Mr. J. System

Fase 1: Pràctica de Sistemes Operatius

Grau d'Enginyeria Electrònica de Telecomunicacions

Universitat Ramon Llull

Pau Olea Reyes
pau.olea@students.salle.url.edu

Alfred Chávez Fernández
alfred.chavez@students.salle.url.edu

15 de desembre de 2024

Professors: Jordi Malé

Curs: 2024-2025

Índex

1 Introducció

Aquest informe detalla la primera fase de la pràctica de Sistemes Operatius, incloent el diagrama de blocs, les estructures de dades utilitzades i la justificació de les decisions de disseny preses.

2 Diagrama de Blocs

A continuació es presenta el diagrama de blocs que explica la Fase 1 de la pràctica. Aquest diagrama mostra les relacions entre els diferents fitxers que conformen el sistema i com aquests s'utilitzen en el procés de desenvolupament.

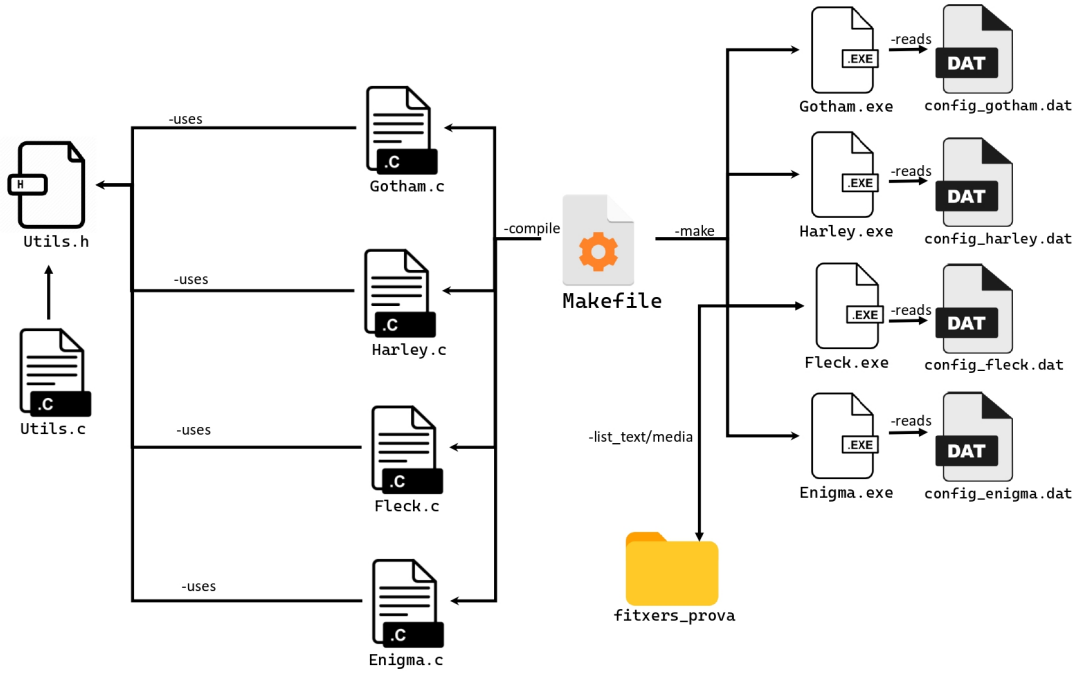

Figura 1: Diagrama de blocs de la Fase 1

El diagrama de blocs il·lustra com es creen i s'interconnecten els processos del sistema per a la Fase 1 del projecte "Mr. J. System". Aquesta fase se centra en la configuració i la preparació dels quatre processos principals: `Fleck`, `Harley`, `Enigma`, i `Gotham`, a partir de la informació dels fitxers de configuració. També s'han afegit les noves funcionalitats de LIST TEXT i LIST MEDIA que permeten gestionar fitxers de text i multimèdia.

- **Fitxers de codi font i capçaleres**: Els fitxers C (`.c`) són els fitxers de codi font que implementen la funcionalitat dels diferents components del sistema. El fitxer `Utils.c`, juntament amb el seu fitxer de capçalera `Utils.h`, conté funcions auxiliars comunes que són utilitzades per altres fitxers, com `Gotham.c`, `Harley.c`, `Fleck.c`, i `Enigma.c`. Aquesta organització modular facilita la reutilització de codi i permet la separació clara de les funcionalitats de cada component.
- **Processos de compilació i generació d'executables**: El fitxer `Makefile` s'encarrega de la compilació automàtica del projecte. Coordina la creació dels executables (`.exe`) a

partir dels fitxers de codi font i capçalera. Mitjançant l'ordre `make`, es generen els executables `Gotham.exe`, `Harley.exe`, `Fleck.exe`, i `Enigma.exe`. Aquesta automatització simplifica la construcció del projecte i garanteix que totes les dependències es gestionin correctament.

- **Configuració inicial dels processos**: Cada procés requereix un fitxer de configuració (`config_gotham.dat`, `config_harley.dat`, `config_fleck.dat`, `config_enigma.dat`) per definir els paràmetres necessaris per a la seva execució, com l'IP, el port i altres configuracions específiques del procés. Els fitxers de configuració són processats a l'inici per a cada procés per establir els paràmetres del sistema. Per exemple, `Fleck` utilitza el fitxer de configuració per obtenir el directori de treball i la informació de connexió amb `Gotham`.
- **Noves funcionalitats LIST TEXT i LIST MEDIA**: Les noves funcionalitats LIST TEXT i LIST MEDIA permeten llistar i gestionar fitxers de text i multimèdia, respectivament. Aquests fitxers es troben al directori `fitxers_prova`, on es troben els fitxers de mostra per executar les proves. La funció LIST TEXT cerca i llista tots els fitxers de text disponibles al directori, mentre que LIST MEDIA fa el mateix amb els fitxers multimèdia. Aquesta funcionalitat facilita la gestió automatitzada dels recursos del sistema.
- **Organització i interdependències**: La relació entre els diferents components està clarament definida. Els fitxers `.c` utilitzen `Utils.h` per accedir a funcions compartides, i el `Makefile` coordina la compilació de tot el projecte per generar els executables que, posteriorment, s'executaran amb les configuracions definides en els fitxers `.dat`. Les noves funcionalitats l·list_textï l·list_mediaïnteractuen amb els processos principals per llistar els fitxers necessaris per a l'execució del sistema.

Aquest diagrama mostra clarament com s'integren els diferents components per crear un sistema on els processos poden interactuar correctament segons la configuració proporcionada. Permet modificar paràmetres sense necessitat de recompilar, ja que els fitxers de configuració es poden actualitzar de manera independent.

3 Estructures de Dades Utilitzades

A continuació es presenten les principals estructures de dades utilitzades en la primera fase de la pràctica, juntament amb una explicació de la seva funció i justificació:

3.1 Estructura `FleckConfig`

Aquesta estructura s'utilitza per emmagatzemar la configuració necessària per la comunicació amb Gotham. Conté informació com l'usuari, directori i l'adreça IP i port de Gotham.

```
1        typedef struct {
2            char *user;
3            char *directory;
4            char *ipGotham;
5            int portGotham;
6        } FleckConfig;
```

Listing 1: Definició de `FleckConfig`

Justificació: L'ús de memòria dinàmica per a l'usuari, directori i IP permet gestionar diferents longituds d'entrada de forma flexible, millorant la gestió de memòria en relació a altres mètodes més rígids, com l'ús de `arrays` fixos.

- `user`: una cadena de caràcters que representa el nom de l'usuari que executa el sistema o està associat amb la configuració. Aquesta informació pot ser útil per a propòsits d'autenticació o identificació.
- `directory`: una cadena de caràcters que indica el directori en el qual es troben els fitxers de configuració o altres recursos necessaris per al sistema. El directori s'utilitza per accedir a aquests fitxers de manera dinàmica durant l'execució del programa.
- `ipGotham`: una cadena de caràcters que conté l'adreça IP del servidor Gotham. Aquesta IP s'utilitza per establir la connexió amb el servidor remot, la qual cosa és fonamental per permetre la comunicació en xarxa.
- `portGotham`: un enter que especifica el port del servidor Gotham. El port és un paràmetre crític per a la comunicació en xarxa, ja que determina el punt d'accés del servidor amb el qual es comunicarà el client.

3.2 Estructura `HarleyConfig`

Aquesta estructura conté les dades de configuració per a la comunicació amb Harley, de forma semblant a `FleckConfig`, però adaptada per Harley.

```
1         typedef struct {
2             char *user;
3             char *directory;
4             char *ipHarley;
5             int portHarley;
6         } HarleyConfig;
```

Listing 2: Definició de `HarleyConfig`

Justificació: L'ús de punters a `char` ens permet obtenir configuracions variables des de fitxers de configuració externs `.dat` i adaptar les estructures segons sigui necessari durant l'execució del programa.

- `ipGotham`: una cadena de caràcters que representa l'adreça IP del servidor Gotham. És utilitzada per connectar-se al servidor quan es necessita interacció amb ell.
- `portGotham`: un enter que especifica el port associat al servidor Gotham. Aquesta informació és necessària per configurar la comunicació de xarxa amb el servidor adequat.
- `ipFleck`: una cadena de caràcters que representa l'adreça IP del servidor Fleck. Es fa servir per establir connexions amb aquest servidor.
- `portFleck`: un enter que indica el port del servidor Fleck. El port permet identificar el servei amb el qual es comunicarà el sistema.
- `directory`: una cadena de caràcters que especifica el directori de treball on es poden trobar fitxers i altres recursos necessaris per a l'operació del sistema.
- `workerType`: una cadena de caràcters que representa el tipus de treballador o mòdul que utilitzarà el sistema. Això pot influir en el comportament o la configuració específica del sistema segons el tipus de tasca que es realitzi.

3.3 Estructura `GothamConfig`

L'estructura `GothamConfig` conté informació relacionada amb Gotham, incloent l'usuari, directori, l'adreça IP i port.

```
typedef struct {
    char *user;
    char *directory;
    char *ipGotham;
    int portGotham;
} GothamConfig;
```

Listing 3: Definició de `GothamConfig`

Justificació: Similar a `FleckConfig` i `HarleyConfig`, aquesta estructura emmagatzema els detalls de configuració del mòdul Gotham. S'utilitzen punters per permetre una gestió flexible de les dades.

- `ipFleck`: una cadena de caràcters que conté l'adreça IP del servidor Fleck. S'utilitza per connectar el sistema al servidor corresponent per a la comunicació de dades.
- `portFleck`: un enter que indica el port del servidor Fleck. El port és necessari per identificar el servei amb el qual es connectarà.
- `ipHarEni`: una cadena de caràcters que representa l'adreça IP del servidor Harley Enigma. Aquesta IP s'utilitza per connectar-se amb Harley Enigma per a operacions de xarxa.
- `portHarEni`: un enter que especifica el port del servidor Harley Enigma, necessari per accedir al servei correcte.

3.4 Estructura `EnigmaConfig`

L'estructura `EnigmaConfig` s'utilitza per emmagatzemar informació relativa a Enigma, incloent els mateixos camps que en les altres configuracions.

```
1         typedef struct {
2             char *user;
3             char *directory;
4             char *ipEnigma;
5             int portEnigma;
6         } EnigmaConfig;
```

Listing 4: Definició de `EnigmaConfig`

Justificació: L'estructura `EnigmaConfig` segueix el mateix patró que les altres configuracions, per tal de mantenir la coherència i facilitar la gestió del codi i les configuracions externes.

- `ipGotham`: una cadena de caràcters que conté l'adreça IP del servidor Gotham, per establir connexions.
- `portGotham`: un enter que representa el port associat amb el servidor Gotham.
- `ipFleck`: una cadena de caràcters que conté l'adreça IP del servidor Fleck, per a comunicacions.
- `portFleck`: un enter que representa el port associat amb el servidor Fleck.
- `directory`: una cadena de caràcters que especifica el directori on es troba la configuració.
- `workerType`: una cadena de caràcters que indica el tipus de treballador, per determinar el comportament de l'aplicació.

4 Funcions `readConfigFile`

Les funcions `readConfigFile` tenen un paper clau a l'hora de carregar les configuracions des dels fitxers `.dat`. A continuació, es presenta la versió de la funció per a cadascun dels quatre fitxers:

4.1 `readConfigFile` per a Fleck

```
1  // Funci per llegir el fitxer de configuraci i carregar la informaci a la estructura FleckConfig
2  /**********************************************
3  * @Finalitat: Llegeix el fitxer de configuraci especificat i emmagatzema la informaci a la estructura FleckConfig.
4  * @Par metres:
5  *   in: configFile = nom del fitxer de configuraci .
6  *   out: fleckConfig = estructura on s'emmagatzema la configuraci llegida.
7  * @Retorn: ----
8  **********************************************/
9  void readConfigFile(const char *configFile, FleckConfig *fleckConfig) {
10     int fd = open(configFile, O_RDONLY); // Obre el fitxer en mode nom s lectura
11     if (fd == -1) {
12         printF("Error obrint el fitxer de configuraci \n"); // Missatge d'error si no es pot obrir
13         exit(1); // Finalitza el programa en cas d'error
14     }
15
16     // Llegeix i assigna mem ria per a cada camp de la configuraci
17     fleckConfig->user = readUntil(fd, '\n'); // Llegeix el nom de l'usuari
18     removeChar(fleckConfig->user, '&'); // Elimina el car cter '&' del nom de l'usuari
19     fleckConfig->directory = trim(readUntil(fd, '\n')); // Elimina espais al voltant del directori
20     fleckConfig->ipGotham = readUntil(fd, '\n'); // Llegeix la IP del servidor Gotham
21
22     char *portStr = readUntil(fd, '\n'); // Llegeix el port com a cadena
23     fleckConfig->portGotham = atoi(portStr); // Converteix el port a enter
24     free(portStr); // Allibera la mem ria de la cadena temporal
25     close(fd); // Tanca el fitxer
26
27     fleckConfig->user = trim(fleckConfig->user); // Crear un buffer temporal per emmagatzemar el missatge
28     printF("\n");
29     printF(fleckConfig->user);
30     printF(" user initialized\n\n");
31     printF("File read correctly:");
32     printF("\nUser - ");
33     printF(fleckConfig->user);
34     printF("\nDirectory - ");
35     printF(fleckConfig->directory);
```

```
36              printF("\nIP - ");
37              printF(fleckConfig->ipGotham);
38
39              printF("\nPort - ");
40              char* portGothamStr = NULL;
41              asprintf(&portGothamStr, "%d\n", fleckConfig->portGotham); //
                    Converteix el port a cadena
42              printF(portGothamStr);
43              free(portGothamStr); // Allibera la mem ria de la cadena
                    temporal
44          }
```

Listing 5: Funció `readConfigFile` per a Fleck

La funció readConfigFile llegeix un fitxer de configuració per carregar la informació necessària dins de l'estructura FleckConfig. Comença obrint el fitxer especificat en mode només lectura utilitzant la funció open, i si no pot obrir el fitxer, mostra un missatge d'error i finalitza el programa. A continuació, la funció llegeix cada línia del fitxer fins al salt de línia mitjançant readUntil, que llegeix caràcter per caràcter fins trobar el delimitador especificat. La primera línia correspon al nom de l'usuari i s'emmagatzema al camp user de l'estructura, eliminant qualsevol caràcter '' que pugui estar present. La segona línia correspon al directori, s'aplica la funció trim per eliminar espais en blanc, i s'assigna al camp directory. La tercera línia conté la IP del servidor Gotham i s'emmagatzema al camp ipGotham. La quarta línia llegeix el port del servidor en format de text, que es converteix a un enter amb atoi i s'assigna al camp portGotham, alliberant la memòria utilitzada per la cadena de text temporal. Després de llegir i emmagatzemar les dades, es tanca el fitxer amb la funció close. La funció mostra la informació llegida per verificar que s'ha carregat correctament, incloent el nom d'usuari, el directori, la IP i el port de Gotham, assegurant-se també que el nom d'usuari no tingui espais en blanc no desitjats amb una nova aplicació de trim. Aquesta funció és clau per carregar la configuració inicial del sistema Fleck, ja que utilitza memòria dinàmica per gestionar cadenes de longitud variable i garanteix una correcta neteja de memòria per evitar fuites.

4.2 `readConfigFile` per a Harley

Funció semblant a la de `Fleck`, però assignada a `HarleyConfig`:

```
1         // Funci  per llegir el fitxer de configuraci  de Harley
2         /*********************************************
3         * @Finalitat: Llegeix el fitxer de configuraci  especificat i
              emmagatzema la informaci  a la estructura HarleyConfig.
4         * @Par metres:
5         *   in: configFile = nom del fitxer de configuraci .
6         *   out: harleyConfig = estructura on s'emmagatzema la configuraci
              llegida.
7         * @Retorn: ----
8         *********************************************/
9         void readConfigFile(const char *configFile, HarleyConfig *
              harleyConfig) {
10            int fd = open(configFile, O_RDONLY); // Obre el fitxer en mode
                  nom s lectura
11
12            if (fd == -1) {
13                printF("Error obrint el fitxer de configuraci \n"); //
                      Missatge d'error si no es pot obrir
14                exit(1); // Finalitza el programa en cas d'error
15            }
16
17            // Llegeix i assigna mem ria per a cada camp de la
                  configuraci
18            harleyConfig->ipGotham = readUntil(fd, '\n'); // Llegeix la IP
                  del servidor Gotham
19            char *portGotham = readUntil(fd, '\n'); // Llegeix el port com a
                   cadena de text
20            harleyConfig->portGotham = atoi(portGotham); // Converteix el
                  port a enter
21            free(portGotham); // Allibera la mem ria de la cadena temporal
22
23            harleyConfig->ipFleck = readUntil(fd, '\n'); // Llegeix la IP
                  del servidor Fleck
24            char* portFleck = readUntil(fd, '\n'); // Llegeix el port com a
                  cadena de text
25            harleyConfig->portFleck = atoi(portFleck); // Converteix el port
                   a enter
26            free(portFleck); // Allibera la mem ria de la cadena temporal
27
28            harleyConfig->directory = readUntil(fd, '\n'); // Llegeix el
                  directori de treball
29            harleyConfig->workerType = readUntil(fd, '\n'); // Llegeix el
                  tipus de treballador
30
31            close(fd); // Tanca el fitxer
32
33            // Mostra la configuraci  llegida per a verificaci
34            printF("Fitxer llegit correctament:\n");
35            printF("Gotham IP - ");
36            printF(harleyConfig->ipGotham);
37            printF("\nGotham Port - ");
38            char* portGothamStr = NULL;
39            asprintf(&portGothamStr, "%d", harleyConfig->portGotham); //
                  Converteix el port a cadena de text
40            printF(portGothamStr);
```

```
41              free(portGothamStr); // Allibera la mem ria de la cadena
                    temporal
42
43              printF("\nIP Fleck - ");
44              printF(harleyConfig->ipFleck);
45              printF("\nPort Fleck - ");
46              char* portFleckStr = NULL;
47              asprintf(&portFleckStr, "%d", harleyConfig->portFleck); //
                    Converteix el port a cadena de text
48              printF(portFleckStr);
49              free(portFleckStr); // Allibera la mem ria de la cadena
                    temporal
50
51              printF("\nDirectory - ");
52              printF(harleyConfig->directory);
53              printF("\nWorker Type - ");
54              printF(harleyConfig->workerType);
55              printF("\n");
56          }
```

Listing 6: Funció `readConfigFile` per a Harley

La funció readConfigFile per a Harley llegeix un fitxer de configuració i emmagatzema la informació llegida a l'estructura HarleyConfig. Primer, obre el fitxer especificat en mode només lectura utilitzant la funció open. Si no és possible obrir el fitxer, mostra un missatge d'error i finalitza l'execució del programa. La funció continua llegint les línies del fitxer una a una, utilitzant la funció readUntil, que llegeix fins al salt de línia. La primera línia conté la IP del servidor Gotham, que s'emmagatzema al camp ipGotham de l'estructura. La següent línia és el port del servidor Gotham, que es llegeix com una cadena de text, es converteix a un enter amb atoi, i es guarda a portGotham, alliberant després la memòria de la cadena temporal. Les següents dues línies segueixen el mateix procés per a la IP i el port del servidor Fleck, assignant els valors als camps ipFleck i portFleck, respectivament. Després, llegeix el directori de treball i el tipus de treballador, i els guarda als camps directory i workerType. Un cop s'ha llegit tota la informació, el fitxer es tanca amb la funció close.

4.3 `readConfigFile` per a Gotham

```
1  // Funci per llegir el fitxer de configuraci de Gotham
2  /*********************************************
3  * @Finalitat: Llegeix el fitxer de configuraci especificat i
       emmagatzema la informaci a la estructura GothamConfig.
4  * @Par metres:
5  *   in: configFile = nom del fitxer de configuraci .
6  *   out: gothamConfig = estructura on s'emmagatzema la configuraci
       llegida.
7  * @Retorn: ----
8  *********************************************/
9  void readConfigFile(const char *configFile, GothamConfig *
       gothamConfig) {
10     int fd = open(configFile, O_RDONLY); // Obre el fitxer en mode
           nom s lectura
11
12     if (fd == -1) {
13         printF("Error obrint el fitxer de configuraci \n"); //
               Missatge d'error si no es pot obrir
14         exit(1); // Finalitza el programa en cas d'error
15     }
16
17     // Llegeix i assigna mem ria per a cada camp de la
           configuraci
18     gothamConfig->ipFleck = readUntil(fd, '\n'); // Llegeix la IP
           del servidor Fleck
19     char* portFleck = readUntil(fd, '\n'); // Llegeix el port com a
           cadena
20     gothamConfig->portFleck = atoi(portFleck); // Converteix el port
           a enter
21     free(portFleck); // Allibera la mem ria de la cadena temporal
22
23     gothamConfig->ipHarEni = readUntil(fd, '\n'); // Llegeix la IP
           del servidor Harley Enigma
24     char *portHarEni = readUntil(fd, '\n'); // Llegeix el port com a
           cadena
25     gothamConfig->portHarEni = atoi(portHarEni); // Converteix el
           port a enter
26     free(portHarEni); // Allibera la mem ria de la cadena temporal
27
28     close(fd); // Tanca el fitxer
29
30     // Mostra la configuraci llegida per a verificaci
31     printF("IP Fleck - ");
32     printF(gothamConfig->ipFleck);
33     printF("\nPort Fleck - ");
34     char* portFleckStr = NULL;
35     asprintf(&portFleckStr, "%d", gothamConfig->portFleck); //
           Converteix el port a cadena de text
36     printF(portFleckStr);
37     free(portFleckStr); // Allibera la mem ria de la cadena
           temporal
38
39     printF("\nIP Harley Enigma - ");
40     printF(gothamConfig->ipHarEni);
41     printF("\nPort Harley Enigma - ");
42     char* portHarEniStr = NULL;
```

```
43              asprintf(&portHarEniStr, "%d\n", gothamConfig->portHarEni); //
                    Converteix el port a cadena de text
44              printF(portHarEniStr);
45              free(portHarEniStr); // Allibera la mem ria de la cadena
                    temporal
46          }
```

Listing 7: Funció `readConfigFile` per a Gotham

La funció readConfigFile per a Gotham té com a objectiu llegir el fitxer de configuració especificat i emmagatzemar la informació en una estructura GothamConfig. Comença obrint el fitxer en mode de lectura utilitzant la funció open, i si el fitxer no es pot obrir, es mostra un missatge d'error i el programa es finalitza amb exit per indicar un error crític.

La funció llegeix la configuració camp per camp. Primer, llegeix la IP del servidor Fleck amb la funció readUntil, i després llegeix el port associat com una cadena de text. Aquesta cadena es converteix a un enter mitjançant la funció atoi i s'assigna al camp portFleck de l'estructura. A continuació, allibera la memòria assignada per la cadena temporal que contenia el port.

Després, la funció continua llegint la IP i el port del servidor Harley Enigma, seguint el mateix procés. Llegeix la IP, després el port com una cadena de text, la converteix a un enter, l'assigna al camp portHarEni, i allibera la memòria de la cadena temporal.

Un cop s'ha llegit tota la informació necessària, la funció tanca el fitxer amb close. Finalment, es mostren per pantalla els valors llegits per verificar que s'han carregat correctament. Això inclou la impressió de les IPs i els ports de Fleck i Harley Enigma. Els ports es converteixen a cadena de text utilitzant asprintf per assegurar-se que es mostren correctament, i la memòria temporal utilitzada per aquestes conversions s'allibera després de cada ús.

4.4 `readConfigFile` per a Enigma

```
// Funci per llegir el fitxer de configuraci d'Enigma
/*********************************************
* @Finalitat: Llegeix el fitxer de configuraci especificat i
    carrega la informaci en una estructura EnigmaConfig.
* @Par metres:
*   in: configFile = nom del fitxer de configuraci .
*   out: enigmaConfig = estructura on s'emmagatzema la configuraci
    llegida.
* @Retorn: ----
*********************************************/
void readConfigFile(const char *configFile, EnigmaConfig *
   enigmaConfig) {
    int fd = open(configFile, O_RDONLY); // Obre el fitxer en mode
       nom s lectura

    if (fd == -1) {
        printF("Error obrint el fitxer de configuraci \n"); //
           Missatge d'error si no es pot obrir
        exit(1); // Finalitza el programa en cas d'error
    }

    // Llegeix i assigna la mem ria per a cada camp de la
       configuraci
    enigmaConfig->ipGotham = readUntil(fd, '\n'); // Llegeix la IP
       del servidor Gotham
    char* portGotham = readUntil(fd, '\n'); // Llegeix el port com a
        cadena de text
    enigmaConfig->portGotham = atoi(portGotham); // Converteix el
       port a enter
    free(portGotham); // Allibera la mem ria de la cadena temporal

    enigmaConfig->ipFleck = readUntil(fd, '\n'); // Llegeix la IP
       del servidor Fleck
    char *portFleck = readUntil(fd, '\n'); // Llegeix el port com a
       cadena de text
    enigmaConfig->portFleck = atoi(portFleck); // Converteix el port
        a enter
    free(portFleck); // Allibera la mem ria de la cadena temporal

    enigmaConfig->directory = readUntil(fd, '\n'); // Llegeix el
       directori de configuraci
    enigmaConfig->workerType = readUntil(fd, '\n'); // Llegeix el
       tipus de treballador

    close(fd); // Tanca el fitxer

    // Mostra la configuraci llegida per a verificaci
    printF("Ip Gotham - ");
    printF(enigmaConfig->ipGotham);
    printF("\nPort Gotham - ");
    char* portGothamStr = NULL;
    asprintf(&portGothamStr, "%d", enigmaConfig->portGotham); //
       Converteix el port a cadena de text
    printF(portGothamStr);
    free(portGothamStr); // Allibera la mem ria de la cadena
       temporal
```

```
41
42              printF("\nIp Fleck - ");
43              printF(enigmaConfig->ipFleck);
44              printF("\nPort Fleck - ");
45              char* portFleckStr = NULL;
46              asprintf(&portFleckStr, "%d", enigmaConfig->portFleck); //
                    Converteix el port a cadena de text
47              printF(portFleckStr);
48              free(portFleckStr); // Allibera la mem ria de la cadena
                    temporal
49
50              printF("\nDirectory - ");
51              printF(enigmaConfig->directory);
52
53              printF("\nWorker Type - ");
54              printF(enigmaConfig->workerType);
55              printF("\n");
56          }
```

Listing 8: Funció `readConfigFile` per a Enigma

La funció readConfigFile per a Enigma s'encarrega de llegir un fitxer de configuració i emmagatzemar la informació obtinguda en una estructura EnigmaConfig. Comença obrint el fitxer especificat en mode de lectura amb la funció open. Si el fitxer no es pot obrir, la funció mostra un missatge d'error i finalitza el programa mitjançant la funció exit.

Després, la funció llegeix cada línia del fitxer per obtenir els diferents paràmetres de configuració. Utilitza la funció readUntil per llegir fins al caràcter de salt de línia. La primera línia llegeix la IP del servidor Gotham i l'emmagatzema al camp ipGotham de l'estructura. La següent línia llegeix el port del servidor Gotham, que es converteix de cadena de text a enter amb atoi i s'assigna al camp portGotham, alliberant la memòria de la cadena temporal. A continuació, llegeix la IP i el port del servidor Fleck de manera similar, guardant aquests valors als camps ipFleck i portFleck, respectivament, i també alliberant la memòria de les cadenes temporals.

Un cop llegides aquestes dades, la funció continua amb la lectura del directori de configuració i el tipus de treballador, emmagatzemant els valors obtinguts als camps directory i workerType de l'estructura EnigmaConfig. Un cop tota la informació ha estat llegida i assignada, el fitxer de configuració es tanca amb la funció close.

Finalment, la funció mostra per pantalla els valors llegits per verificar que s'han carregat correctament. Per a això, imprimeix els valors de les IPs i ports de Gotham i Fleck, així com el directori i el tipus de treballador, utilitzant la funció printF. Aquesta verificació inclou convertir els ports a cadena de text utilitzant asprintf per assegurar-se que s'imprimeixen correctament.

5 Conclusió

En la Fase 1 del projecte "Mr. J. System", hem establert una arquitectura modular que permet la gestió i configuració fàcil dels diferents components del sistema. Les estructures de dades han estat dissenyades per ser coherents i flexibles, permetent l'ús de memòria dinàmica per gestionar diferents longituds d'entrada.

El treball en equip ha estat clau per a l'èxit del projecte. Hem utilitzat GitHub per gestionar el codi, amb revisions de codi i *pull requests* per assegurar la qualitat. Les reunions a Discord ens han permès resoldre problemes en temps real i coordinar les tasques de forma eficient.

En aquesta fase, hem assegurat que cada procés pugui evolucionar independentment i ser configurat fàcilment a partir de fitxers `.dat`. La modularitat del codi facilita la futura extensió i manteniment del sistema, assegurant que es poden afegir nous components o modificar els existents sense afectar negativament la resta del projecte.

Disseny i Implementació de la Fase 2

Sistema Gotham

AUTORS:

Pau Olea Reyes
Alfred Chávez Fernández

Assignatura: Sistemes Operatius
Universitat Ramon Llull – La Salle

December 15, 2024

1 *

2 Introducció

La segona fase del projecte del Sistema Gotham està enfocada a la integració de múltiples processos i la comunicació efectiva entre ells, utilitzant recursos del sistema com **sockets, fils (threads), mutexos**, i fitxers de configuració.

2.1 Esquema funcional de comunicació entre processos

La Figura 1 mostra com es comuniquen els diferents processos del sistema:

- **Gotham.exe:** Actua com a servidor central que rep peticions i coordina la distribució de tasques entre els treballadors.
- **Enigma.exe:** Un tipus de treballador que es registra a Gotham i processa tasques específiques de text.
- **Harley.exe:** Un altre tipus de treballador que processa contingut multimèdia, com àudio o imatges.
- **Fleck.exe:** Un treballador encarregat de tasques d'indexació o processament addicional.
- La comunicació es fa mitjançant sockets TCP i segueix un protocol definit amb missatges estructurats per garantir la compatibilitat.

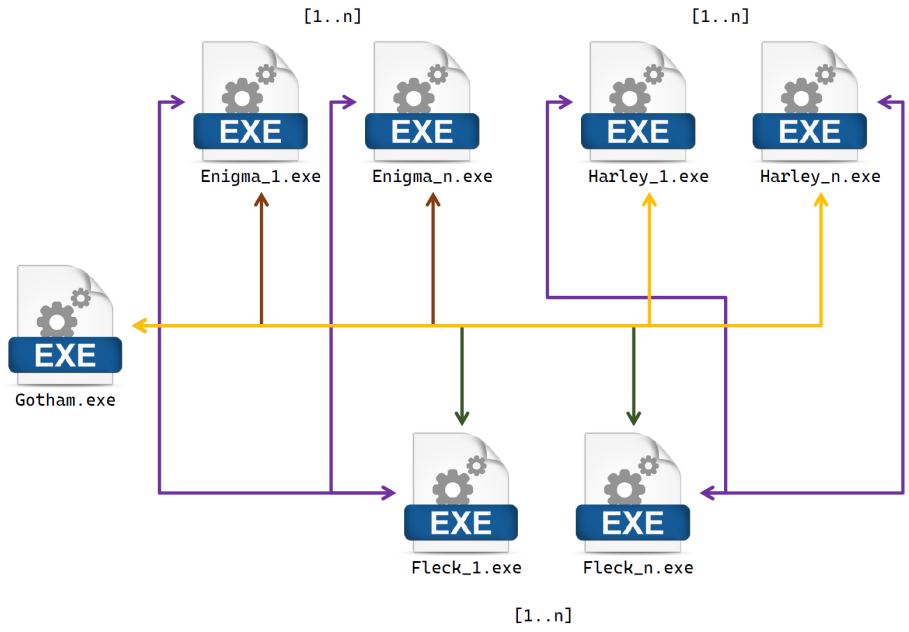

Figure 1: Esquema funcional de comunicació entre processos.

2.2 Estructura de la implementació de la fase 2

La Figura 2 presenta la jerarquia de fitxers i les seves dependències:

- Cada fitxer de configuració (amb extensió `.dat`) conté els paràmetres específics per al procés corresponent, com ara adreces IP, ports i tipus de treballadors.
- Els mòduls principals del sistema són `Gotham.c`, `Enigma.c`, `Harley.c` i `Fleck.c`.
- Els mòduls auxiliars (`Networking.c`, `FileReader.c`, `StringUtils.c`, `DataConversion.c`) proporcionen funcionalitats compartides, com la gestió de sockets, lectura de fitxers o manipulació de dades.
- El fitxer `Makefile` automatitza la compilació del sistema.

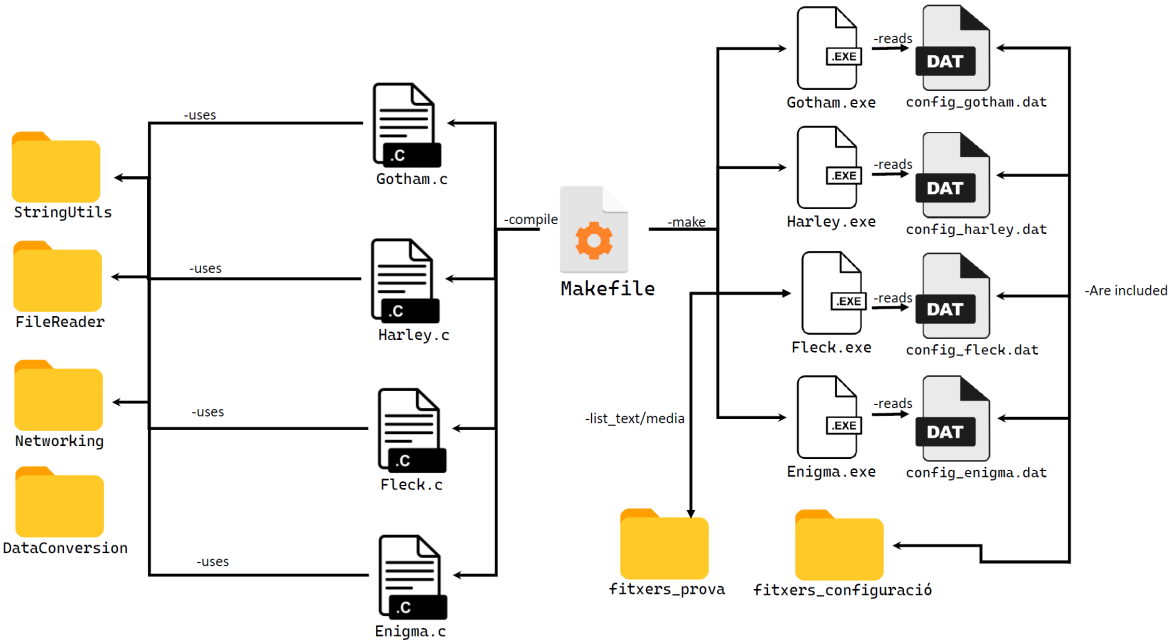

Figure 2: Estructura de la implementació de la fase 2.

3 Disseny del Sistema

3.1 Estructures de dades

Per a aquesta implementació, s'han definit estructures de dades clau per gestionar la informació dels *workers* i les comunicacions:

- **WorkerInfo:** Conté informació sobre la IP, port, tipus i descriptor de socket de cada *worker*.
- **WorkerManager:** Administra una llista dinàmica de *workers* amb accés sincronitzat mitjançant **mutexos**.

3.2 Recursos del sistema utilitzats

- **Sockets TCP:** Utilitzats per la comunicació entre processos (Gotham, Enigma, Harley, Fleck).
- **Threads:** Cada connexió amb un client és gestionada en un fil separat per millorar el rendiment.
- **Mutexos:** Garantitzen l'accés segur a les estructures compartides, com ara la llista de *workers*.
- **Fitxers de configuració:** Cada procés llegeix un fitxer de configuració específic que conté els paràmetres necessaris.

4 Implementació

4.1 Mòduls principals

El sistema s'ha dividit en diversos mòduls per garantir una millor modularitat i mantenibilitat:

- **Gotham.c:** Actua com a servidor central que coordina els *workers*.
- **Enigma.c, Harley.c i Fleck.c:** Clients que es connecten a Gotham per realitzar tasques específiques.
- **Networking.c:** Conté funcions genèriques per a la gestió de sockets i comunicacions.
- **FileReader.c:** Facilita la lectura de fitxers de configuració.
- **StringUtils.c i DataConversion.c:** Mòduls per al tractament de cadenes i conversió de dades.

4.2 Flux d'execució

1. **Inicialització:** Cada procés llegeix el seu fitxer de configuració i estableix connexions segons els paràmetres definits.
2. **Registre:** Els *workers* s'enregistren al servidor Gotham amb el seu tipus, IP i port.
3. **Processament:** Gotham delega tasques als *workers* segons el tipus de comanda rebuda.
4. **Finalització:** Els recursos es tanquen i la memòria dinàmica es gestiona adequadament.

5 Opcions implementades

- **Gestió de connexions simultànies:** Utilitzant *threads* per a cada connexió.
- **Suport per múltiples tipus de treballadors:** Enigma, Harley i Fleck poden processar diferents tasques.
- **Validació d'entrada:** Cada comanda és validada abans de processar-se per evitar errors.
- **Registre detallat:** Cada procés genera logs amb colors per facilitar el depurament.

6 Conclusió

La implementació de la fase 2 del projecte del Sistema Gotham ha estat un desafiament tècnic que ens ha permès aprofundir en conceptes fonamentals dels sistemes operatius, com la comunicació entre processos, l'ús de recursos del sistema i el disseny de programes escalables. Aquesta fase ens ha portat a dissenyar i implementar una arquitectura robusta i modular capaç de gestionar múltiples clients, tot mantenint una comunicació segura i eficient amb el servidor central, Gotham.

En primer lloc, hem aconseguit crear una estructura coherent basada en sockets TCP per establir comunicacions bidireccionals entre Gotham i els diversos tipus de treballadors: Enigma, Harley i Fleck. Aquesta elecció ha estat fonamental per garantir la fiabilitat de les transmissions, donant suport tant a la connexió com a l'intercanvi de dades. També hem implementat un sistema de gestió de connexions simultànies utilitzant fils (threads) en el servidor Gotham, que ens ha permès escalar el sistema per donar suport a múltiples clients de forma paral·lela. Aquesta estratègia ha estat clau per mantenir un bon rendiment en entorns amb altes càrregues de treball.

La integració de fitxers de configuració per a cada procés ha estat una decisió important en el nostre disseny. Aquest enfocament ens ha permès separar la lògica del codi de la configuració específica de cada component, facilitant així la portabilitat i la personalització del sistema. Els fitxers de configuració inclouen informació crucial com les adreces IP, els ports i els tipus de treballadors, i són gestionats mitjançant el mòdul 'FileReader'.

Durant aquesta fase, hem posat un èmfasi especial en garantir la sincronització i la integritat de les dades. Per aquest motiu, hem utilitzat mutexos per protegir les estructures compartides, com ara la llista dinàmica de treballadors, dins del servidor Gotham. Això ens ha permès evitar condicions de competició i assegurar que els recursos són gestionats de forma segura en un entorn concurrent.

Hem dissenyat els processos de manera que cadascun tingui una responsabilitat clara i ben definida. Gotham actua com el nucli central que coordina les accions dels treballadors i assigna tasques segons el tipus de comanda rebuda. Els treballadors, per la seva banda, compleixen rols específics (textual o multimèdia) i processen les tasques assignades de manera eficient. Aquesta distribució del treball ens ha permès desenvolupar un sistema escalable i fàcilment adaptable a futures extensions.

A més, hem implementat diverses funcionalitats opcionals que aporten un valor afegit al sistema. Per exemple, hem incorporat un sistema de validació d'entrades per assegurar que només es processen comandes vàlides. També hem creat un mecanisme de registre detallat amb codis de colors per a facilitar el depurament i el seguiment de les operacions del sistema.

Finalment, volem destacar que aquesta fase no només ha suposat un repte tècnic, sinó també una oportunitat per millorar les nostres habilitats de treball en equip i organització. La necessitat de coordinar diversos components, modularitzar el codi i gestionar recursos compartits ens ha obligat a treballar amb rigor i precisió, aspectes essencials en qualsevol projecte real de programari.

En resum, la implementació de la fase 2 ha estat un pas significatiu en el desenvolupament del Sistema Gotham, i estem orgullosos dels resultats obtinguts. El sistema resultant és robust, modular i escalable, i compleix els requisits plantejats en l'enunciat de la pràctica. A més, hem creat una base sòlida per afrontar futures ampliacions i millores, consolidant els coneixements adquirits i desenvolupant noves competències que seran invaluables en projectes futurs.

Pràctica: Mr. J. System

Fase 3: The Queen of Distortion

AUTORS:

Pau Olea Reyes
Alfred Chávez Fernández

Assignatura: Sistemes Operatius
Universitat Ramon Llull – La Salle

December 15, 2024

1 *

Índex

2 Introducció

La Fase 3 del projecte té com a objectiu implementar un sistema escalable i robust per a la distorsió de fitxers media. Aquesta memòria documenta el disseny, implementació i estructures de dades utilitzades, així com els protocols de comunicació entre processos. A més, s'inclouen diagrames explicatius de la comunicació i l'estructura del projecte.

3 Diagrama de Comunicació

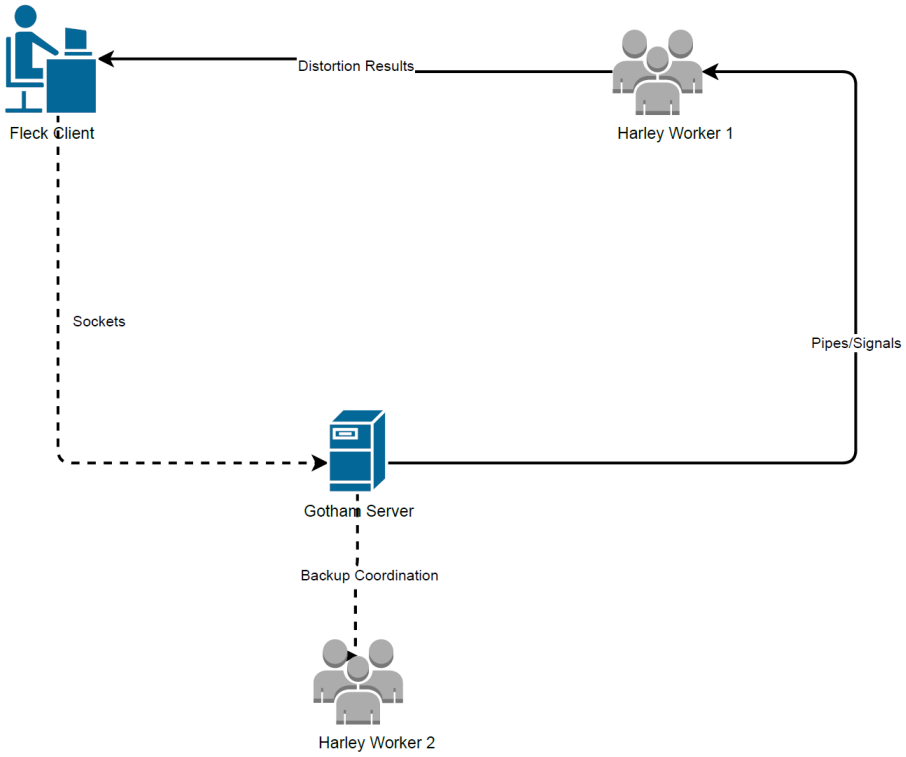

Figure 1: Diagrama de Comunicació entre Processos

El diagrama de la Figura 1 mostra les interaccions principals:

- **Fleck Client**: Actua com a interfície per als usuaris. Genera peticions de distorsió que són enviades al servidor central Gotham mitjançant sockets TCP.
- **Gotham Server**: Actua com a nucli del sistema. Coordina les peticions dels clients i distribueix les tasques de distorsió als Harley Workers. Gestiona també la tolerància a fallades i supervisa l'estat dels workers.
- **Harley Worker 1 i 2**: Els Harley són processos encarregats de dur a terme les distorsions. Un d'ells actua com a principal i l'altre com a backup, llestos per assumir la càrrega en cas de fallada.

Funcionament:

1. El client Fleck envia una petició al servidor Gotham, incloent el fitxer a distorsionar i els paràmetres corresponents.
2. Gotham selecciona un Harley disponible per processar la tasca. Aquesta comunicació es realitza mitjançant pipes.

3. Un cop completada la tasca, el Harley envia els resultats a Gotham, que els retorna al client Fleck.

4. En cas de fallada del Harley principal, Gotham coordina amb el backup per continuar processant les tasques pendents.

3.1 Descripció del Procés de Distorsió

El procés de la Fase 3 s'executa de la següent manera:

1. **Connexió inicial:** El procés Fleck s'inicia i es connecta al servidor central Gotham per enviar una petició de distorsió. Aquesta petició inclou informació del fitxer a processar i els paràmetres de la distorsió.

2. **Distribució de la tasca:** Gotham assigna la tasca a un dels Harley disponibles. Si hi ha múltiples Harley, Gotham fa ús d'un mecanisme de distribució per assignar la càrrega de manera equitativa.

3. **Transferència de dades per trames:**

   - Un cop assignada la tasca, Fleck estableix una connexió directa amb Harley i comença a enviar les dades del fitxer en fragments o trames.
   - Cada trama té una mida de 247 bytes, per deixar espai per a metadades com el checksum i el timestamp.
   - L'algoritme `MD5SUM` és utilitzat per verificar que les dades rebudes per Harley coincideixen amb les dades enviades per Fleck, garantint la integritat de la informació.

4. **Processament i compressió:**

   - Harley rep totes les trames, les reordena si cal, i crea el fitxer complet.
   - El fitxer és processat amb les eines de compressió disponibles, com la llibreria `SO_compression.h`.
   - El fitxer resultant es guarda al directori local de Harley (`harley_directory`).

5. **Planificació de descàrrega:**

   - Tot i que encara no està completament implementat, s'ha previst utilitzar eines com `FileZilla` per permetre la descàrrega remota del fitxer des del directori de Harley al client final.
   - Aquesta funcionalitat garanteix l'accés segur i controlat als fitxers resultants, integrant opcions per escalar a entorns reals.

Aquest procés assegura que els fitxers siguin gestionats amb robustesa, garantint la integritat de les dades i facilitant la compressió i transferència en entorns distribuïts.

4 Diagrama d'Estructura del Projecte

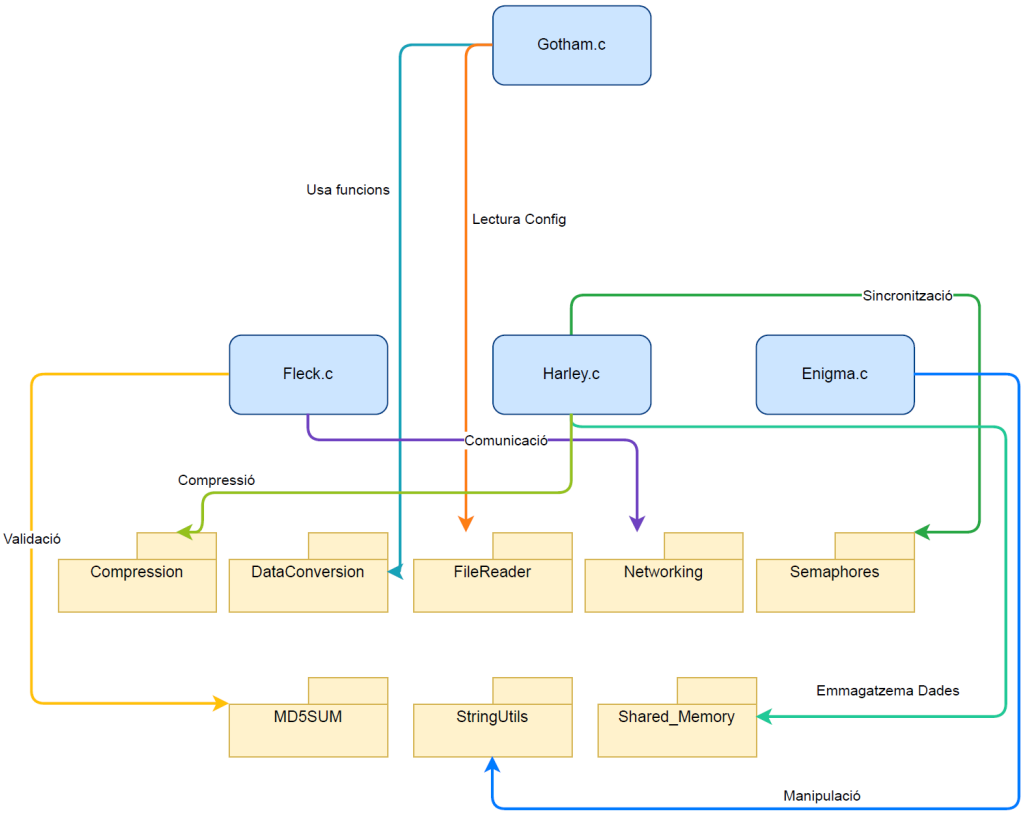

Figure 2: Diagrama d'Estructura del Projecte

El diagrama de la Figura 2 descriu l'organització modular del projecte. Els mòduls es divideixen de la següent manera:

- **Fitxers principals**: `Gotham.c`, `Harley.c`, `Fleck.c` i `Enigma.c`. Aquests fitxers contenen la lògica principal de cada procés i són el punt d'entrada del sistema.
- **Mòduls auxiliars**:
  - `FileReader`: Facilita la lectura de fitxers de configuració.
  - `Networking`: Gestiona la comunicació amb sockets TCP.
  - `Semaphores`: Implementa la sincronització entre processos.
  - `Compression` i `DataConversion`: Proporcionen les funcionalitats de compressió i modificació dels fitxers media.
  - `MD5SUM`: Valida la integritat dels fitxers mitjançant el càlcul de l'MD5.
  - `Shared_Memory`: Permet compartir dades entre processos de manera eficient.

5 Estructures de Dades

Aquesta secció descriu les estructures de dades clau implementades al projecte, juntament amb una justificació del seu ús i funcionalitat.

5.1 SharedMemory

```
typedef struct {
        int shmid;          // ID del segmento de memoria
            compartida
        void *shmaddr;      // Direcci n de la memoria
           compartida
        semaphore sem;      // Sem foro para
           sincronizaci n
} SharedMemory;
```

Listing 1: Estructura `SharedMemory`

Funció: Aquesta estructura gestiona la memòria compartida entre diversos processos.

5.2 SharedData

```
typedef struct {
        char fileName[256];    // Nom del fitxer
        char user[64];         // Usuari associat al
           fitxer
        char status[16];       // Estat del processament
            ("PENDING", "DONE", etc.)
        char result[1024];     // Resultat (MD5,
           validaci , etc.)
} SharedData;
```

Listing 2: Estructura `SharedData`

Funció: Emmagatzema informació de les peticions de fitxers i el seu estat.

5.3 Frame

```
typedef struct {
        uint8_t type;                   // Tipus de
           frame (dades, control, etc.)
        uint16_t data_length;           // Longitud de
           les dades
        char data[240];                 // Dades del
           frame
        uint16_t checksum;              // Verificaci
            d'integritat
        uint32_t timestamp;             // Marca de
           temps per ordenar frames
} Frame;
```

Listing 3: Estructura `Frame`

Funció: Transmet dades en paquets més petits i ordenats.

6 Recursos del Sistema Utilitzats

En aquesta secció es detallen els recursos del sistema utilitzats en la Fase 3 del projecte, destacant-ne la seva funcionalitat, implementació i la justificació de l'ús dins del sistema.

6.1 Threads

Els **threads** són fils d'execució dins d'un mateix procés que permeten la realització de múltiples tasques de manera concurrent.

- **Ús en Harley:** Els threads es fan servir en els processos Harley per gestionar múltiples peticions simultàniament. Per exemple, si arriben diverses tasques de distorsió alhora, cada tasca s'assigna a un thread independent per maximitzar l'eficiència i evitar bloquejos del sistema.
- **Avantatges:**
  - *Compartició de memòria:* Com que els threads comparteixen la mateixa memòria del procés pare, poden accedir fàcilment a les estructures de dades compartides com `SharedMemory`.
  - *Baix cost de creació:* Els threads són més lleugers que els processos i tenen un menor cost de creació i gestió.
  - *Execució paral·lela:* Permeten que Harley pugui gestionar múltiples connexions o peticions al mateix temps, aprofitant els processadors multinucli.
- **Implementació:** Els threads es creen amb la llibreria `pthread.h`. Exemple d'ús:

```
1                          pthread_t thread_id;
2                          pthread_create(&thread_id, NULL,
                              processTask, (void*) taskData);
```

Listing 4: Creació de threads en Harley

6.2 Semàfors

Els **semàfors** són mecanismes de sincronització que asseguren l'accés exclusiu a recursos compartits en entorns multithread o multiprocés.

- **Ús en Harley i Gotham:** Es fan servir per sincronitzar l'accés a la memòria compartida entre múltiples processos. Per exemple, quan diversos Harley accedeixen a la mateixa `SharedMemory`, un semàfor bloqueja l'accés mentre un procés està modificant les dades.
- **Avantatges:**
  - *Prevenció de condicions de competència:* Eviten que dos processos o threads modifiquin les mateixes dades al mateix temps, eliminant possibles errors.
  - *Coordinació entre processos:* Asseguren que l'ordre de les operacions sigui correcte i que no es produeixin inconsistències.
- **Implementació:** L'ús de semàfors es basa en la llibreria POSIX (`semaphore.h`). Exemple d'ús:

```
1                          sem_t semaphore;
2                          sem_init(&semaphore, 1, 1); // Inicialitza
                               el sem for
3
4                          sem_wait(&semaphore); // Bloqueja l'acc s
5                          // Modifica les dades
6                          sem_post(&semaphore); // Allibera l'acc s
```

Listing 5: Ús de semàfors per accedir a memòria compartida

6.3 Sockets

Els **sockets** són canals de comunicació utilitzats per intercanviar dades entre processos que s'executen en màquines diferents o en la mateixa màquina.

- **Ús entre Gotham i Fleck:** Els sockets TCP connecten Gotham amb Fleck per transmetre peticions i resultats. Això permet la comunicació bidireccional fiable, on Gotham pot rebre fitxers i retornar-ne els resultats al client.
- **Avantatges:**
  - *Fiabilitat:* El protocol TCP assegura que totes les dades s'envien i es reben correctament.
  - *Flexibilitat:* Els sockets poden connectar processos en màquines remotes, facilitant l'escalabilitat del sistema.
- **Implementació:** Els sockets es creen amb la llibreria `sys/socket.h`. Exemple:

```
1 int server_socket = socket(AF_INET,
      SOCK_STREAM, 0);
2 bind(server_socket, (struct sockaddr*)&
      server_addr, sizeof(server_addr));
3 listen(server_socket, 10);
4 int client_socket = accept(server_socket,
      NULL, NULL);
```

Listing 6: Creació d'un socket TCP en Gotham

6.4 Memòria Compartida

La **memòria compartida** és un mecanisme que permet que diversos processos accedeixin a la mateixa regió de memòria, millorant l'eficiència i velocitat de la comunicació.

- **Ús entre Gotham i Harley:** Es fa servir per compartir dades com l'estat de les tasques i els resultats generats per Harley, evitant la necessitat de còpies addicionals de dades.
- **Avantatges:**
  - *Alta velocitat:* És molt més ràpida que altres mecanismes com pipes o sockets, ja que no implica operacions d'E/S.
  - *Compartició de dades:* Tots els processos poden veure i modificar les mateixes dades, fent que sigui ideal per a estructures com `SharedData`.
- **Implementació:** La memòria compartida es gestiona amb `shmget`, `shmat` i `shmdt`. Exemple:

```
1 int shmid = shmget(IPC_PRIVATE, sizeof(
      SharedData), IPC_CREAT | 0666);
2 SharedData *data = (SharedData*) shmat(
      shmid, NULL, 0);
3 // Acc s i modificaci  de dades
4 shmdt(data); // Desvincula la mem ria
      compartida
```

Listing 7: Creació de memòria compartida

6.5 Forks

Els **forks** són utilitzats per crear processos fills independents a partir d'un procés pare.

- **Ús en Gotham i Harley:** Quan Gotham ha de gestionar una nova petició o Harley ha de processar una tasca, es crea un procés fill per executar aquesta tasca sense afectar el procés pare.
- **Avantatges:**
  - *Independència:* Els processos fills hereten l'estat inicial del procés pare, però després són completament independents.
  - *Aïllament:* Si un procés fill falla, no afecta el procés pare ni altres fills.
- **Implementació:** Els forks es creen amb la funció `fork()`:

```
pid_t pid = fork();
if (pid == 0) {
        // Codi del proc s fill
        executeTask();
} else {
        // Codi del proc s pare

}
```

Listing 8: Creació de processos amb fork()

Amb aquests recursos, el sistema garanteix escalabilitat, robustesa i un ús eficient dels recursos del sistema operatiu per satisfer els requisits de la Fase 3 del projecte.

7 Funcions Principals

7.1 handleFleckConnection

```
void handleFleckConnection(int clientSocket) {
        char buffer[1024];
        int bytesReceived = recv(clientSocket, buffer,
            sizeof(buffer), 0);
        if (bytesReceived > 0) {
                // Processar la petici  i assignar a
                    un Harley Worker
                assignTaskToHarley(buffer);
        }
        close(clientSocket);
}
```

Listing 9: Codi de recepció de peticions a Gotham

Funció: Gestiona connexions de Fleck, processa la petició i delega la tasca.

7.2 updateSharedMemory

```
sem_t *semaphore = sem_open("/harley_semaphore",
    O_CREAT, 0644, 1);

void updateSharedMemory(SharedData *data) {
        sem_wait(semaphore); // Bloqueja l'acc s
        strcpy(data->status, "IN_PROGRESS");
        printF("Actualitzant mem ria compartida...\n")
            ;
        sem_post(semaphore); // Allibera l'acc s
}
```

Listing 10: Codi de sincronització amb semàfors

Funció: Actualitza l'estat de memòria compartida amb protecció de concurrència.

7.3 handleHarleyFailure

```
void handleHarleyFailure() {
        printF("Harley principal ha fallat! Assignant
            el backup...\n");
        int backupHarley = findBackupHarley();
        if (backupHarley != -1) {
                notifyHarley(backupHarley, "
                    RESUME_TASKS");
        } else {
                printF("No hi ha backups disponibles!\n
                    ");
        }
}
```

Listing 11: Codi de recuperació davant fallades

Funció: Detecta fallades, assigna el backup i assegura continuïtat.

8 Conclusió

La Fase 3 del projecte Mr. J. System representa un avanç significatiu en el desenvolupament d'un sistema distribuït escalable, robust i tolerant a fallades per a la gestió i distorsió de fitxers media. Aquesta implementació ha suposat un repte tècnic que ens ha permès aprofundir en conceptes fonamentals dels sistemes operatius i les comunicacions interprocessos, alhora que hem aplicat metodologies modernes de programació i disseny modular.

En primer lloc, el disseny arquitectònic ha estat clau per assolir els objectius del projecte. La separació clara entre els processos principals (Gotham, Harley, Fleck) i els mòduls auxiliars (Networking, FileReader, Compression, etc.) ha permès garantir la mantenibilitat i extensibilitat del sistema. Aquesta modularitat no només facilita l'addició de noves funcionalitats en el futur, sinó que també simplifica la depuració i millora l'eficiència del desenvolupament.

A més, els mecanismes de comunicació i sincronització implementats han jugat un paper fonamental en el bon funcionament del sistema. L'ús de sockets TCP per a la comunicació entre processos distribuïts ha assegurat una transmissió fiable de dades entre Gotham i Fleck. D'altra banda, la memòria compartida i els semàfors han garantit la coherència i integritat de les dades en entorns concurrentes, mentre que els pipes i els senyals han facilitat una coordinació fluida entre Gotham i els processos Harley.

L'ús de threads dins dels Harley ha permès gestionar múltiples peticions simultàniament, aprofitant així els avantatges dels sistemes multinucli i millorant significativament el rendiment. També hem aplicat estratègies de tolerància a fallades, com l'activació automàtica d'un Harley de backup en cas de fallada del principal, la qual cosa assegura la continuïtat del servei fins i tot en situacions imprevistes.

Destaquem, també, la implementació de recursos opcionals que han aportat un valor afegit al sistema. La integració de la llibreria `SO_compression.h` per a la distorsió de fitxers i l'ús de validació MD5SUM per garantir la integritat dels fitxers han demostrat el compromís amb l'excel·lència tècnica i la seguretat de les dades. Aquests elements opcionals no només compleixen amb els requisits inicials del projecte, sinó que també amplien les capacitats del sistema per abordar problemes del món real.

Pel que fa al treball en equip, aquesta fase ens ha obligat a coordinar-nos de manera efectiva, aplicant eines de gestió de versions com Git per garantir un flux de treball ordenat i col·laboratiu. Les reunions periòdiques per discutir problemes tècnics i compartir progressos han estat crucials per mantenir una alineació constant amb els objectius del projecte.

Finalment, cal destacar que la implementació de la Fase 3 ens ha preparat per a reptes futurs en el desenvolupament de sistemes distribuïts i la gestió de recursos compartits. Hem adquirit experiència pràctica en l'ús de mecanismes del sistema operatiu com forks, threads, semàfors i sockets, i hem consolidat habilitats fonamentals per a la resolució de problemes en entorns de programació concurrent i distribuïda.

En resum, aquesta fase del projecte ha estat una oportunitat única per integrar i aplicar

coneixements tècnics avançats en un context realista, mentre dissenyàvem un sistema sòlid, escalable i preparat per a escenaris crítics. El resultat obtingut és un sistema que no només compleix amb les especificacions inicials, sinó que també serveix com una base robusta per a futures ampliacions i millores.