École Nationale Supérieure
d'Électronique, Informatique,
Télécommunications, Mathématique et
Mécanique de Bordeaux

Filière Télécommunications, Semestre 7
PR204 : Projet Système et Réseau
Rapport

Implémentation d'une Mémoire Partagée Distribuée (DSM)

Élèves :
Arnaud SERRES
Huy Son NGUYEN

Encadrants :
Guillaume MERCIER
Joachim
BRUNEAU-QUEYREIX
Philippe SWARTVAGHER

Table des matières

Introduction

L'objectif de ce projet est d'implémenter un système de distribution de mémoire partagée (DSM), permettant la communication entre plusieurs machines physiques distinctes. Ces machines peuvent ainsi collaborer sur une même tâche avec des processus indépendants. Par exemple, dans le cadre d'un calcul matriciel, la DSM permet de répartir les calculs entre différentes machines pour optimiser les performances.

Le projet est divisé en deux phases principales :

— La Phase 1 vise à lancer les processus distants et à établir la communication entre eux.

— La Phase 2 se concentre sur la mise en place de la mémoire partagée en définissant des espaces d'adressage appropriés.

1 Phase 1

Pour la Phase 1, une base de code a été fournie avec des indications permettant de réaliser les objectifs attendus. Cette phase a principalement consisté à gérer le lancement des processus sur des machines distantes et à établir les connexions nécessaires pour la communication entre eux.

1.1 `dsmexec.c`

Le fichier `dsmexec.c` constitue le lanceur principal des processus DSM. La première étape a été de lire le fichier `machine_file` contenant les noms des machines cibles. Pour cela, nous avons implémenté la fonction `lire_machinefile`, qui ignore les lignes vides et stocke les noms des machines dans une structure dynamique.

Ensuite, `dsmexec.c` crée une socket d'écoute en utilisant la fonction `create_listen_socket`. Cette socket permet d'accepter les connexions entrantes des processus distants. Le choix de laisser le système sélectionner le port disponible (en définissant `PORT 0`) offre une flexibilité et évite les conflits de ports.

Le lanceur initialise également des pipes pour rediriger les flux `stdout` et `stderr` des processus enfants, permettant ainsi de capturer les sorties des processus distants pour une surveillance centralisée.

La boucle de création des processus utilise `fork` et `execvp` pour lancer `dsmwrap` sur chaque machine distante via SSH. Cette approche garantit que chaque processus DSM s'exécute dans un environnement isolé tout en maintenant une communication efficace avec le lanceur.

Enfin, `dsmexec.c` gère l'acceptation des connexions entrantes des processus DSM distants, récupère les informations d'identification telles que le nom de la machine et le PID, et met en place le protocole d'échange de données nécessaire pour la suite du projet.

Nous reviendrons sur l'utilisation des fonctions `lire_machinefile`, `create_listen_socket` et `nettoyage` dans la partie `common.c` et `common_impl.h` de notre rapport.

Dans notre implémentation pour la phase 1, nous avons utilisé le fichier `machine_file` contenant les noms des machines auxquelles se connecter et non pas les adresses IP des

machines pour la connexion. Il s'agit d'un choix pratique car stocker le nom est plus parlant et facile à vérifier que de conserver les adresses IP.

Les processus distants se connectent à la socket d'écoute. Le programme `dsmexec.c` accepte ces connexions avec accept et lit les données envoyées (nom d'hôte, PID, etc...). Cette approche permet de centraliser les informations des processus distants au niveau de notre lanceur. Cela permet également d'implémenter des mécanismes de validation ou d'échange de données au moment de la connexion. Nous aurions pu penser à une approche décentralisée où chaque processus distant communique avec tous les autres afin de réduire la charge sur `dsmexec`, mais cela aurait complexifié le protocole de communication.

Afin de pouvoir lancer le fichier `dsmwrap.c` il était nécessaire de définir un chemin de manière dynamique. En effet, l'interopérabilité des lanceurs était un aspect important permettant d'éviter des problèmes de compatibilité lors de l'exécution pour lequel le sujet imposait une approche.

Le sujet nous rappelait également que nous avions le choix d'utiliser les fonctions `execvp`, `execvpe` et `execlp`. Nous avons décidé de rester sur le premier choix, car `execvpe` est particulièrement adapté dans le cas où il faudrait spécifier un environnement avec `envp` et `execlp` nécessite que les arguments soient passés individuellement plutôt que sous forme de tableau. Dans notre cas, `newargv` représentait un tableau d'arguments et les variables d'environnement étaient suffisantes, d'où notre choix de passer par `execvp`.

Dans notre fichier `dsmexec.c`, pour les redirections de flux et les acceptations de connexion, nous avons réalisé une implémentation séquentielle (une connexion après l'autre, une lecture après l'autre). Cette gestion des descripteurs à la suite était une approche classique, plus simple et suffisante car les descripteurs ne changent pas. Avec du recul, il aurait été intéressant d'utiliser la fonction `poll` permettant de gérer un nombre plus important de descripteurs et réduire le risque d'erreur/blocage pour un descripteur de fichier.

1.2 `dsmwrap.c`

Le fichier `dsmwrap.c` agit comme un intermédiaire entre le lanceur `dsmexec.c` et le processus de mémoire partagée. Son rôle principal est de faciliter la connexion au serveur DSM en établissant une communication initiale et en envoyant les informations d'identification nécessaires.

Lors de son exécution, `dsmwrap.c` récupère le nom de la machine locale et résout l'adresse IP du serveur DSM en utilisant les arguments fournis. Il crée ensuite une socket et tente de se connecter au lanceur `dsmexec.c` en utilisant les informations de port et d'adresse IP.

Une fois la connexion établie, `dsmwrap.c` envoie le nom de la machine et éventuellement le PID au lanceur, ce qui permet à `dsmexec.c` de gérer correctement les connexions et d'associer chaque processus à son identifiant unique.

Pour être certains que `dsmwrap.c` soit correctement appelé, nous avons mis en place une vérification du nombre d'arguments. Il était nécessaire de fournir le port ainsi que l'adresse du server `dsmexec`. Dans le cas où une information était manquante ou qu'un autre argument s'était glissé lors de l'exécution du programme, ce test y mettait alors

un terme. Avec du recul il aurait été préférable d'utiliser les fonctions `inet_pton` ou `inet_aton` car ces dernières assurent une meilleure vérification des arguments (format des adresses IP).

Nous avons également décidé d'utiliser la fonction `gethostname` pour récupérer le nom de la machine locale au lieu d'utiliser directement l'adresse IP locale. Cette deuxième approche pourrait éviter des erreurs liées à des configurations DNS incorrectes et serait moins facile à implémenter alors que la première approche est plus simple et permet une meilleure lisibilité dans les logs.

Pour résoudre l'adresse du server dsmexec en adresse IP, nous avons choisi la fonction `gethostbyname`. Cela nous permet d'adresser dynamiquement le serveur en fonction des configurations réseau. Nous aurions pu utiliser getaddrinfo qui renvoie une liste chaînée de structures et qui est également compatible pour IPv6.

Dans notre implémentation de `dsmwrap`, nous avons fait en sorte que ce dernier envoie son nom de machine ainsi que d'autres informations telles que le PID à `dsmexec`. Nous avons fait ce choix afin d'obtenir une cartographie complète des processus distants, pouvant faciliter les communications futures.

1.3 `common.c` et `common_impl.h`

Les fichiers `common.c` et `common_impl.h` contiennent des fonctions utilitaires partagées entre `dsmexec` et `dsmwrap`. Ces fonctions facilitent la gestion des connexions, la lecture des fichiers de configuration, et la manipulation des structures de données communes.

Parmi les fonctions clés implémentées dans `common.c` :

— **ligne_vide** :
Vérifie si une ligne du fichier de machines est vide ou contient uniquement des espaces, permettant ainsi d'ignorer les entrées inutiles en réalisant une boucle en vérifiant avec `isspace`.

— **lire_machinefile** :
Lit un fichier contenant des noms de machines. Tout d'abord, le fichier est ouvert en mode lecture à l'aide de la fonction `fopen`, ce qui permet d'accéder à son contenu. Ensuite, les lignes sont lues une par une grâce à la fonction `fgets`, qui permet de traiter chaque ligne individuellement. Pour nettoyer les chaînes de caractères, la fonction `strcspn` est utilisée afin de supprimer les retours à la ligne inutiles. Une vérification supplémentaire est effectuée avec la fonction `ligne_vide`, qui ignore les lignes ne contenant que des espaces ou étant vides.

— **create_listen_socket** : Crée une socket d'écoute sur une machine donnée, résout le nom d'hôte en adresse IP, et associe la socket à un port disponible choisi dynamiquement. Elle commence par résoudre le nom de la machine (`hostname`) en une adresse IP grâce à la fonction `gethostbyname`. Si la résolution échoue ou si aucune adresse IP n'est disponible, la fonction termine avec un message d'erreur. Une fois l'adresse IP récupérée, une socket est créée avec socket, spécifiant une communication basée sur le protocole TCP/IP.

La structure `server_addr` est ensuite initialisée pour stocker les informations de connexion, incluant le type d'adresse (`AF_INET`), un port choisi dynamiquement (0), et l'adresse IP obtenue. Cette structure est associée à la socket via `bind`, ce qui permet de lier la socket à l'adresse et au port. Si aucune erreur ne survient, la fonction `getsockname` est utilisée pour récupérer le port attribué dynamiquement par le système, qui est stocké dans la variable fournie en paramètre (`*port`).

Enfin, la socket est configurée en mode écoute avec `listen`, autorisant un nombre maximal de connexions simultanées défini par `nbr_proc`. La fonction retourne le descripteur de la socket créée et affiche les informations de l'adresse et du port pour confirmation.

— **nettoyage** : Libère la mémoire allouée pour les noms des machines, garantissant ainsi une gestion efficace des ressources à l'aide de la fonction `free` utilisée sur chaque élément du tableau (`machine_file`) puis sur la mémoire du tableau.

Notre choix de lire les lignes du fichier les unes à la suite (où chaque ligne est un nom de machine). Cette approche était la plus simple mais était suffisante. Il n'était pas nécessaire de passer par une table de hachage pour s'occuper d'éventuelles duplications.

De manière similaire à notre `dsmwrap.c`, nous avons utilisé `gethostbyname` dans notre fonction `create_listen_socket` pour résoudre le nom de la machine en adresse IP. Nous aurions pu utiliser également `getaddrinfo`.

Le sujet suggérait une approche dynamique concernant la gestion des ports, là où un port statique préconfiguré aurait encouru le risque de conflits si plusieurs instances du programme tournaient en parallèle. Nous avons ainsi passé 0 dans le champ `sin_port` de `sockaddr_in`, permettant au système de choisir automatiquement un port libre.

De même avec `getsockname` pour la récupération du port dynamique.

Pour chaque étape (`socket`, `bind`, `getsockname`, `listen`) nous avons fait une vérification avec gestion stricte des erreurs de manière similaire à notre implémentation du `dsmwrap`. Nous aurions pu utiliser la fonction `poll` mais cela n'était pas nécessaire étant donné que les descripteurs de fichiers ne changeaient pas.

Ces fonctions ont initialement été créées pour éviter des répétitions dans `dsmexec.c` et `dsmwrap.c`. Mais ne les ayant utilisées qu'un faible nombre de fois, avec du recul, il s'agissait d'un moyen pour nous de factoriser notre code et permettre une meilleure lisibilité de notre travail.

Avec plus de temps, nous aurions aimé pouvoir factoriser encore plus notre code présent dans `dsmexec.c` et `dsmwrap.c`.

2 Phase 2

La Phase 2 reprend le code de la Phase 1 et met en place le système de distribution de mémoire partagée. Nous avons décidé de récupérer le code fonctionnel pour la Phase 1 de nos professeurs afin d'éviter d'éventuels problèmes, notre travail n'étant pas complet.

2.1 `dsm_impl.h` et `dsm.h`

Le fichier `dsm_impl.h` définit les structures de données et les constantes essentielles pour la gestion de la mémoire partagée. Il introduit des concepts tels que les pages de mémoire, les droits d'accès, et l'état des pages. L'utilisation d'énumérations pour les types d'accès (`dsm_page_access_t`) et les états des pages (`dsm_page_state_t`) permet de gérer les droits et états de chaque page de mémoire partagée.

De plus, la définition de structures comme `dsm_proc_conn_t` permet de centraliser les informations de connexion des processus DSM, facilitant ainsi la gestion des interactions réseau et des accès mémoire.

Les fonctions essentielles `dsm_init` et `dsm_finalize`, permettant aux applications d'initialiser le système DSM et de le terminer proprement une fois les opérations terminées.

Le programme `dsm.h` fournit des fonctions publiques telles que `dsm_init` et `dsm_finalize`. Il sert d'interface utilisateur pour les programmes externes utilisant le DSM.

2.2 `dsm.c`

Le fichier `dsm.c` constitue le cœur de la gestion de la mémoire partagée. Les fonctionnalités clés sont :

- **Gestion des Pages de Mémoire** : Les fonctions telles que `dsm_alloc_page`, `dsm_protect_page`, et `dsm_free_page` permettent d'allouer, de modifier les protections, et de libérer des pages de mémoire partagée. L'utilisation de `mmap` pour l'allocation de mémoire garantit une gestion efficace et flexible des blocs mémoire.

- **Traduction d'Adresses** : Les fonctions `num2address`, `address2num`, et `address2pgaddr` facilitent la conversion entre les numéros de pages et les adresses mémoire.

- **Gestion des Accès et des Propriétaires** : La structure `dsm_page_info_t` et les fonctions associées permettent de suivre l'état et le propriétaire de chaque page, garantissant les accès et la prévention des conflits entre les processus.

- **Gestion des Signaux** : (`segv_handler`) permet de détecter et de gérer les accès invalides à la mémoire partagée. En interceptant les erreurs de segmentation, le système peut s'adapter, que ce soit en corrigeant l'accès ou en terminant le processus de manière sécurisée.

- **Communication Inter-Processus** : Le thread de communication (`dsm_comm_daemon`) est responsable de l'écoute et du traitement des requêtes provenant des autres processus DSM.

- **Initialisation et Finalisation** : Les fonctions `dsm_init` et `dsm_finalize` s'occupent du démarrage et de l'arrêt du système DSM. `dsm_init` configure les connexions, initialise les structures de données, et met en place les mécanismes de gestion des pages. `dsm_finalize` assure une fermeture des connexions et la libération des ressources allouées.

Dans notre implémentation du `dsm.c`, au niveau de la réception des informations des processus distants, rien ne garantissait que les fonctions `read` et `write` traitaient bien toute l'information. C'est pour cela que nous avons ajouté une boucle avec un `while`, assurant que l'information soit complètement reçue ou envoyée.

Au cours de cette phase, nous avons rencontré un problème concernant la connexion aux processus distants. En effet, lorsqu'on compilait et exécutait notre code, il y avait de façon irrégulière des refus de connexion. Après réflexion, nous avons alors rajouté une boucle avec `while` au niveau des appels à `connect` pour les sockets distantes. Ainsi, nous avons protégé la connexion contre les délais pouvant la nuire.

Un autre enjeu de cette phase était la connexion inter-processus distants. À partir du processus en cours d'exécution, nous avons décidé de nous connecter à tous les processus de rang supérieur et d'accepter la connexion de tous les processus de rang inférieur. De cette manière, nous avons évité les redondances tout en assurant la connexion de tous les processus par le biais d'une seule socket.

Pour l'implémentation de notre traitant de signal, nous avons complété la fonction `dsm_handler`. Cette dernière identifie le propriétaire actuel de la page que l'on tente d'accéder et lui envoie une requête avec le numéro de page et l'identifiant du processus demandeur. Nous récupérons ensuite la page en question et nous prévenons les autres processus distants que nous sommes le nouveau propriétaire de la page, afin que chacun mette à jour son tableau `table_page`, qui référence les propriétaires de chaque page. La fonction `dsm_handler` est essentielle pour gérer les fautes de page dans le système DSM.

Ensuite pour répondre à une demande de page, nous avons utilisé un thread à l'aide de `dsm_comm_daemon` qui attend et traite les requêtes des autres processus. La fonction `poll` permet de surveiller l'activité sur la socket reliant les processus distants et le type de requête étant connu, un `switch` permet de traiter la demande en fonction de sa nature. De plus, si la vérification du propriétaire avant d'envoyer la page échoue, nous avons choisi de transférer la demande au bon propriétaire. Cela répond aux attentes de surveillance des connexions réseau, de gestion flexible des requêtes et du maintien de la cohérence de la DSM.

Pour éviter les conflits entre les processus lors de l'envoi de requêtes ou de pages, nous utilisons des `mutex` pour les synchroniser. Le verrou est pris par le processus qui envoie la requête et est libéré par le thread une fois la requête traitée. Cette approche garantit qu'un seul processus ou thread peut accéder à une ressource critique à un instant donné. En effet, sans cela, plusieurs processus pourraient essayer de modifier simultanément les mêmes données ce qui entraîne des incohérences ou des erreurs.

3 Répartition du travail

Lors de ce projet, nous avons décidé de travailler majoritairement ensemble sur les mêmes tâches afin de mieux comprendre ce que nous faisions et d'éviter des problèmes de compatibilité entre deux fichiers. Néanmoins, nous avons tout de même assigné des parties à chacun d'entre nous.

— **Arnaud Serres** s'est principalement concentré sur le développement de `dsmexec.c`, `common.c`, et l'implémentation des mécanismes de gestion des processus et des connexions réseau dans la Phase 1. Dans la Phase 2, il a travaillé sur `dsm.c` et sur `dsm_impl.h`, en se focalisant sur la gestion de la mémoire partagée et la mise en place des handlers de signaux.

— **Huy Son Nguyen** a pris en charge `dsmwrap.c` et l'implémentation des fonctions utilitaires dans `common_impl.h` durant la Phase 1. Pour la Phase 2, il a travaillé sur `dsm.c`, concernant la récupération des ports, des noms de machine ainsi que du traitant de signal.

L'un des problèmes que nous avons rencontré était que **Huy Son Nguyen** voulait, au cours du projet, implémenter des fonctions dans le `common_impl.h` alors qu'**Arnaud Serres** avait déjà commencé à implémenter ces fonctions directement dans le `dsmexec.c`. Après réflexion, nous avons décidé de supprimer celles du `common_impl.h`, ce qui nous a fait perdre du temps.

Au final, ne pas trop diviser notre travail et questionner les choix de l'autre membre du binôme nous a permis d'assimiler plus facilement le projet et d'avancer plus rapidement.

Conclusion

Pour conclure, ce projet a été une bonne opportunité pour consolider nos connaissances en programmation système et en programmation réseau. Nous avons également amélioré notre compréhension du langage de programmation C. Il s'agit d'un travail synthétique qui nous a permis de développer une rigueur de travail ainsi que de nous rendre compte des possibilités offertes par l'ensemble des cours que nous suivons.