BORDEAUX INP Enseirb-Matmeca

Rapport

# Projet Système et Réseau (PR204) Implémentation d'une Mémoire Partagée Distribuée (DSM)

**Rédigé par :**
Mehdi Khabouze
Yassine Hamed

**Encadrants :**
Joachim Bruneau-Queyreix,
Philippe Swartvagher,
Guillaume Mercier.

Décembre 2024

# Table des matières

# 1 Introduction

Ce projet s'inscrit dans le cadre du module PR204 et porte sur l'implémentation d'un système de mémoire partagée distribuée (DSM). L'objectif principal est de permettre à plusieurs processus, répartis sur différentes machines, d'accéder à une zone mémoire partagée comme s'ils fonctionnaient sur un même système. Pour atteindre cet objectif, nous avons divisé notre travail en deux grandes phases. La première phase (dsmexec/dsmwrap) consiste à déployer et initialiser les processus distants, en assurant la communication avec un lanceur central. La seconde phase (DSM) se concentre sur la gestion des pages mémoire, leur protection et leur transfert dynamique entre les processus via des sockets.

Ce rapport présente les choix techniques et les différentes étapes d'implémentation.

# 2 Phase1 : dsmexec

## 2.1 Gestion des processus enfants et des signaux

Pour la gestion des processus enfants, nous avons choisi d'utiliser la fonction `fork()` afin de créer un processus pour chaque machine et pour paralléliser les connexions.

La gestion des signaux repose sur la fonction `sigaction()`, configurée pour intercepter les signaux `SIGCHLD`. Ce choix technique permet de gèrer la terminsaison des processus enfants, pour éviter les processus zombies. Grâce à une boucle utilisant `waitpid()` avec l'option `WNOHANG`, nous surveillons et nettoyons les processus terminés de manière non bloquante. Cela garantit que le processus parent reste réactif et disponible pour gérer d'autres tâches, même lorsqu'un grand nombre de processus enfants sont en cours d'exécution.

## 2.2 Lecture et traitement des fichiers de machines

Pour gérer la lecture des machines, nous avons mis en place une fonction `trim_whitespace()` qui nettoie les espaces au début et à la fin de chaque ligne afin d'avoir des noms de machines bien formatés avant leur traitement, pour éviter des bugs subtils lors de l'établissement des connexions.

Ensuite, on stocke ces informations dans la structure dynamique `proc_array`. Étant donné que chaque élément de ce tableau contient des détails sur une machine, comme son nom, son rang et les descripteurs de fichiers pour la communication, nous avons utilisé `realloc()` pour adapter la taille du tableau au nombre de machines détectées, ce qui est nécessaire si ce nombre varie d'un fichier `machine_file` à l'autre. Ce choix technique rend le code flexible et facile à maintenir, puisqu'il suffit d'ajouter une nouvelle machine dans le fichier pour qu'elle soit automatiquement prise en compte. De plus, on initialise chaque champ important (port, PID, etc.) à des valeurs par défaut pour éviter des comportements imprévisibles plus tard dans l'exécution du programme.

## 2.3 Création et gestion des sockets

Pour la création et la gestion des sockets, nous avons commencé par utiliser la fonction `socket()` pour ouvrir un point de communication réseau. On a opté pour le protocole TCP (`SOCK_STREAM`) car il garantit un échange fiable des données ( acquittements). Une fois la socket créée, on a activé l'option `SO_REUSEADDR` avec `setsockopt()`. Nous avons fait ce choix afin de pouvoir réutiliser rapidement un port même s'il est encore en état TIME_WAIT après une fermeture récente et donc pour éviter, lors des tests d'être bloqué par des ports occupés.

Ensuite, on a utilisé `bind()` pour associer la socket à une adresse et un port. On a laissé le système choisir automatiquement un port libre en passant 0, pour éviter des conflits. Après cela, la fonction `listen()` a été mise en place pour activer l'écoute des connexions entrantes.

Pour accepter ces connexions, on a utilisé la fonction `accept()` qui crée une nouvelle socket dédiée à chaque client. Cette séparation entre la socket d'écoute et les sockets des clients facilite la gestion des connexions multiples. En plus, avec `getsockname()`, nous récupérons et affichons le port assigné dynamiquement, pour tester et débugger les connexions en cas de problème.

## 2.4 Communication et envoi des données aux processus distants

Pour gérer la communication avec les processus distants, on a défini un protocole d'échange clair et structuré. L'idée, c'était d'envoyer trois types d'informations essentielles : d'abord, le nombre total de processus pour que chacun ait une vue globale du système ; ensuite, le rang individuel de chaque processus pour leur assigner un identifiant unique ; et enfin, les détails de connexion (nom de la machine et numéro de port) pour établir les communications réseau. Ce découpage en étapes distinctes simplifie non seulement l'envoi des données, mais aussi leur réception côté machine distante.

On a choisi d'utiliser les fonctions `htonl()` et `ntohl()` pour convertir les entiers au format réseau. Ce choix technique est important pour éviter les problèmes d'incompatibilité entre différentes architectures (big-endian et little-endian).

Pour l'envoi des données, on a préféré une approche plus fiable que l'utilisation d'un simple `write()`, en s'inspirant des bonnes pratiques vues dans le cours d'IF210. On a implémenté une boucle qui vérifie et garantit que toutes les données sont bien transmises, même si l'envoi doit être fragmenté par le système. Ce choix permet d'éviter les pertes ou les transmissions incomplètes, surtout lorsque la taille des données dépasse celle des buffers réseau. En cas d'erreur, le code gère proprement la situation en affichant un message et en libérant la mémoire avant de quitter.

Pour la partie connexion, on a optimisé l'envoi des informations avec un buffer unique, pour réduire le nombre d'appels système et améliorer les performances. Et bien sûr, toutes les transmissions sont vérifiées avec des printf pour détecter d'éventuelles erreurs et agir en conséquence.

## 2.5 Les Tubes : Gestion des communications parent-enfants

Pour gérer la communication entre le processus parent et ses enfants, nous avons opté pour l'utilisation de tubes (`pipe()`) pour rediriger les sorties des processus. Chaque processus enfant dispose de deux tubes distincts : un pour capturer la sortie standard (`stdout`) et un autre pour la sortie d'erreur (`stderr`). Lors de la création des tubes, nous avons généré deux descripteurs de fichiers par tube, une extrémité pour l'écriture et l'autre pour la lecture. Le parent garde les extrémités en lecture ouvertes pour surveiller les données, tandis que les extrémités en écriture sont fermées, car elles sont inutiles de son côté. À l'inverse, chaque enfant ferme les extrémités en lecture et conserve uniquement celles en écriture pour rediriger ses sorties vers les tubes.

Dans le processus enfant, nous avons utilisé la fonction `dup2()` pour remplacer les descripteurs par défaut de `stdout` et `stderr` avec les extrémités en écriture des tubes. À cet effet, tout ce que l'enfant imprime ou logue sera automatiquement redirigé dans les tubes. Une fois cette redirection implémentée, les extrémités en écriture des tubes sont fermées dans l'enfant pour éviter des fuites de descripteurs. Ce choix d'architecture nous permet d'avoir une isolation complète des sorties des enfants et facilite la collecte des logs par le parent, même en cas d'exécution concurrente de plusieurs processus.

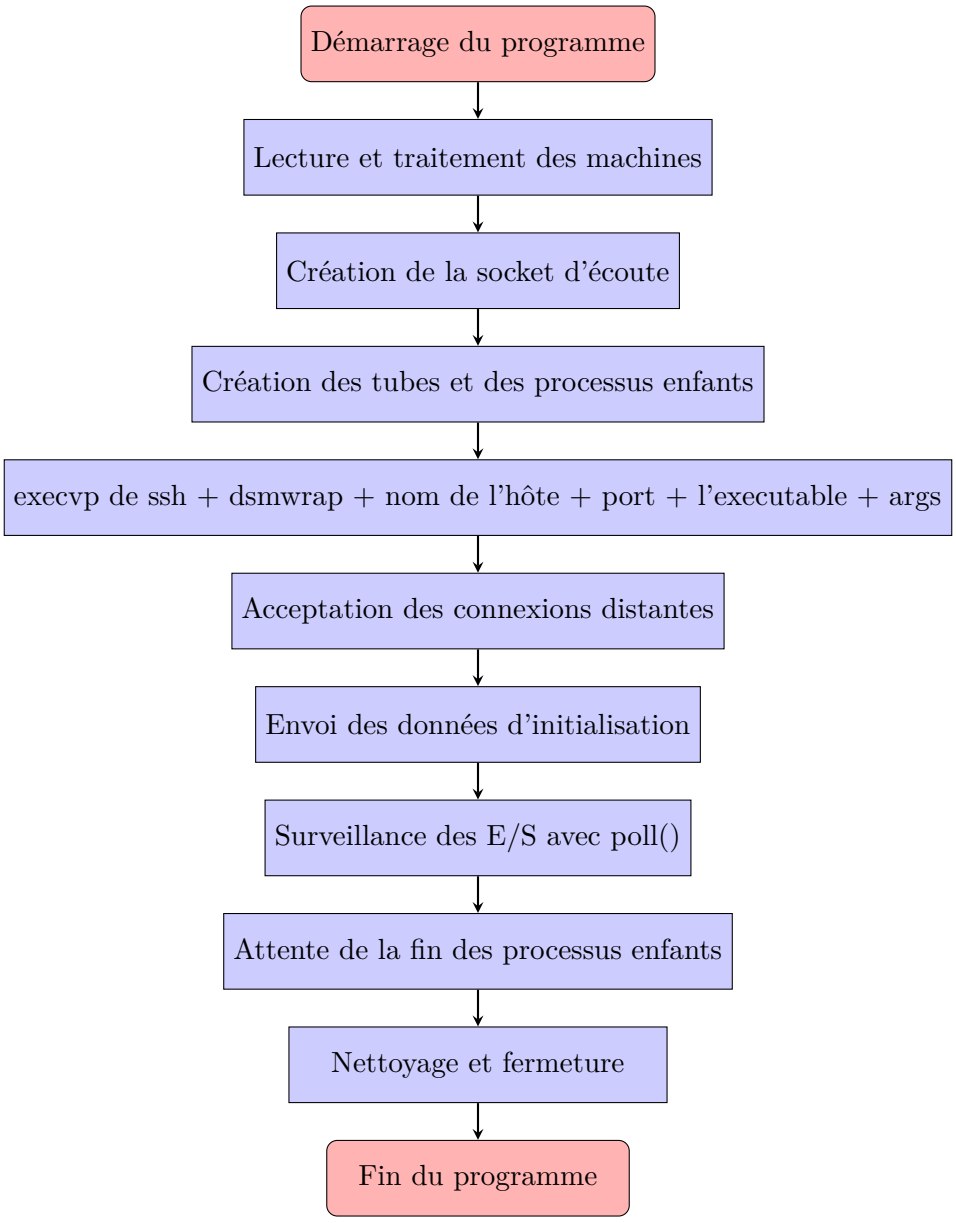
## 2.6 Gestion des E/S avec poll() : Surveillance simultanée des flux

Pour traiter les données en provenance des tubes, nous avons encapsulé la gestion des entrées/sorties dans une fonction dédiée : `handle_io()`. Cette fonction repose sur `poll()`, un mécanisme permettant de surveiller simultanément plusieurs descripteurs. Chaque tube associé

à `stdout` et `stderr` est configuré pour détecter les événements de lecture (`POLLIN`). Dès qu'un événement est signalé, la fonction lit les données disponibles et les affiche immédiatement, en précisant leur origine (stdout ou stderr) ainsi que le processus correspondant.

Pour garantir une gestion propre des ressources, `handle_io()` détecte automatiquement les fermetures de tubes lorsqu'un enfant termine son exécution. Les descripteurs correspondants sont alors fermés et marqués comme inactifs. Cette approche optimise l'utilisation des ressources et évite tout blocage en cas d'inactivité des processus. En combinant `poll()` avec des lectures non bloquantes, nous avons conçu un système capable de traiter efficacement les sorties de plusieurs processus simultanés, tout en assurant une gestion dynamique et réactive des flux d'informations.

# Diagramme des étapes principales de dsmexec



# 3 Phase 1 : dsmwrap

## 3.1 Traitement des arguments et vérifications initiales

Pour commencer, nous avons mis en place une vérification stricte des arguments passés au programme (`argc` et `argv`). Cette étape est importante car elle nous permet de vérifier que le nombre d'arguments est suffisant pour exécuter le processus correctement. En cas d'arguments manquants, un message d'erreur est affiché avec `fprintf()`, et le programme quitte immédiatement. Cela permet d'éviter des comportements imprévisibles plus tard dans l'exécution. Ensuite, nous avons extrait les informations essentielles comme le port du lanceur, son nom d'hôte et les arguments destinés à l'exécutable final (avec une boucle for allant de argv[2] jusqu'à argv[argc]). Ces informations sont stockées pour être réutilisées dans les connexions et lors de l'exécution du programme.

## 3.2 Connexion au lanceur (dsmexec)

La connexion au lanceur a été réalisée en utilisant une socket TCP créée avec `socket()`. Nous avons utilisé `gethostbyname()` pour traduire le nom d'hôte en adresse IP, afin de rendre le code adaptable à différentes configurations réseau. Ensuite, avec `connect()`, nous avons établi la connexion au lanceur en utilisant le port qui a été fourni en argument (avec le execvp de ssh). Une fois connecté, `dsmwrap` envoie les informations d'identification essentielles comme son nom d'hôte et son PID à l'aide de `write()`. Ces informations sont nécessaires pour que `dsmexec` puisse gérer correctement les connexions et les processus distants. En cas d'échec à n'importe quelle étape, des messages d'erreur sont affichés, et les sockets sont fermées pour éviter des fuites de ressources.

## 3.3 Création et gestion de la socket d'écoute

Pour permettre la communication avec d'autres processus DSM, nous avons configuré une socket d'écoute locale. Cette socket est créée avec `socket()` en utilisant le protocole TCP pour assurer la fiabilité des échanges. Ensuite, nous avons lié la socket à un port dynamique choisi automatiquement par le système avec `bind()` en spécifiant le port 0 afin d'éviter les conflits avec des ports déjà utilisés. Une fois la liaison réussie, nous avons récupéré le port attribué avec `getsockname()` et l'avons envoyé au lanceur via `write()`. À cet effet dsmexec pourra informer les autres processus sur la manière d'établir une connexion. Finalement, nous avons activé l'écoute des connexions entrantes avec `listen()` pour rendre le processus prêt à accepter des connexions tout en assurant la gestion des erreurs avec des fermetures propres des sockets en cas d'échec.

## 3.4 Exécution de l'exécutable final

Pour lancer l'exécutable final, nous avons utilisé `execvp()`, pour remplacer le code du processus courant par celui de l'exécutable spécifié. Les arguments pour ce dernier sont récupérés directement à partir des paramètres initiaux du programme. Si l'exécution échoue, un message d'erreur est affiché avec `perror()`, et le programme ferme proprement les sockets avant de quitter.

### 3.5 Gestion des erreurs

Une attention particulière a été portée à la gestion des erreurs et au nettoyage des ressources. À chaque étape critique, des vérifications sont effectuées, et en cas d'échec, un message d'erreur est affiché avant de libérer les ressources utilisées, comme les sockets. Les appels à `close()` assurent une libération immédiate des descripteurs ouverts pour éviter les fuites. Même en cas de succès, toutes les ressources sont explicitement fermées à la fin pour garantir un fonctionnement propre et éviter tout comportement imprévisible. Bien que cette approche ait grandement détecté des erreurs, il faut admettre que le débogage des erreurs s'est avéré difficile et fastidieux.

## 4 Phase 2 : DSM

### 4.1 Gestion des communications

Pour permettre aux processus DSM de communiquer efficacement, on a mis en place un système basé sur des sockets TCP. Chaque processus démarre par la création d'une socket d'écoute grâce à `socket()`, suivi d'une liaison dynamique à un port libre avec `bind()`. Ensuite, il passe en mode écoute avec `listen()` pour accepter les connexions entrantes. Ainsi on pourrait assurer une communication fiable et bidirectionnelle, adaptée aux échanges de données entre les processus. Les connexions sont ensuite établies en utilisant soit `accept()` pour les processus de rang inférieur, soit `connect()` pour ceux de rang supérieur. Nous avons opté pour cette organisation hiérarchique car elle simplifie la synchronisation et évite les conflits.

Pour gérer plusieurs connexions simultanées, on a utilisé un thread dédié avec `select()` dans la fonction `dsm_comm_daemon()`. Ça nous a peris de surveiller en parallèle les descripteurs de fichiers pour détecter les événements réseau. Dès qu'un message arrive, il est traité immédiatement en fonction de son type (demande de page, notification, ou finalisation). On a également intégré une gestion des erreurs pour fermer proprement les sockets en cas de problème, évitant ainsi des fuites de ressources.

### 4.2 Protocole d'échange des pages

Pour organiser l'échange des pages mémoire, on a défini un protocole structuré basé sur des messages réseau. Lorsqu'un processus accède à une page qu'il ne possède pas, un signal `SIGSEGV` est déclenché. Le gestionnaire de signaux identifie la page et envoie une demande (`DSM_REQ`) au processus propriétaire. Ce dernier répond avec un message contenant la page demandée (`DSM_PAGE`) et met à jour son état pour indiquer qu'il n'en est plus responsable. À cet effet, les pages sont transférées dynamiquement et uniquement lorsque c'est nécessaire, et donc on optimisera l'utilisation des ressources mémoire.

En parallèle, une notification (`DSM_NREQ`) est envoyée aux autres processus pour les informer du changement de propriétaire. Cela permet de maintenir une vue cohérente des statuts des pages dans tout le système. Les transferts sont réalisés avec `dsm_send()` et `dsm_recv()`, qui découpent les données si nécessaire et vérifient leur intégrité à chaque étape.

### 4.3 Gestion des erreurs de segmentation (SIGSEGV)

Pour gérer les erreurs d'accès à la mémoire partagée, nous avons opter pour un traitant de signal. Dès qu'une erreur survient, la fonction `segv_handler()` identifie l'adresse fautive (celle qui a causé l'erreur) et vérifie si elle appartient à la zone mémoire partagée. Si c'est le cas, elle appelle la fonction `dsm_handler()` pour demander la page au propriétaire actuel. Ainsi nous pourrons intercepter les erreurs de ségmentation et les transformer en requêtes réseau plutôt que d'arrêter le programme.

Dans `dsm_handler()`, on déclenche une demande au propriétaire et on attend la page demandée. Une fois la page reçue, elle est copiée en mémoire et marquée comme écrivable localement. Les protections de la page sont ensuite ajustées pour autoriser l'accès. Cela repose sur des contrôles stricts pour éviter des situations d'incohérence, comme plusieurs propriétaires pour une même page. On a également prévu des notifications aux autres processus pour les informer des mises à jour.

## 4.4 Allocation et Protection des Pages Mémoire

Pour gérer les pages mémoire, nous avons utilisé des fonctions dédiées comme `mmap()` pour allouer de nouvelles pages et `mprotect()` pour contrôler leurs permissions. Dès l'initialisation, chaque processus se voit attribuer un sous-ensemble des pages en tourniquet afin qu'aucun processus ne contrôle toutes les pages. Chaque page est initialement allouée en lecture-écriture pour permettre son remplissage, puis protégée en lecture seule pour déclencher des erreurs de ségmentation potentielles lors des écritures.

Quand un processus reçoit une page d'un autre, il met à jour ses informations locales avec `dsm_change_info()` pour savoir le nouvel état et le propriétaire. Ensuite, on utilise `mprotect()` pour donner les bonnes permissions et éviter des modifications non autorisées. Pour simplifier la gestion, toutes les pages sont libérées avec `munmap()` lors de la finalisation.

## 4.5 Nettoyage et Finalisation

Pour éviter les fuites de ressources, on a implémenté la fonction `dsm_finalize()` qui s'occupe de fermer toutes les connexions réseau et de libérer les pages mémoire. Avant de se terminer, chaque processus envoie un message de finalisation (`DSM_FINALIZE`) aux autres pour qu'ils puissent également se nettoyer. Cela assure que tous les processus terminent correctement, même en cas d'arrêt inattendu.

Le nettoyage inclut aussi la fermeture des descripteurs de fichiers et la suppression des pages allouées. On arrête proprement le thread de communication avec `pthread_cancel()` et `pthread_join()`, ainsi toutes les requêtes en cours sont traitées avant l'arrêt. Ce choix nous a évité de nombreux bugs difficiles à trouver lors des tests.

## 4.6 Synchronisation

La synchronisation est importante dans un système comme le nôtre où plusieurs processus accèdent en même temps à des ressources partagées. Utiliser des mutex (`pthread_mutex_t`) permettrait de protéger les sections critiques, comme la table des pages mémoire, les infos sur les propriétaires et les statuts des pages. En verrouillant et déverrouillant autour des accès sensibles, on éviterait les accès en parallèle qui peuvent causer des soucis de désynchronisation. Ajouter un mutex dédié pour gérer la fin du programme aiderait aussi à fermer tout proprement, même si plusieurs requêtes arrivent en même temps. Donc, avoir des verrous bien séparés pour chaque partie critique réduirait les blocages inutiles et garderait le code clair et facile à gérer.

## 4.7 Problèmes de Connexion et Limites des Tests

Malheureusement, nous n'avons pas réussi à faire fonctionner correctement la partie connexion entre les processus DSM. Malgré plusieurs tentatives de débogage et des tests sur les échanges de messages et les sockets, l'erreur reste difficile à localiser. Les connexions échouent sans retour d'information clair, ce qui complique énormément l'identification du problème. Ce blocage nous empêche d'avancer sur la suite du projet et de tester si les autres parties que nous avons implémentées fonctionnent correctement. Comme tout repose sur l'établissement des connexions

entre les processus, on n'a pas pu vérifier le transfert des pages, la gestion des accès mémoire ou encore la communication entre les threads.

# 5 Conclusion

Ce projet nous a permis d'explorer en détail la gestion des processus, des communications réseau et des accès mémoire dans un environnement distribué. Nous avons réussi à mettre en place plusieurs mécanismes importants, notamment la gestion des processus enfants, la redirection des entrées/sorties avec des tubes et la gestion des pages mémoire avec protection. Nous avons également implémenté un protocole d'échange structuré pour transférer les pages entre processus.

Cependant, la partie connexion entre les processus DSM n'a pas pu être finalisée. Ce problème nous a empêchés de tester pleinement le reste des fonctionnalités et de valider le système dans des conditions réelles. Pour résoudre ce blocage, un débogage plus approfondi et l'ajout de la synchronisation seraient nécessaires.