

Bensaid Hicham

January 20, 2025

Contents

1	Objectif du travail	3
2	Étapes du travail	3
2.1	1ère étape : Explication détaillée du problème	3
2.2	2ème étape : Présentation générale des pipes et leur lien avec le projet	3
2.3	3ème étape : Solution et implémentation du code	4
2.4	Explication détaillée du code	4
2.4.1	Inclusion des bibliothèques et définition des constantes	4
2.4.2	Déclaration des variables principales	5
2.4.3	Gestion des stations actives	5
2.4.4	Création des pipes	6
2.4.5	Création des processus fils (stations)	6
2.4.6	Code exécuté par les processus enfants	7
2.4.7	Boucle principale de chaque station	7
2.4.8	Gestion des messages	7
2.4.9	Initialisation et simulation de panne	8
2.4.10	Attente des processus enfants	9
2.4.11	Résumé	9
2.5	5ème étape : Tests et exécution du code	9
3	Conclusion	12

1 Objectif du travail

L'objectif de ce travail est de simuler un réseau Token Ring à l'aide du langage C. Chaque station du réseau sera représentée par un processus indépendant, et les communications entre les stations seront réalisées via des pipes. Le jeton circulera dans le réseau, permettant à une station de transmettre des données lorsqu'elle le reçoit. Ce projet inclut la mise en œuvre du simulateur, son test et sa validation, en tenant compte des pannes éventuelles des stations.

2 Étapes du travail

2.1 1ère étape : Explication détaillée du problème

Dans cette étape, il est important de comprendre le fonctionnement d'un réseau *Token Ring*, constitué de stations interconnectées où un jeton circule. Le jeton permet à une station de transmettre des données. Chaque station attend de recevoir le jeton avant d'envoyer ses données, puis passe le jeton à la station suivante. Cela évite les collisions et gère l'accès exclusif à la transmission.

Notre simulation recrée ce mécanisme en C, où chaque station reçoit et transmet des données sous l'autorité du jeton. En cas de panne d'une station, le jeton sera redirigé vers une station active pour maintenir la continuité du réseau.

2.2 2ème étape : Présentation générale des pipes et leur lien avec le projet

Les *pipes* sont un mécanisme de communication entre processus dans le système d'exploitation. Dans le cadre de ce projet, chaque station sera représentée par un processus distinct. Les *pipes* permettent à ces processus de communiquer entre eux en transmettant des messages, y compris le jeton et les données.

Les *pipes* sont utilisés de manière à ce que chaque station puisse recevoir le jeton via un pipe, effectuer des opérations de transmission, puis passer le jeton au processus suivant à travers un autre pipe. Ce mécanisme permet de simuler la communication entre les stations du réseau *Token Ring* sans avoir à recourir à des mécanismes plus complexes comme les sockets. En utilisant des *pipes*, chaque station lit les messages du pipe de la station précédente et écrit dans le pipe de la station suivante. Ce modèle simplifie la

gestion de la communication dans le cadre d'un réseau circulaire de stations sans nécessiter de gestion complexe de connexions réseau.

2.3 3ème étape :Solution et implémentation du code

Le code simule un réseau **Token Ring** avec plusieurs stations, où chaque station passe un jeton et, si nécessaire, transmet des messages à la station suivante. Le principe du **token passing** permet de gérer l'accès au réseau en faisant circuler un jeton entre les stations.

Ce code, implémenté dans le fichier `pipe.c` et disponible dans le repository GitHub associé au projet qui a pour lien (<https://github.com/NASRHOUDA/ATELIER>), permet de :

- Créer les pipes nécessaires à la communication entre les stations ;
- Créer un processus pour chaque station, gérant la réception du jeton, la transmission des trames et la redirection du jeton ;
- Implémenter l'algorithme de circulation du jeton, où chaque station attend de recevoir le jeton avant d'envoyer ses données. Si une station n'a pas de données, elle passe le jeton à la station suivante.

2.4 Explication détaillée du code

Le code ci-dessus simule un réseau **Token Ring** avec plusieurs stations, où chaque station passe un jeton et, si nécessaire, transmet des messages à la station suivante. Le fonctionnement suit le principe du **token passing**, un mécanisme où un jeton (un message spécial) circule dans un anneau de stations pour gérer l'accès au réseau.

2.4.1 Inclusion des bibliothèques et définition des constantes

Bibliothèques:

- `stdio.h` : Pour les fonctions d'entrée/sortie (ex : `printf`, `perror`).
- `unistd.h` : Pour les fonctions système (ex : `pipe`, `fork`, `read`, `write`).
- `stdlib.h` : Pour des fonctions utilitaires (ex : `exit`).
- `string.h` : Pour manipuler des chaînes de caractères.
- `stdbool.h` : Pour utiliser le type booléen (`true`, `false`).

- `sys/wait.h` : Pour gérer la synchronisation des processus (ex : `wait`).

Les constantes suivantes sont définies :

- `TOKEN` : Une chaîne représentant le jeton circulant entre les stations.
- `BUFFER_SIZE` : Taille maximale du buffer pour les messages échangés.
- `NUM_STATIONS` : Nombre total de stations participant à l'échange.

2.4.2 Déclaration des variables principales

```
int pipes[NUM_STATIONS][2];
pid_t pids[NUM_STATIONS];
char buffer[BUFFER_SIZE];
```

Description :

- `pipes` : tableau contenant les descripteurs des pipes pour chaque station. Chaque station a un pipe d'entrée/sortie.
- `pids` : tableau pour stocker les identifiants des processus enfants créés par `fork`.
- `buffer` : utilisé pour stocker les messages échangés entre les stations.

2.4.3 Gestion des stations actives

```
bool station_active[NUM_STATIONS]; // Tableau pour surveiller les stations actives

void simulate_failure(int station_id) {
    station_active[station_id] = false;
    printf("Station %d : Défaillance simulée.\n", station_id);
}
```

Description :

- **Tableau `station_active`** : Permet de surveiller l'état de chaque station (active ou en panne) et est initialisé avec `true` pour indiquer que toutes les stations sont actives au départ.
- **Fonction `simulate_failure`** : Marque une station comme inactive en mettant `station_active[station_id]` à `false`, et affiche un message indiquant la panne.

2.4.4 Création des pipes

```
for (int i = 0; i < NUM_STATIONS; i++) {
    if (pipe(pipes[i]) == -1) {
        perror("Erreur lors de la création des pipes");
        exit(EXIT_FAILURE);
    }
}
```

Pipes :

- pipes[i][0] : Lecture.
- pipes[i][1] : Écriture.

Chaque station a son propre pipe pour recevoir des données de la station précédente et envoyer des données à la suivante.

Si la création d'un pipe échoue, une erreur est affichée et le programme se termine.

2.4.5 Création des processus fils (stations)

```
for (int i = 0; i < NUM_STATIONS; i++) {
    pids[i] = fork();
    if (pids[i] == -1) {
        perror("Erreur lors du fork");
        exit(EXIT_FAILURE);
    }
}
```

Description :

- Boucle : crée un processus enfant pour chaque station.
- fork : renvoie 0 dans le processus enfant.
- Renvoie le PID du processus enfant dans le processus parent.
- Si le fork échoue, une erreur est affichée.

2.4.6 Code exécuté par les processus enfants

```
if (pids[i] == 0) {
    int station_id = i;
    close(pipes[station_id][1]);
    close(pipes[(station_id + 1) % NUM_STATIONS][0]);
}
```

Description :

- Chaque processus enfant :
 - `station_id` : identifie la station.
 - Ferme :
 - * L'extrémité d'écriture de son propre pipe (`pipes[station_id][1]`).
 - * L'extrémité de lecture du pipe de la station suivante (`pipes[(station_id + 1) % NUM_STATIONS][0]`).

2.4.7 Boucle principale de chaque station

```
while (true) {
    if (read(pipes[station_id][0], buffer, BUFFER_SIZE) <= 0) {
        continue;
    }
}
```

Description :

- Chaque station écoute en continu les messages entrants via `read`.
- Si aucune donnée n'est reçue, elle passe à l'itération suivante.

2.4.8 Gestion des messages

```
}if (strcmp(buffer, TOKEN) == 0) {
    printf("Station %d : Jeton reçu.\n", station_id);

    if (station_id == 1) {
        printf("Station %d : Transmission d'une trame.\n", station_id);
        write(pipes[(station_id + 1) % NUM_STATIONS][1], "Message de Station");
    }
}
```

```

    } else {
        printf("Station %d : Aucun message. Jeton passé.\n", station_id);
        int next_station = (station_id + 1) % NUM_STATIONS;
        while (!station_active[next_station]) {
            next_station = (next_station + 1) % NUM_STATIONS;
        }
        write(pipes[next_station][1], TOKEN, strlen(TOKEN) + 1);
    }
}

```

Cas du jeton

```

if (station_id == 1) {
    printf("Station %d : Transmission d'une trame.\n", station_id);
    write(pipes[(station_id + 1) % NUM_STATIONS][1], "Message de Station 1"
}

```

Description : Si le jeton est reçu : Station 1 envoie un message spécifique et Autres stations passent le jeton à la station suivante opérationnelle.

Cas d'un message : Description : Affiche le message reçu. et passe le jeton à la prochaine station active.

2.4.9 Initialisation et simulation de panne

```

for (int i = 0; i < NUM_STATIONS; i++) {
    close(pipes[i][0]);
}

sleep(5);
simulate_failure(3);
write(pipes[0][1], TOKEN, strlen(TOKEN) + 1);
printf("Processus principal : Jeton initial envoyé à la station 0.\n");

```

Description :

- Le processus principal ferme ses lectures pour éviter des conflits.
- Simule une panne après 5 secondes (station 3). 3 Envoie le jeton initial à la station 0.

2.4.10 Attente des processus enfants

```
for (int i = 0; i < NUM_STATIONS; i++) {  
    wait(NULL);  
}
```

Description :

- Le processus principal attend que tous les processus enfants se terminent avant de quitter.

2.4.11 Résumé

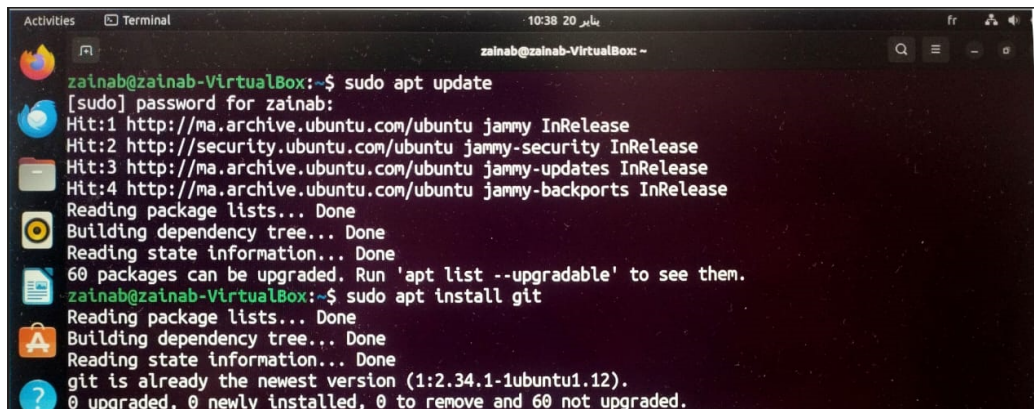
Ce programme simule un réseau Token Ring en utilisant des processus et des pipes. Chaque station reçoit un jeton, traite ou transmet un message, puis le passe à la station suivante. La station 1 envoie un message spécifique lorsqu'elle reçoit le jeton. Le programme gère également la simulation de pannes en contournant les stations défectueuses.

2.5 5ème étape : Tests et exécution du code

- **Vérification du passage du jeton** : Nous avons vérifié si le jeton circulait correctement entre les stations. Chaque station devait soit transmettre un message, soit passer le jeton à la station suivante, selon le cas.
- **Transmission des données** : Certaines stations étaient configurées pour envoyer des messages. Nous avons testé si ces messages étaient correctement envoyés, reçus et affichés par les stations suivantes.
- **Gestion des pannes** : Nous avons simulé des pannes de certaines stations pour observer si le jeton était correctement redirigé vers les stations actives, garantissant ainsi la continuité du fonctionnement du réseau.

La compilation et l'exécution du code se font via Git avec les étapes suivantes :

Installation de git sur Linux

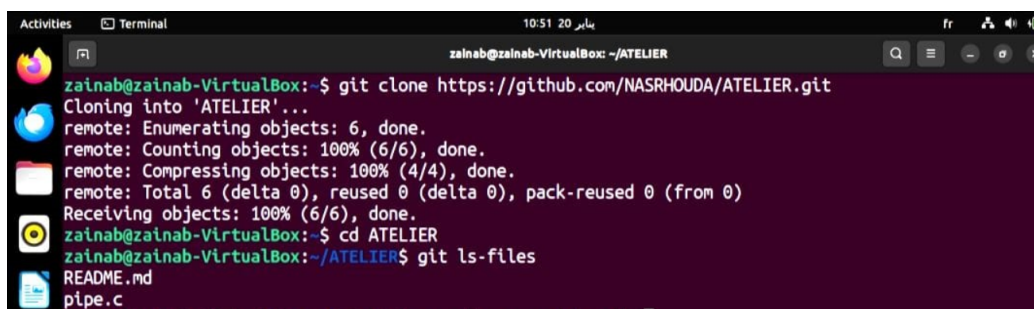


```
zainab@zainab-VirtualBox:~$ sudo apt update
[sudo] password for zainab:
Hit:1 http://ma.archive.ubuntu.com/ubuntu jammy InRelease
Hit:2 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:3 http://ma.archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:4 http://ma.archive.ubuntu.com/ubuntu jammy-backports InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
60 packages can be upgraded. Run 'apt list --upgradable' to see them.
zainab@zainab-VirtualBox:~$ sudo apt install git
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
git is already the newest version (1:2.34.1-1ubuntu1.12).
0 upgraded, 0 newly installed, 0 to remove and 60 not upgraded.
```

Figure 1

Cloner le repository Git :

```
git clone https://github.com/NASRHOUDA/ATELIER.git
```

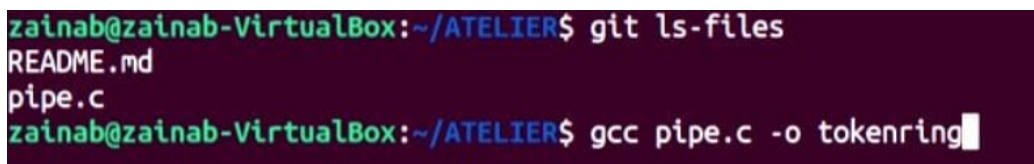


```
zainab@zainab-VirtualBox:~$ git clone https://github.com/NASRHOUDA/ATELIER.git
Cloning into 'ATELIER'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
zainab@zainab-VirtualBox:~$ cd ATELIER
zainab@zainab-VirtualBox:~/ATELIER$ git ls-files
README.md
pipe.c
```

Figure 2

Compiler le code : Une fois qu'on a cloné le repository ATELIER, il faut se rendre dans le dossier du projet et compiler le fichier `pipe.c` (où le code est présent) pour générer l'exécutable `tokenring`. On peut le faire avec la commande suivante :

```
gcc -o pipe.c tokenring
```



```
zainab@zainab-VirtualBox:~/ATELIER$ git ls-files
README.md
pipe.c
zainab@zainab-VirtualBox:~/ATELIER$ gcc pipe.c -o tokenring
```

Figure 3

Exécuter le simulateur : Après la compilation, on exécute le programme avec :

```
./tokenring
```

```
zainab@zainab-VirtualBox:~/ATELIER$ gcc pipe.c -o tokenring
zainab@zainab-VirtualBox:~/ATELIER$ ./tokenring
```

Figure 4

Résultat de l'exécution: Lors de l'exécution de ce programme, chaque station (représentée par un processus enfant) reçoit un jeton via un pipe, effectue une action en fonction de ce jeton (soit transmettre un message, soit simplement passer le jeton à la station suivante). La station 1, lorsqu'elle reçoit le jeton, envoie un message spécifique aux autres stations. Les autres stations, quant à elles, se contentent de passer le jeton sans transmettre de message. Le processus principal initialise le jeton et attend la fin des processus enfants avant de se terminer. Ce programme simule une communication circulaire en anneau entre plusieurs stations en utilisant des pipes et des processus.

```
Activities Terminal 10:54 20 يناير fr
zainab@zainab-VirtualBox: ~/ATELIER
Station 0 : Aucun message. Jeton passé.
Station 1 : Jeton reçu.
Station 1 : Transmission d'une trame.
Station 2 : Message reçu - "Message de Station 1"
Station 3 : Jeton reçu.
Station 3 : Aucun message. Jeton passé.
Station 4 : Jeton reçu.
Station 4 : Aucun message. Jeton passé.
Station 5 : Jeton reçu.
Station 5 : Aucun message. Jeton passé.
Station 0 : Jeton reçu.
Station 0 : Aucun message. Jeton passé.
Station 1 : Jeton reçu.
Station 1 : Transmission d'une trame.
Station 2 : Message reçu - "Message de Station 1"
Station 3 : Jeton reçu.
Station 3 : Aucun message. Jeton passé.
Station 4 : Jeton reçu.
Station 4 : Aucun message. Jeton passé.
Station 5 : Jeton reçu.
Station 5 : Aucun message. Jeton passé.
Station 0 : Jeton reçu.
Station 0 : Aucun message. Jeton passé.
Station 1 : Jeton reçu.
Station 1 : Transmission d'une trame.
Station 2 : Message reçu - "Message de Station 1"
Station 3 : Jeton reçu.
Station 3 : Aucun message. Jeton passé.
```

Figure 5

3 Conclusion

Le simulateur de réseau Token Ring avec pipes a permis de modéliser la circulation du jeton entre les nœuds et de simuler la transmission de données. Les tests ont montré que le système fonctionne correctement avec différents nombres de nœuds et en cas de pannes. Cette approche permet de mieux comprendre le fonctionnement d'un réseau Token Ring en utilisant des mécanismes de communication inter-processus. Le travail a également permis d'appliquer des concepts de programmation en C et de gestion de processus à un problème réel.