir y escribir en logs.txt (en modo append), garantizamos la **exclusión mutua** sin necesidad de semáforos o mutexes adicionales: solo él escribe en el fichero, así que no hay riesgo de intercalado de bytes o corrupciones de datos.

¿Por qué forks y no threads para Arkham?

Optamos por un fork para arrancar Arkham en lugar de un hilo porque en este caso queremos un proceso completamente independiente, con espacio de direcciones aislado. Si lo hubiésemos hecho con threads, ambos compartirían memoria, ficheros y estado, complicando la sincronización y potencialmente propagando fallos de uno al otro. Con fork, Arkham puede fallar, abortar o cerrarse sin colapsar el proceso principal de Gotham, y el sistema operativo se encarga de limpiar sus recursos aislados. Esto simplifica el diseño y mejora la robustez del sistema de logging.

Notificaciones vía socket y heartbeat

En lugar de colas de mensajes, Gotham envía por TCP/IP una trama especial (TYPE_ASSIGN_NEW_PRINCIPAL) a todos los Workers cuando detecta la caída del principal.

Cada hilo de Worker ya tiene abierto un socket estable con Gotham para responder heartbeats; sobre ese mismo canal espera la orden de “eres tú el nuevo principal”.

De este modo no necesitamos otro mecanismo IPC, y garantizamos que la señal de failover llegue de forma fiable y ordenada.

Persistencia de conexiones y rendimiento

Conexiones estables: Fleck mantiene viva la conexión TCP tanto con Gotham como con su Worker asignado en caso de estar realizando una distorsión. En caso de que caiga su worker asignado, le preguntará a Gotham de nuevo si hay otro Worker disponible, y en caso afirmativo, Gotham contesta con los datos de este para que Fleck se conecte y continúe la distorsión con él.

Escalabilidad lineal: Al aumentar el número de Flecks, Gotham los atiende en paralelo con hilos; cada Worker atiende múltiples clientes simultáneos, también con hilos.

1.4 Opcionales implementados

Las tramas HEARTBEAT son mensajes periódicos enviados entre componentes de un sistema distribuido para verificar su disponibilidad. Su propósito principal es detectar desconexiones o fallos en los componentes y mantener las conexiones activas.

En este proyecto, hemos implementado el envío de tramas HEARTBEAT entre los Workers y Flecks hacia Gotham. Cada componente envía una trama HEARTBEAT a intervalos regulares, que incluye su identificador.

2. Problemas observados y cómo se han solucionado

Durante el desarrollo de la práctica hemos topado con diferentes problemas:

- Problemas de sincronización entre procesos: En varias ocasiones nos hemos enfrentado a problemas de sincronización entre procesos, y esto hacía que ambos procesos se quedaran bloqueados en distintos puntos; uno esperando una señal que ya se había enviado o similares. Ha sido un problema, sobre todo, a la hora de crear nuevos sockets y hacer que éstos se conecten. A veces, un proceso tenía el socket preparado para enviar señales de connect, pero por la otra parte aún no estaba preparada para aceptar peticiones.
Para solucionar estos problemas, hemos hecho uso de sockets y mutex para poder sincronizar ambos procesos y garantizar que estuvieran preparados uno por otro en todo momento.

- Cierres prematuros de conexión: En varios puntos cerrábamos la conexión tan pronto como llegaba el handshake (“conexión aceptada”), sin esperar al envío o recepción de los datos. Esto cortaba la comunicación antes de tiempo.
Para solucionarlo reestructuramos el flujo para que el socket sólo se cierre al finalizar todo el protocolo de distorsión (handshake → transferencia de datos → comprobación final).

- Acceso concurrente a listas de Workers: Protegido con pthread_mutex_lock/unlock para evitar que dos hilos modifiquen a la vez la lista (registro o baja de un Worker).

- Problemas de gestión de las variables: En muchas ocasiones, nos hemos encontrado con problemas con la manipulación de cadenas. Hemos tenido desbordamientos de buffer, asignación de memoria por asprintf que se provoca fugas de memoria o en ocasiones problemas de double free, problemas con el uso de strtok porque modificaba la cadena original, y problemas a la hora de asegurar el Null Terminator ('\0') de todas las cadenas que se utilizan durante la práctica.
Para solucionar estos problemas, hemos tenido que trabajar de forma meticulosa, vigilando en todo momento las cadenas que generamos, haciendo uso de printf y write(1,..), para poder tener un seguimiento de la evolución de éstas durante la ejecución de programa y poder encontrar dónde se encontraban los errores.

3. Estimación temporal

Categorías	**Horas**
Investigación	4 horas
Diseño	5 horas
Implementación	38 horas
Testing	30 horas
Documentación	10 horas
TOTAL	**82 horas**

Al principio de la práctica, el tiempo dedicado a la investigación y al diseño no fue muy elevado. Pero nos dimos cuenta que era una implementación de código más complicada de lo que esperábamos, lo que nos llevó a dedicarle muchas más horas. Del mismo modo, el testing ha requerido un gran número de horas, ya que encontramos numerosos problemas, especialmente relacionados con la gestión de la memoria.

4. Conclusiones y propuestas de mejora

Tras la realización de esta práctica hemos podido consolidar y ampliar considerablemente nuestros conocimientos sobre comunicación entre procesos, gestión concurrente de recursos y programación en sistemas distribuidos utilizando threads, sockets, memoria compartida y pipes. Ha resultado especialmente enriquecedor enfrentar problemas reales relacionados con la sincronización y manejo eficiente de múltiples conexiones simultáneas, lo que nos ha permitido comprender mejor la complejidad inherente a los sistemas concurrentes.

Uno de los mayores aprendizajes ha sido la importancia crítica de la gestión adecuada de memoria dinámica y los recursos del sistema. Durante la práctica, hemos constatado que cualquier fallo en la manipulación de memoria, como desbordamientos de buffer o errores en la liberación de memoria dinámica (fugas o dobles liberaciones), puede ocasionar comportamientos impredecibles que son difíciles de solucionar. Por ello, ha sido fundamental tener un hábito meticuloso en la verificación constante del estado de los recursos y en el aseguramiento de las condiciones de terminación de cadenas de caracteres.

Además, hemos observado que la elección de threads frente a forks y sockets frente a otros mecanismos de comunicación interprocesos (como memoria compartida o pipes) ha resultado altamente acertada. Los threads han aportado eficiencia y simplicidad, especialmente en escenarios con múltiples clientes simultáneos, mientras que el uso de sockets ha garantizado la posibilidad de despliegue en entornos distribuidos, asegurando robustez y escalabilidad.

La incorporación del sistema de logs mediante Arkham ha demostrado ser extremadamente valiosa, facilitando significativamente la monitorización del sistema y permitiendo una mejor depuración de eventos en tiempo real.

Propuestas de mejora:

Gestión avanzada de errores: Incorporar mecanismos más robustos de manejo de errores y excepciones, particularmente en situaciones de alta carga o fallos en la red, para garantizar una recuperación más rápida y estable.

Mayor modularidad: Rediseñar ciertas partes del código, especialmente aquellas relacionadas con el manejo de sockets y threads, para aumentar la modularidad y reutilización del código, facilitando futuras ampliaciones o modificaciones.

En definitiva, esta práctica no solo ha contribuido a nuestro desarrollo técnico, sino que también ha subrayado la necesidad de una buena planificación para afrontar proyectos de este tipo.

5. Bibliografía

GeeksforGeeks. (2025, 7 febrero). What is the MD5 Algorithm? GeeksforGeeks. https://www.geeksforgeeks.org/computer-networks/what-is-the-md5-algorithm/

Programación.com.py. (n.d.). Sockets en C, parte I: Linux. Retrieved from https://www.programacion.com.py/noticias/sockets-en-c-parte-i-linux

Salvador Pont, J. (2014, 23 de septiembre). Programació en UNIX per a pràctiques de Sistemes Operatius. Enginyeria i Arquitectura La Salle.