Réalisation d'une mémoire partagée répartie

Encadrant : **Pierre Sens,**

Étudiants : **Nicolas Guittonneau, Florian Reynier, Yoann Ghigoff**

Table des matières

1 Présentation du sujet

Ce projet réalise une bibliothèque implémentant une mémoire partagée répartie en langage C. Cette bibliothèque abstrait le fait que la mémoire est partagée entre plusieurs processus distants.

Les processus utilisant la bibliothèque peuvent accéder à la mémoire partagée en lecture et en écriture de manière concurrente. La bibliothèque implémente donc des fonctions de synchronisation assurant la cohérence de la mémoire partagée face aux accès concurrents des différents processus.

La mémoire partagée est découpée en pages comme une mémoire virtuelle ordinaire. Elle est initialement disponible pour un processus. Lorsqu'un processus accède à une donnée de la mémoire partagée via la bibliothèque, c'est-à-dire à une adresse de celle-ci, la page contenant la donnée lui est alors envoyée via le réseau si elle est disponible. Une page peut donc être dupliquée au travers du réseau.

Cependant, en cas d'accès écriture/écriture ou écriture/lecture par différents processus à une même page, un mécanisme de synchronisation est mis en oeuvre afin de garantir la cohérence des données de la page. Ce mécanisme est implémenté par une exclusion mutuelle répartie, opérée par un processus central, appelé **Master**. Les autres processus étant nommés **Slave**.

La communication entre les processus s'effectue à l'aide de sockets connectées et d'un protocole basé sur l'envoi et la réception de messages.

2 Documentation utilisateur

2.1 Structuration des fichiers sources

Voici un bref descriptif des fichiers que vous trouverez dans le dossier "/src" :

- *dsm* : contient toutes les fonctions publiques de la librairie.
- *dsm_core* : les primitives utilisée par les daemon lors de la reception des messages, ainsi qui la distribution des pages.
- *dsm_master* : Les primitives d'initialisation et de destruction du maître.
- *dsm_memory* : Les primitives liées à la gestion de la mémoire partagée.
- *dsm_protocol* : Les primitives de conversion des messages liée à la librairie binn, ainsi que les structures propres à chaque type de message existant.
- *dsm_socket* : Les primitives liée aux sockets (création, initialisation, destruction).
- *dsm_util* : définitions de macros pour une gestion des affichages simplifié.
- *list* : listes chaînées génériques utilisées pour la gestion des pages.
- *binn* : la librairie binn.

2.2 Documentation du code

Les commentaires du code a été faite afin de pouvoir générer la documentation avec l'outil **doxygen**. Celle-ci est disponible dans le fichier annexe `documentation-code.pdf`.

2.3 Documentation de la bibliothèque réalisée

Comme spécifié dans le sujet, notre bibliothèque offre, à l'utilisateur les quatre primitives permettant de garantir la cohérence des données :

```
void *InitMaster(int port, size_t page_count)
```

Primitive permettant d'initialiser le processus maître, prenant respectivement deux arguments ; le port d'écoute du maître et le nombre total de pages de la mémoire partagée. Cette fonction va allouer et initialiser la mémoire partagée ainsi que donner par défaut les droit en lecture et écriture au maître sur toutes les pages. Un thread permettant de recevoir les requêtes est crée et la socket est initialisée. Elle retourne l'adresse de la mémoire partagée répartie.

Cette primitive ne doit être appelée qu'une fois et par un seul processus.

```
void *InitSlave(char *HostMaster, int port)
```

Primitive permettant d'initialiser un processus esclave, prenant respectivement deux arguments ; l'adresse IP du maître et le port utilisé par l'esclave. De la même manière que `InitMaster()`, un thread est créé et la socket initialisée. Un échange de message est effectué avec le maître afin de récupérer diverses informations sur la mémoire, initialisée avec aucun droit sur aucune page. Elle retourne l'adresse de la mémoire partagée répartie.

Cette primitive doit être appelée une fois pour chaque processus esclave.

```
void lock_read(void *adr)
```

Primitive bloquante garantissant l'accès en lecture à la dernière version de la page contenant l'adresse *adr. Si le processus possède déjà la dernière version, la fonction retourne immédiatement.

```
void lock_write(void *adr)
```

Primitive bloquante garantissant l'accès exclusif en écriture à la dernière version de la page contenant l'adresse *adr. Cette primitive est bloquante lorsqu'un autre site possède des droits sur cette page (lecture ou écriture)

En plus de ces quatre primitives demandées dans le sujet de notre projet, nous avons décidé d'ajouter quatre autres primitives à notre bibliothèque afin de la rendre plus simple à utiliser et implémenter.

```
void unlock_read(void *adr)
```

Permet de relâcher l'accès en lecture sur la page contenant l'adresse *adr.

```
void unlock_write(void *adr)
```

Permet de relâcher l'accès en écriture sur la page contenant l'adresse *adr.

`void sync_barrier(int nb_slaves)`

Primitive bloquante simulant une barrière, elle se termine lorsque nb_slaves ont réalisés sont appel.

`void QuitDSM(void)`

Primitive quittant la mémoire partagée distribuée. Si elle est appelée par un processus esclave, le processus envoie toutes les pages dont il possède le droit d'écriture au processus maître avant de détruire ses structures de données et de se déconnecter du système.
Si elle est appelée par le processus maître, la primitive est bloquante jusqu'à ce que l'ensemble des processus esclaves quitte la mémoire partagée distribuée. De même le processus maître détruit ensuite ses structures de données.

Un exemple de bonne utilisation de ces primitives est disponible en annexe A page 15.

3 Documentation technique

3.1 Choix techniques

3.1.1 Choix du langage et norme C99

Ce projet a été réalisé en langage de programmation C pour l'environnement Linux. Ces aspects était imposés mais s'expliquent du fait que le langage C permet de manipuler facilement la mémoire notamment grâce aux notions de pointeur et d'adresse, mais aussi du fait que la Glibc [1] fournit des primitives de gestion mémoire telles que `mmap()` ou `mprotect()`.
Nous avons cependant décidé de développer la bibliothèque en appliquant la norme C99 du langage C tout simplement car elle est plus commode et agréable à utiliser que les normes plus anciennes, notamment en permettant de mélanger déclaration de variables et code.

3.1.2 Compilation de la bibliothèque

Il est possible de compiler la bibliothèque au format shared-object (.so) afin de créer une librairie dynamique.
Elle s'obtient via la commande `make libdynamic`

Il est aussi possible de compiler la bibliothèque pour en faire une librairie statique (.a). Elle s'obtient via la commande `make libstatic`

La commande `make` créé les deux versions de la librairie.

1. GNU C Library

3.1.3 Protocole de transport réseau

Parmi les choix techniques que nous avons eu à faire se trouve celui du type de socket que nous allions utiliser afin de faire communiquer les différents processus entre eux. Notre choix devait se faire entre :

- SOCK_STREAM correspondant à l'implémentation du protocole de transport réseau TCP [2]. Ce type de socket permet une communication fiable et bidirectionnelle entre deux processus sous la forme d'un flux d'octets mais nécessite l'établissement d'une connexion préalablement à l'envoi de données.
- SOCK_DGRAM correspondant à l'implémentation du protocole de transport réseau UDP [3]. Ce type de socket fonctionne sous forme de datagrammes, des messages de taille maximale fixe et en mode non connecté mais ne fournit pas de contrôle d'erreurs, il ne garantit donc pas l'intégrité ni la réception des messages au processus destinataire.

Nous avons donc choisi d'utiliser des sockets de type SOCK_STREAM (et donc le protocole TCP) car celui-ci assure que les données envoyées sont bien reçues par le destinataire telles qu'envoyées par la source. Il permet aussi de maintenir une connexion entre deux processus et d'identifier le processus distant simplement au moyen d'un descripteur de fichier.

3.2 Choix d'implémentation

Lors de la réalisation de ce projet, nous avons été confrontés à plusieurs choix d'implémentation.

Comme mentionné précédemment, notre réalisation repose sur deux types de processus :

- les processus Slaves qui sont simplement des processus réalisant des accès, en lecture et écriture, à la mémoire partagée.
- un processus Master chargé de gérer les informations sur l'état des pages mémoire et la synchronisation entre les différents accès des processus.

3.2.1 Topologie du système

Notre premier choix d'implémentation c'est donc porté sur la manière dont communiqueraient les différents processus entre eux. Nous pouvions décider de faire communiquer chaque processus avec n'importe quel autre ou bien de faire communiquer les processus Slaves uniquement avec le processus Master. Nous avons opté pour la seconde option et ce pour plusieurs raisons.
Tout d'abord, elle permet de centraliser les communications et ainsi les traitements qui

2. Transmission Control Protocol
3. User Datagram Protocol

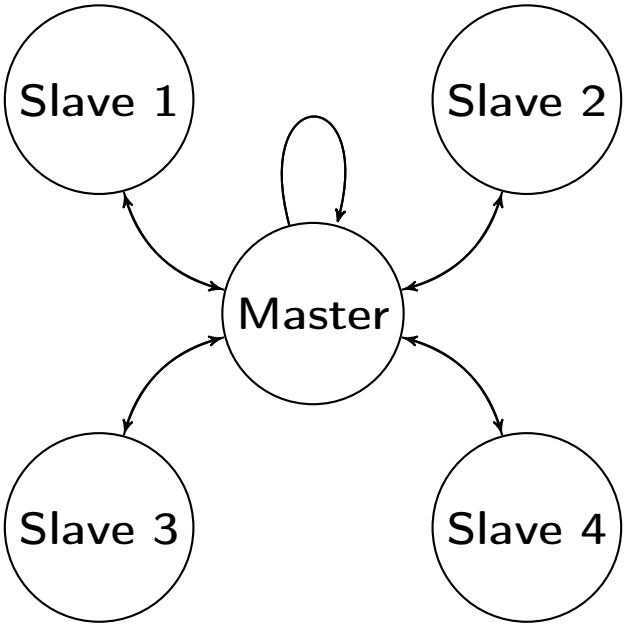


Figure 1 – Topologie en étoile adoptée

leur est associés, ce qui simplifie le développement. Elle permet aussi d'identifier plus simplement les différents processus du système puisque les processus Slaves se connectent au processus Master, créant ainsi une topologie en étoile et permettant au processus Master de connaître l'ensemble des processus partageant la mémoire. De plus l'utilisation d'un serveur centralisé permet à n'importe quel machine de n'importe quel réseau de rejoindre le système à partir du moment où la machine sur laquelle s'exécute le processus Master est accessible depuis Internet.
Cette option possède aussi un désavantage puisque le processus Master centralise tous les échanges de messages. Il représente donc un goulot d'étranglement, cependant le temps de traitement des messages étant relativement faible, nous considérons ce désavantage comme négligeable.

3.2.2 Communication par messages asynchrones

La communication entre les processus se fait exclusivement par messages asynchrones. C'est-à-dire que lorsqu'un processus envoie un message, il n'attend jamais activement sa réponse. Un thread, appelé listener daemon, est chargé d'écouter les messages reçus par la bibliothèque, de les désérialiser, et des les transmettre aux fonctions (handlers) qui vont réaliser les traitements associés à chaque type de message.

Afin de réaliser cette bibliothèque, nous avons eu à décrire un protocole définissant 9 types de message :
— CONNECT pour permettre à un processus Slave de rejoindre le système.
— CONNECT_ACK pour que le processus Slave puisse signaler à un Slave s'il est

accepté ou non dans le système.
— LOCKPAGE pour effectuer une demande de verrouillage, en lecture ou en écriture d'une page, au processus Master.
— INVALIDATE pour signaler à un processus Slave possédant une copie d'une page que sa version n'est plus valide.
— INVALIDATE_ACK pour signaler au processus Master qu'un Slave à abandonner son droit de lecture sur une page.
— GIVEPAGE pour transmettre une page à un processus.
— SYNC_BARRIER pour implémenter une barrière de synchronisation afin de pouvoir contraindre le fil d'exécution d'un ensemble de processus.
— BARRIER_ACK pour débloquer les processus en attente sur une barrière de synchronisation.
— TERMINATE pour quitter le système proprement.

3.2.3 Rôle des processus

Nous avons aussi fait le choix de considérer le processus Master comme un processus Slave, il est cependant différent puisqu'il joue le rôle de serveur et dispose les données nécessaires à la gestion des accès concurrents. Pour se faire, nous avons fait en sorte que le processus Master soit client de lui-même, c'est-à-dire qu'il se connecte à lui même lors de son initialisation. Cela permet de simplifier grandement le code puisque le processus Master n'a pas à différencier les clients de lui-même, les traitements à la réception des messages sont donc les mêmes.
Cela implique cependant une perte de performance pour le processus Master puisqu'il utilise son interface réseau loopback pour gérer sa propre mémoire alors que cela pourrait être fait de manière directe en séparant les traitements du processus Master de celui des processus Slave.

3.2.4 Compatibilités entre machines

Une autre particularité du processus Master est qu'il impose sa configuration de la mémoire partagée distribuée aux processus Slaves.
Lors de l'initialisation de la mémoire partagée pour le processus Master, l'utilisateur de la bibliothèque spécifie le nombre de page qu'il souhaite pour la taille de la mémoire partagée. La taille d'une page mémoire étant celle fournit par le système d'exploitation. L'initialisation de la mémoire partagée pour un processus Slave est différente, le processus Slave commence par se connecter au processus Master en lui envoyant un message de type CONNECT indiquant l'architecture (32 ou 64 bits), ainsi que la taille d'une page utilisées par le système d'exploitation sur lequel il s'exécute. A la réception d'un message CONNECT, le processus Master renvoie un message CONNECT_ACK spécifiant si le processus Master et le processus Slave possèdent la même architecture et la même taille de page, ainsi que le nombre de page de la mémoire partagée spécifié auparavant lors de l'initialisation du processus Master.

Ainsi s'il ne possède pas la même architecture ou la même taille de page que le processus Master, le processus Slave lève une erreur lors de son initialisation. Sinon il utilise le nombre de page de la réponse du processus Master pour initialiser sa mémoire locale.

4 Dépendances

Les dépendances de notre projet sont minimes puisqu'elles sont directement intégrées à celui-ci ou sont souvent déjà résolues car disponibles sur de nombreuses distributions Linux.

4.1 Librairie binn

Lors de la réalisation de ce projet, nous nous sommes posés la question de l'interopérabilité des communications réseaux sachant que l'hétérogénéité de la représentation des données en mémoire est souvent un problème lorsque plusieurs processus s'exécutant sur différentes architectures essayent de communiquer entre eux via des sockets.
Pour pallier à ce problème et après avoir recherché des solutions sur Internet, nous avons décidé d'utiliser la librairie binn[1]. La librairie binn propose un format, aussi appelé binn, permettant de sérialiser de manière simple, rapide et compacte un grand nombre de types de donnée notamment des entiers de 8 à 64 bits.
L'utilisation de cette librairie dans le cadre de notre projet nous permet d'assurer l'unicité de la représentation des données échangées par les messages de l'application.

La librairie binn est composée d'un unique fichier source et de son fichier d'en-tête. Nous avons donc décider d'inclure ceux-ci directement aux sources de notre projet, afin que l'utilisation de notre librairie ne nécessite pas de résolution de dépendance. Les sources de cette librairie sont disponibles sur la plateforme GitHub sous la licence Apache 2.0, nous permettant ainsi de les utiliser librement sous condition d'inclure une copie de cette licence dans notre propre librairie.

Un exemple d'utilisation de la librairie dans le cadre de notre projet est fournit en annexe B page 15.

4.2 Librairie pthread

Nous utilisons aussi la librairie POSIX thread[2]. Cette librairie définie un ensemble types et de primitives permettant de créer des processus légers (threads). De plus, elle implémente de nombreux mécanismes de synchronisation, tels que les mutex que nous avons eu à utiliser dans notre bibliothèque.
Cette librairie est souvent incluse dans la Glibc.

4.3 Outils de développement

Nous avons utilisé **GCC** dans sa version 6.3.1 comme compilateur et éditeur de lien pour notre bibliothèque partagée (format .so).

La commande **ar** nous permet de produire une version statique de notre bibliothèque (format .a).

Pour automatiser la compilation de notre bibliothèque et de nos tests, nous avons utilisé la version 4.2.1 de l'utilitaire **make**.

Enfin, afin de gérer le versioning et de collaborer à plusieurs sur ce projet, nous avons utilisé l'outil **git** version 2.12.1 et la plateforme **GitHub**.

5 Algorithme d'exclusion mutuelle implémenté

5.1 Processus théorique

Le gestion des pages est centralisée par le Master, toute demande d'accès doit donc se faire avec l'aval de celui-ci.

La maître doit être notifié lors de chaque demande, qui se traduiront par un envoie de page. Chaque libération en écriture se traduit directement par un renvoie de la page modifiée au master. Cela évite d'avoir à redemander plus tard la page au site lors de la prochaine requête, elle aurait été de toute façon fatalement renvoyée dans tous les cas (ou alors la page n'est utilisé que par un unique site)

Chaque page est associée à une structure composée de :
- Un identifiant unique.
- Les droits actuels sur la page.
- Un mutex permettant l'accès exclusif à la page entre le thread principal et son daemon.

Et pour le maître :
- La socket de l'écrivain courant si il exsiste.
- Une liste *request_ queue* contenant les requêtes en attente (couple socket/droits)
- Une seconde liste *current_ readers* contenant tous les sites ayant en leur possession la dernière version de la page en lecture.

Les requêtes sont traitées de manière FIFO, lorsque qu'une requête est satisfaite, elle est supprimée de la liste et le maître passe à la suivante, et ce jusqu'à ce que la liste soit vide.

La primitive `process_list_requests()` du maître permet de vérifier si toutes les conditions sont réunies pour envoyer la page à un site. Une page lue peut être dupliquée à travers le réseau, la condition pour envoyer une page en lecture est simplement qu'il

ne doit pas y avoir d'écrivain possédant la page, dans le cas contraire, il faut attendre la libération de la page.

Une page demandée en écriture ne peut être envoyée que s'il n'y a pas d'autre écrivain, et que la liste des lecteurs courants est vide, et de la même manière, dans le cas contraire il faut attendre le relâchement de la page de la part des sites propriétaires.

Le problème d'envoyer une page à chaque demande est que si nous avons beaucoup de lecteurs utilisant une même page qui n'a pas été modifiée, il faut la redemander au maître alors qu'elle est déjà présente en local dans sa dernière version. Cela provoque un échange de message inutile. C'est pourquoi un bit de validité à été ajouté dans la structure, garantissant la conformité de la page auprès de maître et évitant la communication superflue.

La primitive `lock_read` retourne immédiatement si le bit de validité est à 1 et démarre une nouvelle requête dans le cas contraire.

Lors de la libération du verrou d'une lecture, le mutex de la page est relâché, permettant au daemon de pouvoir modifier la structure de celle-ci au besoin, en revanche le maître n'a pas conscience de cette libération.

Lorsque le site reçoit une demande d'invalidation, le verrou est pris de suite par le daemon, le bit de validité est mit à 0, les droits sont enlevés et *invalid_ack* est envoyé. Si le site était en train de lire la page, le daemon attendra simplement la libération du verrou pour envoyer l'*invalid_ack*.

Pour le maître, lorsqu'une demande en lecture est traitée, si il y a des lecteurs courant, une demande d'invalidation leur est envoyée. La requête d'écriture sera alors satisfaite lorsque tous les lecteurs auront accusé l'invalidation, garantissant leur lecture terminée.

Pour résumer :
- La primitive `process_list_requests()` a pour but de traiter les requêtes, par ordre d'arrivée.
- Elle est appelée à chaque demande de verrou, à chaque réception d'*invalidate_ack*, et à chaque réception de page (libération du verrou en écriture).
- Chaque requête satisfaite est supprimée de la liste des requêtes
- Si la requête est une demande d'écriture, elle provoque une demande d'invalidation pour tous les lecteurs courants.
- La page est délivrée à l'écrivain une fois que tous les lecteur on accusés l'invalidation.
- Une page détenue par un site en lecture peut être verrouillée sans passer par le maître si celle-ci n'a pas été modifiée entre temps.
- Toutes les requêtes seront satisfaites.

5.2 Cas pratiques

Voici la mise en pratique de l'algorithme pour toutes les configuration de lecture/écriture possibles. Pour chaque cas la liste représente les requêtes en attente pour une page donnée, la tête de celle ci étant à gauche. Le maître est seul responsable de la page, il

n'y a donc ni écrivain, ni lecteurs courants. Pour chaque requête, le chiffre correspond à l'identifiant de l'esclave.

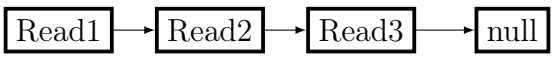


Figure 2 – Liste composée uniquement de lecteurs

Ici, le premier élément de la liste est un lecteur, n'ayant pas d'écrivain sur la page, nous pouvons satisfaire sa requête, la page lui est donc envoyée.'1' est ajouté dans la liste des lecteurs courants et sa requête est supprimée. Une page en lecture pouvant être répliquée sur plusieurs site, l'algorithme reboucle sur la requête suivante. Celle-ci est une requête de lecture de la part de site '2', le fait qu'il y ai déjà un lecteur ne pose pas de problème, sa demande est aussi satisfaite. Idem pour la requête du site 3. La liste des requêtes est donc vide, il y a 3 lecteurs courants sur la page ; '1', '2' et '3'.

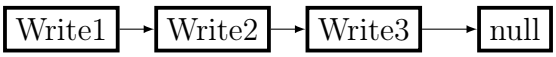


Figure 3 – Liste composée uniquement d'ecrivains

Dans ce second exemple, il n'y a que des écrivains dans la file. La première requête 'write1'est immédiatement satisfaite ; la page est envoyée. Étant donné qu'il ne peut y avoir qu'un seul écrivain, l'algorithme s'arrête. Seuls deux évènements peuvent alors se produire pour le relancer :

- Une nouvelle requête sur la page arrive au maître, cependant aucune demande pendante ne pourra être satisfaite car le site 1 détient toujours la page en écriture.
- Le processus écrivain relâche le verrou, délivrant ainsi au master la dernière version de la page. Nous sommes à présent revenus dans la configuration initiale, c'est à dire avec le maître possédant les droits sur la page, le déroulement sera donc le même pour 'write2', 'write3' ... Et ainsi de suite pour toute suite de requête writes consécutive.

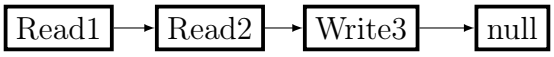
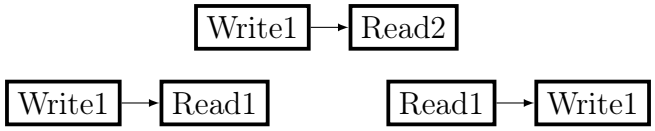


Figure 4 – Liste composée de lecteurs puis d'un écrivain

Le début de la file est identique au premier exemple, la page sera donc envoyé au site '1' puis au site '2', et la liste des lecteurs contiendra leur deux identifiants. Lors du traitement de la troisième requête, 'write3', une demande d'invalidation va être envoyée à chaque

lecteur courant, mais la requête d'écriture reste dans la file. Admettons que le premier *invalid_ack* reçu vienne du processus 1, il est alors retiré de la file des lecteurs, celle-ci contenant toujours au moins un identifiant, et donc un lecteur potentiel, la demande d'écriture est toujours mise en attente. L'*invalidate_ack* du site 2 est reçu, la liste des lecteurs est vide et le maître possède déjà la dernière version de la page qui a juste été lue. De retour dans l'état initial, la requête write3 peut être satisfaite comme dans l'exemple précédent.

Il reste 3 configurations qui n'ont pas été abordées en détail :



Pour chacunes d'entre elles, la logique est déductible à partir des exemples précédents.

Tous les entrelacement de lecture/écriture par deux sites distincts ou identiques sont gérés par l'algorithme, et donc toute requête est garantie d'être satisfaite dans un laps de temps fini, on évite ainsi tout problème potentiel de famine.

6 Pistes d'extensions et éléments pour la maintenance

6.1 Limitations

La principale limitation de notre bibliothèque est relative à l'utilisation de pointeurs au sein de la mémoire partagée.
Il est normal qu'un pointeur référençant une zone mémoire externe de la mémoire partagée ne puisse pas être utilisé dans cette dernière. Cependant, même un pointeur référençant une zone interne de la mémoire partagée ne doit pas se trouver dans la mémoire partagée. Ceci s'explique par une raison simple, la zone mémoire responsable d'accueillir les copies des pages est allouée avec l'appel système `mmap()`, l'adresse de base retournée n'est donc pas la même pour tous les processus.
En utilisant le flag MAP_FIXED en paramètre de l'appel à `mmap()`, il est possible de choisir l'adresse à laquelle la zone est allouée. Cependant il est spécifié que rien ne garantit que la zone n'a pas déjà été allouée, ce qui impliquerait une perte de donnée par écrasement, nous avons donc choisi de ne pas utiliser cette approche pour ne pas créer d'effets de bord indésirables pour les utilisateurs de notre bibliothèque.

Nous sommes partis du principe que les communications était fiables et que les sites ne pouvaient pas se déconnecter inopinément. Par conséquent si un site lock une page et se déconnecte (ou tout simplement oublie de l'unlock), la page sera indéfiniement inaccessible pour tout les autres sites.

De même, notre bibliothèque n'accepte aucune tolérance aux fautes. Si un processus contenant la dernière version d'une page venait à crasher, la page serait définitivement perdue.

6.2 Extensions possibles

Une bonne façon de régler le problème de lock permanent en cas de panne est d'instaurer un délai maximum entre l'envoie d'une donnée (en écriture) et la libération de son verrou. En cas de timeout, la version possédée par le master serait considérée comme à jour, et les requêtes suivantes pourraient ainsi être traitées. Il faudrait aussi supprimer toutes les occurrences de ce site déconnecté dans les autres pages (requêtes, lecteurs courant, écrivains...).

Pour une utilisation plus poussée, il faudrait plus mécanismes de synchronisations, comme pouvoir crée des verrous et des barrières entre un ensemble de sites. La barrière actuelle permet cependant de répondre à ce problème pour une utilisation basique de la bibliothèque.

Une autre extension possible aurait été de définir un handler pour le signal SIGSEGV qui est reçu lorsqu'une erreur de segmentation a lieu dans un programme. Dans ce handler, il aurait été possible de récupérer l'adresse à laquelle à eu lieu l'erreur, ainsi que le type d'accès (lecture ou écriture) sur une architecture x86_64 en lisant un registre particulier. Cette extension aurait donné la possibilité d'abstraire totalement tous les mécanismes de synchronisation pour l'utilisateur de la bibliothèque.

6.3 Maintenance

La documentation du code fournit et les commentaires ajoutés au code source de la bibliothèque peuvent faciliter la maintenance si besoin est.

Le code implémentant le protocole de communication par message est très facilement extensible. Nous avons eu plusieurs fois l'occasion lors du développement de rajouter des fonctionnalités à notre bibliothèque, telles que la barrière de synchronisation, en implémentant de nouveaux types de message.

7 Bibliographie et références

Références

[1] Binn : Binary Serialization Library,
`https://github.com/liteserver/binn`

[2] libpthread : POSIX Threading Library,
`https://www.gnu.org/software/hurd/libpthread.html`

8 Annexe A : Exemple d'utilisation de la bibliothèque réalisée

```
1  void *base_addr = InitSlave("127.0.0.1", 5555);
2
3  /* Wait for 2 other sites to fill the page */
4  sync_barrier(3);
5
6  lock_read(base_addr);
7  /*
8   * do some stuff with the page
9   */
10 unlock_read(base_addr);
11
12 /* End dsm */
13 QuitDSM();
```

9 Annexe B : Exemple d'utilisation de la librairie binn

```
1  #define DSM_MSG_KEY_TYPE "type"
2  #define DSM_MSG_KEY_PAGEID "pageid"
3  #define DSM_MSG_KEY_RIGHTS "rights"
4
5  dsm_message_t *msg;
6  binn *obj;
7  // creation d'un nouvel objet
8  obj = binn_object();
9
10 // ajout de donnees
11 binn_object_set_int32(obj, DSM_MSG_KEY_TYPE,
12                        msg->type);
13 binn_object_set_int32(obj, DSM_MSG_KEY_PAGEID,
14                        msg->lockpage_args.page_id);
15 binn_object_set_int16(obj, DSM_MSG_KEY_RIGHTS,
16                        msg->lockpage_args.access_rights);
17
18 // envoi sur le reseau
19 send(sock, binn_ptr(obj), binn_size(obj));
20
21 // liberation du buffer
22 binn_free(obj);
```