Mr. J. System

Fase 1: Calienta que sales

Sistemes Operatius – Curs 24/25

Alvaro Bello Garrido (alvaro.bello)

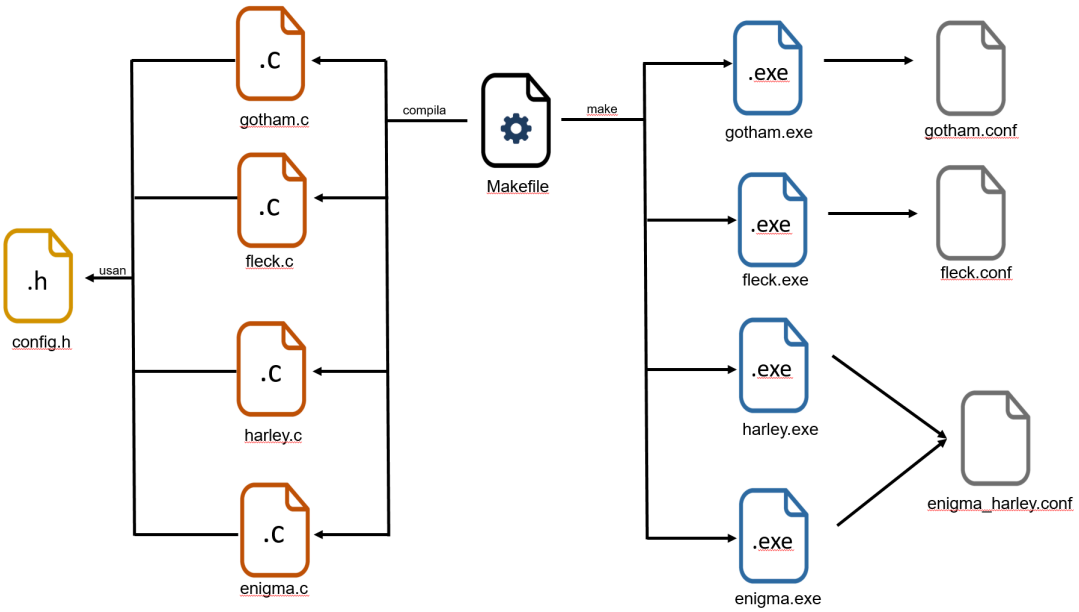
Carla Francos Molina (carla.francos)

20/10/2024

Índice

1. Diagrama de procesos

El siguiente diagrama muestra cómo los componentes del programa están organizados y cómo interactúan entre sí. Los archivos fuente (.c) utilizan las funciones de la librería personalizada config.h dentro de su programa y se compilan en ejecutables mediante un Makefile, y cada ejecutable carga su respectivo archivo de configuración al momento de ejecutarse. El uso de archivos .conf permite modificar configuraciones externas sin necesidad de recompilar el código fuente, lo que facilita la gestión y ejecución de cada componente del sistema.



2. Estructuras de datos

GothamConfig, HarleyConfig/EnigmaConfig y FleckConfig son las estructuras clave que permiten almacenar la configuración de cada componente. La elección de estas estructuras se han pensado así por la necesidad de almacenar datos de forma dinámica como direcciones IP, puertos o directorios, que pueden cambiar según el archivo de configuración.

2.1 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.

2.2 HarleyConfig/EnigmaConfig

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

```
typedef struct {
    char* ip_gotham;    // Dirección IP para Gotham
    int port_gotham;    // Puerto para Gotham
    char* ip_fleck;     // Dirección IP para Fleck
```

```
    int port_worker;    // Puerto para Fleck
    char* worker_dir;   // Directorio de trabajo para Enigma/Harley
    char* worker_type;  // Tipo de worker ("Media" o "Text")
} EnigmaConfig;
```

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.

2.3 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.

3. Justificación del diseño

El diseño del sistema está estructurado para que cada componente (Gotham, Harley/Enigma, Fleck) sea autónomo y esté modularmente separado. Esta separación facilita tanto el desarrollo como el mantenimiento del sistema. Cada componente tiene sus propias responsabilidades y archivos de configuración, lo que permite cambiar un componente sin afectar directamente a los demás.

- **Modularidad**: Los componentes están diseñados para ser lo más independientes posible, permitiendo que se ejecuten y se configuren por separado.
- **Escalabilidad**: Al cambiar los archivos de configuración, es posible reconfigurar el sistema para diferentes entornos o redes sin necesidad de modificar el código fuente, lo que otorga flexibilidad y facilidad de mantenimiento.

3.1 Lectura Dinámica

Cada componente lee su configuración de un archivo externo mediante la función `read_until`, que permite leer hasta un delimitador específico (`\n`). Esto permite:

- **Flexibilidad en el formato de los archivos**: Se pueden añadir nuevas líneas en los archivos de configuración o modificar los valores sin necesidad de cambiar el código fuente.
- **Uso eficiente de memoria**: La función `realloc` permite ajustar dinámicamente el tamaño de las cadenas, utilizando solo la memoria necesaria en cada momento.

3.2 Gestión de Memoria

El sistema hace un uso eficiente de la memoria mediante la liberación de los recursos una vez que ya no son necesarios. Por ejemplo, se utiliza `free` para liberar la memoria asignada dinámicamente a las cadenas y estructuras tras la lectura de la configuración. Esto garantiza que el sistema no sufra de fugas de memoria y maximiza el rendimiento a lo largo del tiempo.

3.3 Optimización para el Futuro

Aunque el comportamiento de Fleck aún no ha sido definido en detalle, el diseño modular actual permitirá su integración sin necesidad de grandes cambios en Gotham o Harley/Enigma. La estructura `GothamConfig` ya anticipa la conexión con Fleck, lo que facilita su futura implementación.