PRÁCTICA SO

M.J .System

Grupo 10
Ignacio Giral y Marti Farré
(ignacio.giral y marti.farre)
18/05/2025

Índice

3. Introducción

El proyecto **Mr. J System** es una plataforma distribuida implementada en C que permite la distorsión concurrente y fiable de archivos de texto e información multimedia (imágenes y audio). Los usuarios emplean un cliente (**Fleck**) para enviar solicitudes a un servidor central (**Gotham**), encargado de encaminar cada petición a procesos especializados: **Enigma** para texto y **Harley** para medios. Una vez procesado el fichero según el factor de distorsión indicado, el trabajador correspondiente devuelve el resultado al cliente.

Todas las comunicaciones se realizan a través de sockets TCP/IP empleando un protocolo de tramas de tamaño fijo, que garantiza la integridad de los datos mediante checksum y verificación MD5. El desarrollo se ha llevado a cabo de forma incremental por fases.

4. Diseño y Estructura del Sistema

4.1 Visión global

Los componentes principales del sistema son:

- **Gotham (Servidor central):** coordina las peticiones de distorsión, gestiona la conexión estable con los clientes (*Fleck*) y asigna dinámicamente los trabajadores disponibles (*Enigma* para texto y *Harley* para medios).
- **Fleck (Cliente):** interfaz en modo shell que permite al usuario conectarse al sistema, listar archivos de texto y multimedia, solicitar distorsiones y supervisar el progreso.
- **Enigma (Trabajador de texto):** realiza la distorsión de archivos de texto eliminando palabras por debajo de un umbral de longitud.
- **Harley (Trabajador de medios):** procesa imágenes y audio usando la biblioteca proporcionada SO_compression.h, escalando dimensiones o recortando intervalos de audio.
- **Arkham (Proceso de logging):** subproceso generado por Gotham en la fase final, encargado de registrar eventos críticos con marca temporal y garantizar la consistencia mediante exclusión mutua.

El desarrollo se organiza en cuatro fases incrementales:

1. **Fase 1 – Shell y configuración:** creación de los cuatro procesos y lectura de sus ficheros de configuración; implementación de comandos básicos en Fleck.
2. **Fase 2 – Comunicaciones:** establecimiento de conexiones por sockets entre Fleck, Gotham y los trabajadores; gestión de errores y caída de procesos.

3. **Fase 3 – Distorsión de medios:** uso de la biblioteca SO_compression.h para procesar imágenes y audio, con recuperación automática ante fallos de los trabajadores de medios.
4. **Fase 4 – Distorsión de texto y logging:** desarrollo de la lógica de Enigma para archivos de texto, protocolo de logging con Arkham y protección de acceso concurrente al fichero de logs.

4.3 Módulos y responsabilidades

El proyecto se estructura en los siguientes módulos principales:

- **config/**: Contiene archivos de configuración (.dat) para cada componente (cliente, servidor central, trabajadores), con parámetros de IP, puertos y rutas de carpetas.
- **data/**: Conjunto de ficheros (texto, imágenes, audio) usados para validar funcionalidades.
- **modules/**: Biblioteca común que agrupa funcionalidades reutilizables:
    - **readconfig.c/h**: parseo y validación de configuraciones desde archivos de texto, define las estructuras FleckConfig, GothamConfig y WorkerConfig para almacenar parámetros como IPs, puertos y rutas de directorios.
    - **socket.c/h**: abstracción de sockets TCP/IP para crear, vincular, escuchar y aceptar conexiones.
    - **trama.c/h**: construcción de tramas fijas (256 B) con checksum y timestamp para envío/lectura mediante sockets.
    - **string.c/h**: utilidades de manipulación de cadenas (minúsculas, recortes, borrar espacios);
    - **files.c/h**: operaciones sobre el sistema de ficheros (listado de ficheros, comprobación de extensiones, tamaño de un fichero).
    - **distorsion.c/h**: lógica de distorsión que invoca la librería so_compression o el algoritmo de texto según el tipo de archivo.
    - **project.h**: Incluye todas las cabeceras de módulos.
    - **So_compression.h**: Librería de compresión de audio e imágenes proporcionada.
- **linkedlist/**: Implementaciones de listas enlazadas genéricas (linkedlist.c/h, linkedlist2.c/h) para gestionar dinámicamente colecciones de clientes, trabajadores.
- **fleck/**: Cliente de línea de comandos (fleck.c) que procesa comandos de usuario, mantiene conexión estable con el servidor central, solicita distorsiones y gestiona la transferencia de archivos. Usa los módulos comunes para comunicación y procesamiento de ficheros.
- **gotham/**: Servidor central (gotham.c) que atiende registros de trabajadores, acepta conexiones de clientes, enruta peticiones al tipo de trabajador adecuado (texto o multimedia), y gestiona la recuperación ante fallos y reasignaciones de procesos. Se apoya en los módulos para el protocolo de tramas y la gestión de estructuras de datos.

- **worker/**: Proceso genérico (worker.c) que se configura como **Enigma** (texto) o **Harley** (medios). Registra su disponibilidad en Gotham, acepta peticiones de distorsión, aplica la lógica correspondiente, verifica la integridad con MD5 y devuelve los resultados.
- ***Makefile:***
    - Define reglas de compilación.

4.4 Estructuras de datos

- **FleckConfig** (struct FleckConfig): reúne los parámetros de configuración del cliente Fleck (nombre de usuario, carpeta de trabajo e IP/puerto de Gotham) que se leen de config/fleck.dat. Permite inicializar el socket y orientar todas las operaciones de usuario.
- **GothamConfig** (struct GothamConfig): almacena las dos direcciones y puertos donde Gotham atiende conexiones de Fleck y de los workers (Enigma/Harley), a partir de config/gotham.dat. Facilita la creación de los dos sockets de escucha en gotham.c.
- **WorkerConfig** (struct WorkerConfig): recoge la configuración de un trabajador (Enigma o Harley) desde config/enigma.dat o config/harley.dat: IP/puerto de Gotham, dirección propia, carpeta local y tipo de worker. Con esta estructura worker.c arranca el proceso y se registra en el servidor.
- **struct trama**: define el formato fijo de 256 B para intercambiar mensajes vía sockets. Incluye campos para tipo de trama, longitud del bloque de datos, un puntero a estos datos, un checksum para validarlos y un timestamp para trazabilidad. Se usa en todo el código de comunicación (modules/trama.h/.c).
- **LinkedList** y **Node** (struct _Node y struct list_t): implementación genérica de lista enlazada para colecciones dinámicas de tamaño variable. Node almacena un elemento genérico (Element) y un puntero al siguiente, mientras que list_t mantiene punteros a la cabeza, posición actual y un código de error. En Gotham se usa para seguir la lista de workers conectados, y en Fleck para la lista de distorsiones activas y finalizadas.
- **LinkedList2** y **Node** de segundo nivel: igual que el anterior pero con Element2, diseñado para colas de mensajes IPC internas en los workers. Permite encolar y desencolar tareas de distorsión.
- **DistortionThreadParams** (struct DistortionThreadParams): paquete de argumentos que se pasa a cada hilo de distorsión en Fleck. Agrupa el tipo de distorsión, el nombre de fichero, el factor (umbral, escala o intervalo) y un puntero al nodo de progreso asociado.
- **MessageQueueElement** (struct MessageQueueElement): estructura que recorre por la cola de mensajes del worker a través de msgsnd/msgrcv. Contiene toda la información necesaria para procesar una solicitud: tipo de mensaje, nombre de fichero, usuario, tipo de worker, factor, tamaños de transferencia, MD5SUM, descriptor de fichero, directorio de trabajo, identificador de hilo y estado de la tarea.

4.5 Recursos del sistema

En el proyecto se utilizan los siguientes mecanismos y llamadas al sistema:

- ***Sockets TCP/IP***

  Se utilizan sockets en todas las comunicaciones remotas. Elegimos TCP porque garantiza entrega ordenada y fiable de datos, esencial para no corromper los ficheros que viajamos entre cliente (Fleck), servidor central (Gotham) y trabajadores (Enigma/Harley).

- ***Pthread***

  Fleck (cliente)
  - Watcher_thread: monitoriza de forma continua la conexión con Gotham.
  - distortFileThread (uno por distorsión): gestiona la transferencia y recepción de fragmentos de fichero con el worker asignado, sin bloquear la consola de comandos.

  Gotham (servidor central)
  - thread_newFleck (hilo por cliente Fleck): gestiona la negociación de conexión y redirección a Enigma/Harley.
  - thread_newWorker (hilo por worker): atiende el registro de nuevos workers y marca disponibilidad.
  - thread_Connecter de escucha en worker y fleck.

  Worker (Enigma/Harley)
  - distortFileThread (uno por distorsión): procesa cada solicitud de distorsión (recibe trama 0x03, transfiere fichero, invoca compresión, verifica MD5 y devuelve resultado).
  - watcher_thread: detecta desconexiones con Gotham o Fleck.

- ***Mutexes (pthread_mutex_t)***
  Se usan para proteger cualquier acceso concurrente a recursos compartidos y evitar condiciones de carrera, por ejemplo:
  - En Fleck, al enviar tramas y actualizar el progreso de distorsiones.
  - En Gotham, al modificar las listas de clientes y workers y al escribir en la tubería de logs.
  - En Workers, al gestionar la cola interna de peticiones.

- ***Señales (SIGINT)***

  - SIGINT: capturada en Fleck, Gotham y en Worker (Enigma/Harley) para detectar Ctrl+C; el handler marca un flag de parada, envía el frame de desconexión y cierra sockets y hilos de forma ordenada.

- ***Colas de mensajes***
    - Usadas en los Workers (Enigma/Harley) para desacoplar la recepción de peticiones de distorsión de su procesamiento.
    - El worker crea una cola con msgget(), el hilo de escucha envía (msgsnd) cada solicitud encapsulada en un MessageQueueElement, y el Queue Manager la recibe con msgrcv.
    - Esto permite pueda seguir aceptado la llegada de nuevas peticiones, mientras el procesador saca de la cola cuando esté libre.
- ***Pipes***
    - En el módulo de distorsión, se crean pipes para ejecutar md5sum en un proceso hijo y capturar su salida en el padre, obteniendo el hash MD5 del fichero sin librerías externas.
    - En Gotham, se establecen dos pipes al hacer fork() del logger Arkham: el proceso padre escribe eventos con timestamp en la tubería y Arkham lee desde el otro extremo para escribirlo en logs.txt.
- ***fork() y exec***
    - En el módulo de distorsión, se invoca fork() seguido de execlp("sh", "sh", "-c", "md5sum <archivo>") para calcular el hash MD5 sin bloquear el proceso principal: el hijo ejecuta md5sum y el padre lee el resultado por un pipe anónimo.
    - En Gotham, se usa fork() para crear el proceso Arkham (logger): el padre escribe eventos en una pipe, y el hijo hereda el extremo de lectura y vuelca esas entradas en logs.txt.

- ***(open/read/write/close)***
    - Usadas en Fleck, Gotham y Workers para todos los accesos a sockets y ficheros.
- ***(opendir/readdir/*closedir*)***
    - All ejecutar los comandos LIST TEXT y LIST MEDIA, se abre la carpeta del usuario con opendir(), se recorren todos los ficheros con readdir() y se filtran por extensión (por ejemplo, .txt o .wav/.png), para después mostrar al usuario la lista de archivos disponibles. Finalmente se cierra la carpeta con closedir().

- ***Memoria dinámica (malloc/free, strdup)***
    - Para gestionar en tiempo de ejecución cadenas y buffers de tamaño variable.

4.6. Desarrollo por Fases

4.6.1 Fase 1: Shell y gestión de configuración

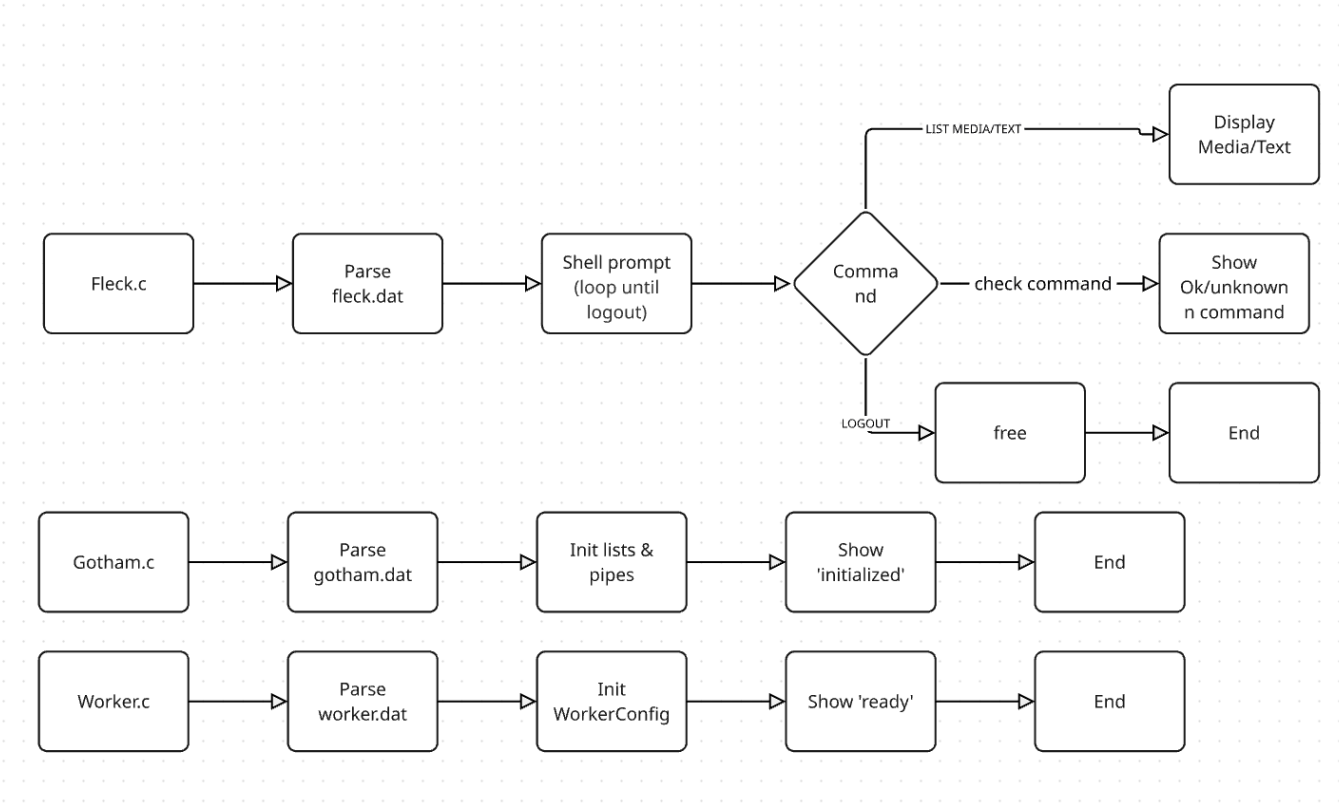
En la Fase 1 se sientan las bases sin todavía abrir sockets ni distorsionar datos; el objetivo es comprobar que cada uno de los cuatro procesos (Fleck, Gotham, Enigma y Harley)

1. Lee correctamente su fichero de configuración
    - Todos usan el módulo readconfig (en modules/readconfig]), que:
        - Abre el .dat correspondiente (Fleck: config/fleck.dat; Gotham: config/gotham.dat; Worker: config/enigma.dat o config/harley.dat).
        - Lee línea a línea con read() y elimina caracteres prohibidos (& para username).
        - Rellena una estructura en memoria (FleckConfig, GothamConfig o WorkerConfig) con los campos IP, puerto, directorio y tipo de worker.
2. Fleck (cliente)
    - Tras cargar FleckConfig, entra en un bucle de shell (prompt) implementado en fleck/fleck.c que:
        - Lee la línea de usuario con read.
        - Normaliza mayúsculas/minúsculas (strcasecmp) para reconocer los comandos:
            - LIST MEDIA / LIST TEXT → llama a modules/files.c, recorre la carpeta de trabajo con opendir()/readdir() y lista solo archivos .wav, .png, .txt…
            - CONNECT, LOGOUT, DISTORT, CHECK STATUS, CLEAR ALL, de momento sólo devuelven un mensaje “Command OK” o “Unknown command” según si la sintaxis es válida.
3. Gotham (servidor central)
    - En gotham/gotham.c invoca READCONFIG_read_config_gotham(), guarda los parámetros en GothamConfig y:
        - Inicializa dos listas enlazadas (linkedlist.h): una para clientes Fleck y otra para workers.

        - Muestra “Gotham server initialized” y termina, validando que el parseo y la inicialización de estructuras funcionan.

4. Enigma y Harley (workers)
    - Ejecuta READCONFIG_read_config_worker(), almacena la configuración en WorkerConfig y muestra mensaje de exito.

5. Liberación de recursos
    - Todos cierran CTRL+C (capturado por signal(SIGINT)) o por el comando LOGOUT, llaman a free_config(), destruyen las y liberan memoria.



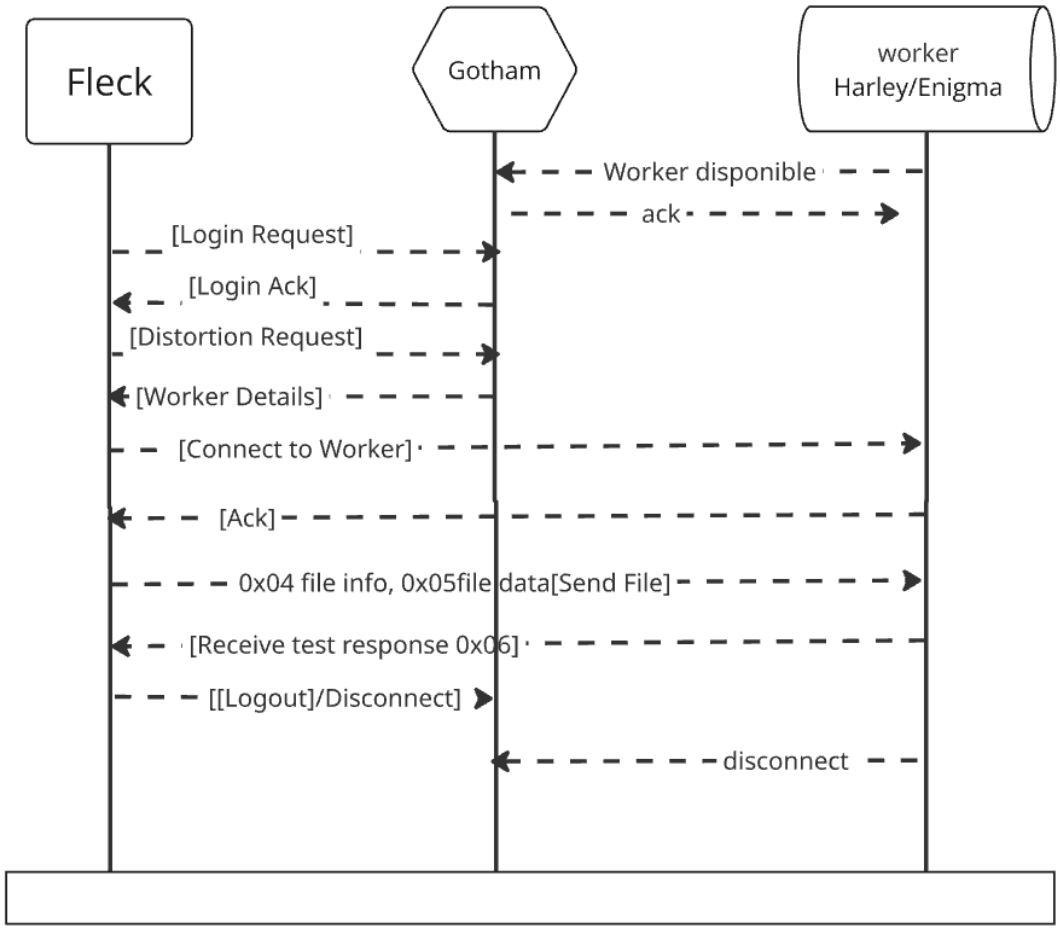
4.6.2 Fase 2: Comunicación por sockets y protocolo de tramas

En esta segunda fase, el enunciado plantea que Gotham debe abrir dos sockets de escucha: uno destinado a los clientes Fleck y otro para los workers (Enigma y Harley). A continuación, cada worker se une al sistema anunciando su presencia mediante un frame 0x02, de modo que Gotham pueda gestionar su disponibilidad. Por su parte, Fleck, al emitir el comando CONNECT, establece una conexión permanente con Gotham enviando un frame 0x01 y espera su confirmación antes de aceptar cualquier otra instrucción.

Cuando el usuario solicita una distorsión con el comando DISTORT, Fleck envía un frame 0x10 al servidor central. Gotham, a su vez, debe responder con la dirección (IP y puerto) de un worker libre o con un frame DISTORT_KO si en ese momento no hay ninguno disponible. Gracias a esa respuesta, Fleck abre entonces un segundo socket estable con el worker asignado y procede a transferir los datos para la distorsión.

- Fleck
    1. Tras el comando CONNECT, invoca SOCKET_createSocket(cfg.server_port, cfg.server_ip) para establecer una conexión duradera con Gotham (fleck/fleck.c).
    2. Envía un frame de tipo 0x01 con su nombre de usuario mediante TRAMA_sendMessage.
    3. Recibe confirmación (frame.tipo == 0x01) y mantiene el sockfd_G abierto para futuras comunicaciones.
    4. Al ejecutar DISTORT <file> <factor>, construye un frame 0x10 (STRING_buildDistortRequest) y lo envía a Gotham.
    5. Procesa la respuesta de Gotham (IP y puerto del worker o DISTORT_KO) y abre una nueva conexión con el worker usando SOCKET_createSocket.
- Gotham
    1. Lanza dos sockets de escucha con SOCKET_initSocket en los puertos de Fleck y de workers (gotham/gotham.c).
    2. Crea hilos: uno que acepta clientes Fleck (pthread_create en listenFlecks), otro que acepta registros de workers (listenWorkers).
    3. En handleFleckSession, interpreta los frames 0x01 (conexión) y 0x10 (petición de distorsión), consulta la lista de workers activos (workerList), y responde con un frame 0x10 conteniendo la dirección del worker seleccionado.
    4. En handleWorkerSession, recibe frames 0x02 de registro de workers (tipo Text o Media) y actualiza dinámicamente workerList para futuras peticiones.
- Workers (Enigma/Harley)
    1. Inician una conexión con Gotham tras leer config/enigma.dat o harley.dat: SOCKET_createSocket(cfg.gotham_port, cfg.gotham_ip) y envían frame 0x02.
    2. Configuran un socket de escucha en su propio puerto (SOCKET_initSocket) y bloquean en accept() para futuros Flecks.

3. En handleFleckConnection, reciben un frame 0x03 con metadatos de distorsión, responden con 0x03, pero en esta fase simulan el proceso de envío/recepción de archivos mediante un par de frames 0x05 de datos de prueba y cierran la sesión.



4.6.3 Fase 3: Distorsión de medios

En la Fase 3 del proyecto se implementa por completo la distorsión de archivos multimedia (imágenes y audio) en los procesos Harley, así como el mecanismo para recuperarse si el worker principal cae.

1. Fleck
    - Cuando el usuario ejecuta DISTORT <fichero> <factor>, Fleck lanza un hilo distortFileThread que se encarga de:

        1. Preguntar a Gotham qué worker debe atender la petición (trama 0x10).

2. Conectarse al Harley indicado (IP/puerto).

3. Enviar un frame 0x03 con metadatos: nombre de usuario, nombre de fichero, tamaño, MD5 y factor.

4. Esperar la confirmación y, si es OK, enviar información de fichero, partir el fichero y enviar cada uno en un frame, actualizando simultáneamente la barra de progreso en memoria.

5. Recibir del worker el frame 0x04 con el nuevo tamaño y MD5 del fichero comprimido, y luego los frames 0x05 con los datos resultantes.

6. Verificar con el frame 0x06 que el MD5 coincide (CHECK_OK) y cerrar la conexión.

connection_watcher detecta la pérdida de socket, su reacción es terminar la sesión de distorsión en curso y cerrar Fleck.

2. Gotham

    - Su threadFleck y threadWorker, mantienen la lista de workers activos.

    - Si un Harley se desconecta, threadWorker detecta el EOF en el socket y lo marca como inactivo, de modo que en la próxima petición de Fleck Gotham derive a otro worker.

3. Harley (worker multimedia)

    - Arranca un listener_thread que atiende accept() y, por cada conexión entrante de Fleck, crea una sesion thread invocando la función distortFileThread

    - Dentro de ese hilo:

        1. Recibe el frame 0x03, 0x04 i 0x05, lee el fichero completo en disco haciendo read() en bucle, y comprueba el MD5 recibido.

        2. Llama a SO_compressImage o SO_compressAudio para procesar el archivo.

        3. Una vez comprimido, obtiene el nuevo MD5 y tamaño, envía el frame 0x04 y, protegido por un mutex (pthread_mutex_t myMutex en distorsion.c), fragmenta y manda el fichero resultante en frames 0x05.

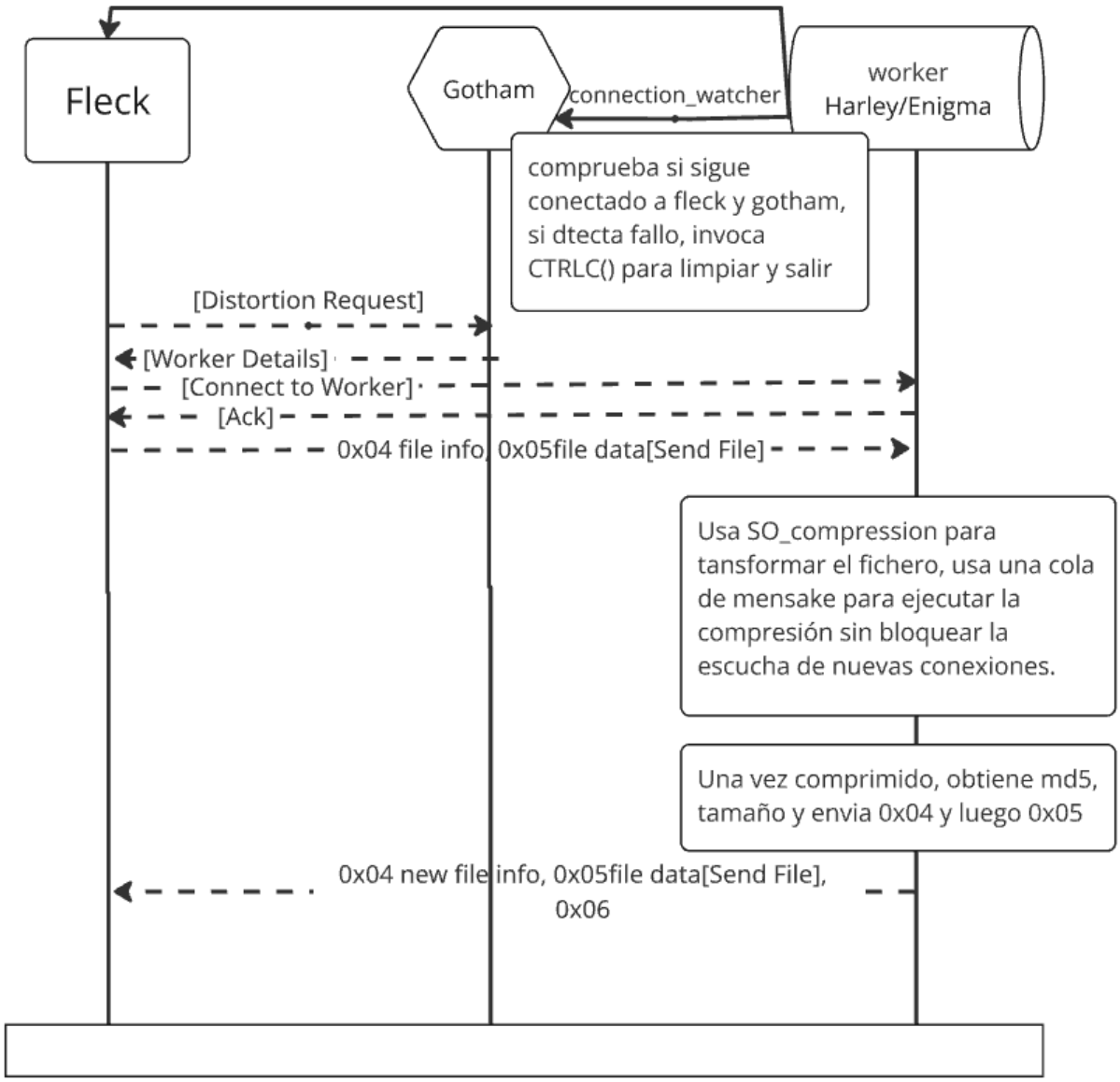
4. Finalmente envía el frame 0x06 con CHECK_OK, cierra el socket y libera memoria.
5. Cada Harley arranca, además, un hilo connection_watcher que cada 5 s comprueba que sí sigue conectado a Gotham y a Fleck. Si detecta fallo, invoca CTRLC() para limpiar y salir ordenadamente.

Cada Harley con connection_watcher comprueba si el socket sigue activo, si detecta la desconexión, llama a la rutina de limpieza (CTRLC) para cerrar sockets y terminar su propio proceso.

Proceso de distorsión:

Usando la librería SO_compression transforma el fichero, internamente, cada Harley usa una cola de mensajes para desacoplar la recepción de tramas de la ejecución de la compresión.

Al recibir uno de estos elementos, ejecuta la compresión sin bloquear la escucha de nuevas conexiones.
Este desacoplo permite que, aunque un hilo de compresión quede bloqueado en un bucle pesado, el worker siga aceptando nuevas peticiones y encolándolas, después envía el resultado al cliente.


Fleck
Gotham
connection_watcher
worker
Harley/Enigma
comprueba si sigue conectado a fleck y gotham, si dtecta fallo, invoca CTRLC() para limpiar y salir
[Distortion Request]
[Worker Details]
[Connect to Worker]
[Ack]
0x04 file info, 0x05file data[Send File]
Usa SO_compression para tansformar el fichero, usa una cola de mensake para ejecutar la compresión sin bloquear la escucha de nuevas conexiones.
Una vez comprimido, obtiene md5, tamaño y envia 0x04 y luego 0x05
0x04 new file info, 0x05file data[Send File], 0x06

4.6.3 Fase 4: Distorsión de texto y sistema de logs

1. Envío del archivo

- Fleck envía un frame 0x03 con metadatos (userName&fileName&fileSize&md5sum&threshold) y, tras recibir el OK, transmite el fichero en trozos de (frames 0x05), verificando en cada uno el checksum (0x06).

Procesamiento en el worker

- En worker.c, en la función DISTORSION_distortFile:
    - Se detecta element->status == 2 y worker_type == "Text".
    - Se llama a DISTORSION_compressText(path, atoi(element->factor)) (módulo modules/distorsion.c).
    - Esta función:
        1. Lee todo el contenido en memoria.
        2. Recorre palabra a palabra, copiando sólo aquellas cuya longitud ≥ umbral.
        3. Sobrescribe el fichero original con el texto filtrado.

Devolución del archivo distorsionado

- Tras el filtrado, el worker calcula md5, envia 0x04, 0x05 y finalmente 0x06, fleck recibe, compara y guarda el fichero distorsionado en su carpeta.

2. Logger Arkham

1. Creación del proceso logger
    - En gotham/gotham.c se crea un pipe (pipe_fds) y luego se fork().
        1. Hijo (Arkham): cierra el extremo de escritura, lee (read(pipe_fds[0],…)) en bucle y vuelca cada línea en logs.txt con open(O_APPEND)/write/close.
        2. Padre (Gotham): cierra el extremo de lectura y continúa su trabajo normal.
2. Emisión de eventos

Cada vez que ocurre se quiere log algo, Gotham llama a:
log_event("Descripción del evento");

- Esta función:
    1. Toma la hora actual (time(NULL)), formatea "[YYYY-MM-DD HH:MM:SS] evento\n".

    2. Con un mutex (log_mutex) asegura un único escritor sobre el pipe.

    3. Escribe la cadena en el pipe

- El uso de pthread_mutex_t en log_event evita que varios hilos de Gotham (p. ej. los que atienden Flecks y workers) se solapen al escribir en la pipe.

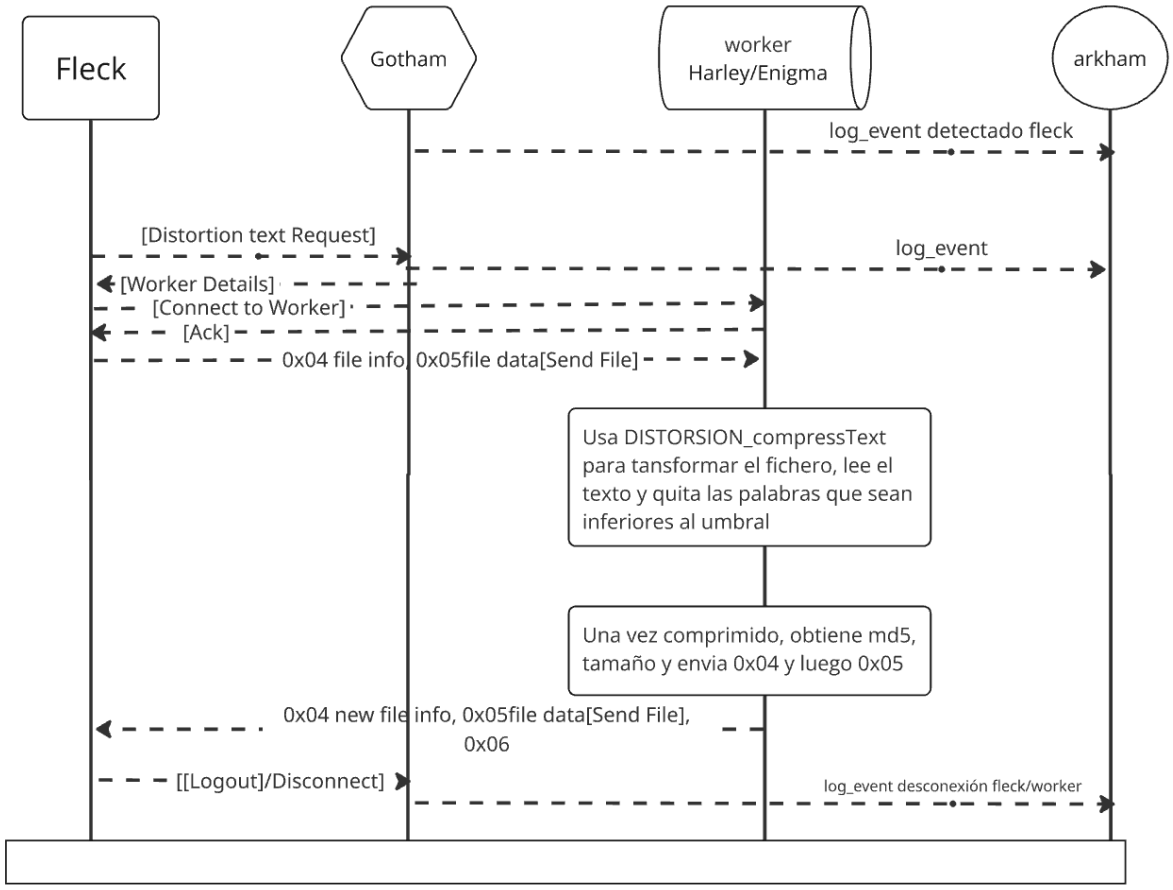
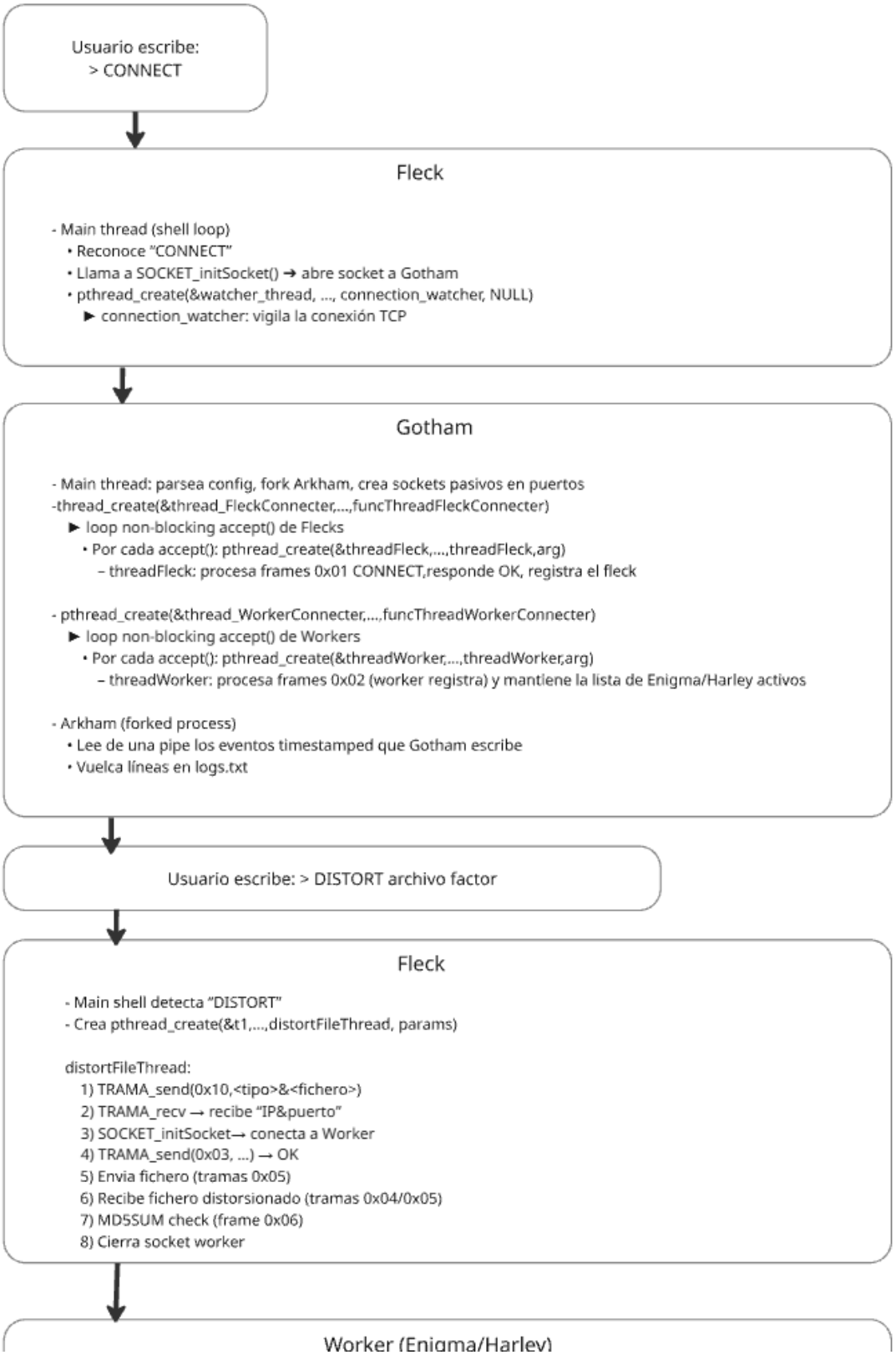
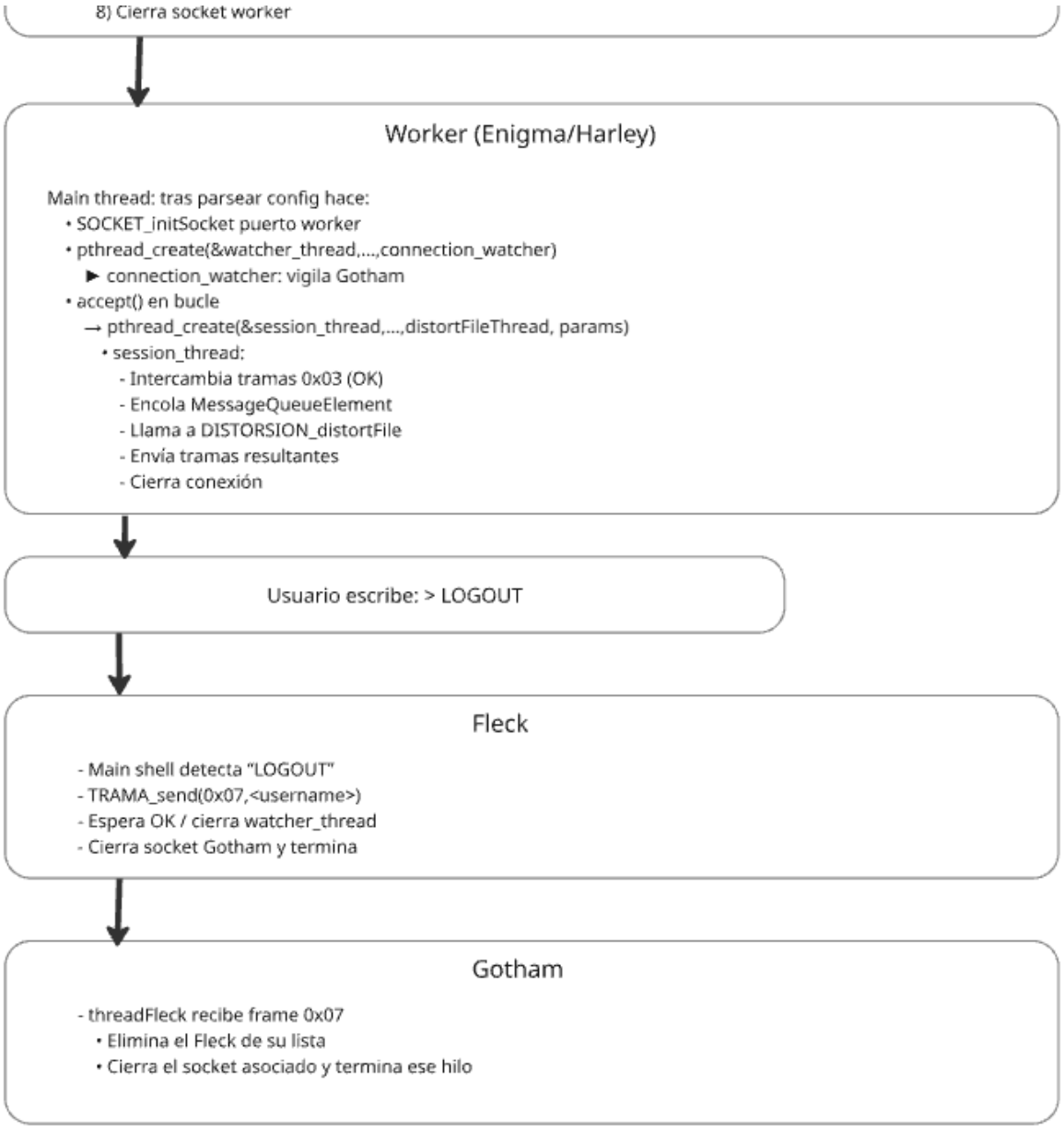


Diagrama Global de comandos connect, distort y logout.




8) Cierra socket worker
Worker (Enigma/Harley)
Main thread: tras parsear config hace:
• SOCKET_initSocket puerto worker
• pthread_create(&watcher_thread,...,connection_watcher)
► connection_watcher: vigila Gotham
• accept() en bucle
→ pthread_create(&session_thread,...,distortFileThread, params)
• session_thread:
- Intercambia tramas 0x03 (OK)
- Encola MessageQueueElement
- Llama a DISTORSION_distortFile
- Envía tramas resultantes
- Cierra conexión
Usuario escribe: > LOGOUT
Fleck
- Main shell detecta "LOGOUT"
- TRAMA_send(0x07,<username>)
- Espera OK / cierra watcher_thread
- Cierra socket Gotham y termina
Gotham
- threadFleck recibe frame 0x07
• Elimina el Fleck de su lista
• Cierra el socket asociado y termina ese hilo

5. Problemas encontrados y soluciones

Acceso concurrente sin protección: Al principio, varias rutinas de hilo intentaban modificar al mismo tiempo estructuras compartidas (la lista de workers en Gotham o la cola de progreso en Fleck), lo que provocaba comportamientos erráticos o bloqueos. La solución fue incorporar mutexes alrededor de las operaciones críticas, asegurando que sólo un hilo a la vez pudiera actualizar esas colecciones.

Fallas en la verificación de integridad: En las primeras pruebas, ocasionalmente detectábamos discrepancias entre el checksum calculado en origen y el recibido, lo que obligaba a descartar archivos válidos. Ajustamos el cálculo de la suma de bytes para que incluyera correctamente el padding de las tramas..

Manejo de interrupciones (Ctrl+C): Al pulsar Ctrl+C en Fleck o en un worker, el cierre a veces era brusco, dejando sockets abiertos o memoria sin liberar. Registramos un handler para SIGINT que envía los frames de logout apropiados, cierra los sockets ordenadamente y libera todos los recursos antes de terminar el proceso.

Por último, durante la revisión con Valgrind surgieron **fugas de memoria** en varios puntos: buffers de trama no liberados, strdup sin free al cerrar una sesión o estructuras de configuración olvidadas. Resolvimos la mayoria revisando cada malloc/strdup y asegurando su correspondiente free al finalizar la funcionalidad..

6. Estimación temporal

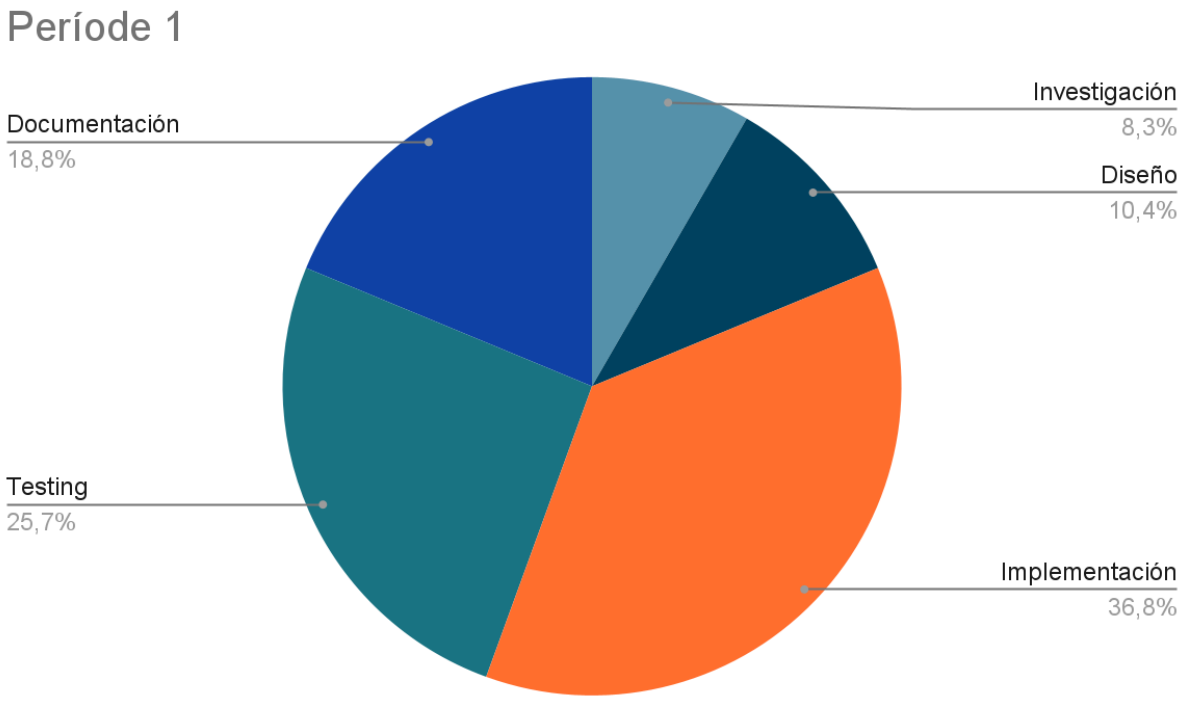
Ignacio:

- Investigación: 14 horas
- Diseño: 10 horas
- Implementación: 58 horas
- Testing: 21 horas
- Documentación: 10 horas

Martí:

- Investigación: 14 horas
- Diseño: 10 horas
- Implementación: 41 horas
- Testing: 14 horas
- Documentación: 16 horas

Total = 208 horas



7. Conclusiones y propuestas de mejora

El desarrollo de Mr. J System ha sido una gran experiencia que ha demostrado el valor de un diseño modular y por fases. Dividir el proyecto en componentes claramente diferenciados el cliente Fleck, el servidor central Gotham, los workers de distorsión Enigma y Harley, y el logger Arkham nos permitió avanzar con seguridad, probando y puliendo cada parte antes de pasar a la siguiente. Gracias a esta aproximación, hemos conseguido un sistema capaz de atender múltiples usuarios concurrentes, de reenviar tareas de distorsión de forma fiable y de registrar todos los eventos sin pérdida de información.
Uno de los mayores logros ha sido la tolerancia a fallos: al diseñar mecanismos de reasignación de workers, la plataforma recupera tareas interrumpidas y mantiene el servicio funcionando incluso cuando ocurren caídas inesperadas. Asimismo, la verificación de integridad con checksum en cada trama y MD5 tras la transferencia garantiza que ningún archivo se corrompa en tránsito.

Para el futuro, hay varias mejoras que podrían llevar a Mr. J System a un nuevo nivel. En primer lugar, se podría mejorar aún más el sistema de recuperación. También podríamos ampliar el abanico de efectos añadiendo filtros de eco, cambio de tono o desenfoque en multimedia, así como alternativas de cifrado o sustitución de sinónimos en texto.
Por último, sería interesante incorporar una interfaz de monitorización en tiempo real que muestre el estado de las colas de trabajo, el uso de recursos y las estadísticas de éxito o fallo.

8. Interpretación de los nombres del sistema

Los nombres están inspirados en el universo de Batman:

- ***Fleck**: toma el apellido de Arthur Fleck, el Joker.*
- ***Gotham**: hace referencia a Gotham City, la ciudad de Batman.*
- ***Enigma**: a The Riddler, villano famoso por sus acertijos.*
- ***Harley**: homenaje a Harley Quinn, compañera del Joker.*
- ***Arkham**: referencia a Arkham Asylum.*

10. Bibliografía

Kerrisk, M. (s.f.). *Linux man pages online*. man7.org.
https://www.man7.org/linux/man-pages/index.html
Computer Science | Kent State University. (s.f.). *Introduction To Unix Signals Programming*.
https://www.cs.kent.edu/~ruttan/sysprog/lectures/signals.html
Beej's Web Page. (s.f.). *Beej's Guide to Network Programming*.
https://beej.us/guide/bgnet/
GeeksforGeeks. (s.f.). *Mutex vs Semaphore - GeeksforGeeks*.
https://www.geeksforgeeks.org/mutex-vs-semaphore/
OpenAI. (s.f.). *ChatGPT*.
https://chat.openai.com/