

Analyse du Code : Mutual Exclusion Distribuée

(Algorithme de Lamport en C avec sockets)

Yassine OUAHMANE – AKHMARI Mohammed-Yassine – Wail YACOUBI

Contents

1	Introduction	2
2	Résumé du Fonctionnement du Code	3
2.1	Horloge logique	3
2.2	Structure de la file de requêtes	3
2.3	Entrée en section critique	4
2.4	Sortie de section critique	4
3	Propriétés de l'Implémentation	5
3.1	Sécurité (Safety)	5
3.2	Vivacité (Liveness)	5
3.3	Ordre total	5
4	Limites de l'Implémentation	6
4.1	1. Pas de mécanisme de tolérance aux pannes	6
4.2	2. Scalabilité limitée	6
4.3	3. File stockée localement	6
4.4	4. Pas de détection ou reprise après déconnexion	7
4.5	5. Threads de réception sans gestion d'erreurs	7
5	Améliorations Possibles	8
5.1	1. Passage à Ricart–Agrawala	8
5.2	2. Exclusion mutuelle par jeton (token ring)	8
5.3	3. Ajout de timeouts et retransmissions	8
5.4	4. Optimisation de la file	8
5.5	5. Threads : meilleure synchronisation	9
5.6	6. Support de réseaux distribués réels	9
6	Cas Particuliers	10
6.1	Perte de messages	10
6.2	Deux processus envoyant REQ simultanément	10
6.3	Processus lent	10
7	Conclusion	11

Chapter 1

Introduction

Ce rapport analyse une implémentation C de l’algorithme de Lamport pour l’exclusion mutuelle distribuée. Le programme repose sur une architecture *pair-à-pair*, utilise les horloges de Lamport, des sockets TCP, et une file globalement ordonnée par timestamps.

L’objectif de ce document est de fournir :

- une description détaillée du fonctionnement,
- une analyse des propriétés de sûreté et vivacité,
- une étude de scalabilité,
- les limites de l’implémentation,
- des propositions d’améliorations,
- l’analyse de cas particuliers.

Chapter 2

Résumé du Fonctionnement du Code

L'algorithme implémente les trois types de messages classiques de Lamport :

- **REQ** : demande d'entrée dans la section critique,
- **ACK** : accusé de réception d'une requête,
- **REL** : libération du verrou.

2.1 Horloge logique

L'horloge de Lamport est stockée dans :

```
int clock;
```

Deux opérations sont disponibles :

- `get_clock()` pour générer un nouveau timestamp,
- `update_clock()` lors de la réception d'un message.

2.2 Structure de la file de requêtes

La file est un tableau trié globalement :

```
typedef struct {
    int timestamp;
    int pid;
    int active;
} Request;
```

Elle est triée par :

1. timestamp croissant,
2. puis pid croissant (définit un ordre total).

2.3 Entrée en section critique

La procédure `request_lock()` :

1. incrémente l'horloge,
2. ajoute sa propre requête,
3. diffuse un message **REQ**,
4. attend :
 - d'être premier dans la file,
 - d'avoir reçu un **ACK** de chaque processus.

2.4 Sortie de section critique

La procédure `release_lock()` :

1. retire sa requête de la file,
2. diffuse un message **REL**.

C'est conforme au protocole de Lamport.

Chapter 3

Propriétés de l'Implémentation

3.1 Sécurité (Safety)

L'algorithme garantit qu'un seul processus entre dans la section critique à la fois. La preuve repose sur :

- l'ordre total induit par (timestamp, pid),
- la condition que chaque processus doit recevoir tous les ACK,
- le fait que les requêtes sont insérées dans un ordre déterministe.

3.2 Vivacité (Liveness)

Tout processus finit par entrer en section critique si :

- tous les processus répondent,
- aucune perte de message ne se produit,
- aucune connexion réseau ne se bloque.

3.3 Ordre total

L'algorithme respecte la relation de Lamport :

$$(t_i, pid_i) < (t_j, pid_j)$$

Chapter 4

Limites de l'Implémentation

4.1 1. Pas de mécanisme de tolérance aux pannes

Si un seul processus tombe en panne :

- aucun autre ne peut plus entrer en section critique,
- car il manquera un ACK.

4.2 2. Scalabilité limitée

L'algorithme de Lamport nécessite :

$$O(N)$$

messages pour :

- chaque requête,
- chaque libération.

Donc un coût global :

$$O(N) \text{ communication par section critique.}$$

Pour un grand nombre de processus (ex : $N = 1000$), la charge explose.

4.3 3. File stockée localement

Chaque processus maintient sa propre file :

- Possible incohérence temporaire,
- Pas de validation croisée,
- Si un message est perdu, l'ordre est brisé.

4.4 4. Pas de détection ou reprise après déconnexion

Le code n'a pas de :

- timeouts,
- reconnexion automatique,
- retransmission.

4.5 5. Threads de réception sans gestion d'erreurs

Le receiver thread ne gère pas :

- $\text{recv} = 0$ (déconnexion),
- $\text{recv} = -1$ (erreur),
- messages partiels.

Chapter 5

Améliorations Possibles

5.1 1. Passage à Ricart–Agrawala

Permet de réduire :

$$2N \rightarrow N$$

messages.

5.2 2. Exclusion mutuelle par jeton (token ring)

Coût message :

$$O(1)$$

par entrée en section critique.

Plus scalable pour N grands.

5.3 3. Ajout de timeouts et retransmissions

Implémenter :

- accusés de réception au niveau socket,
- timers,
- reconnexion après crash.

5.4 4. Optimisation de la file

Utilisation d'un :

- tas binaire (min-heap),
- arbre équilibré (red-black),
- skip-list.

Au lieu d'un tableau shifté en $O(N)$.

5.5 5. Threads : meilleure synchronisation

- utiliser des `condition_variables`
- éviter le busy-waiting (`usleep`)

5.6 6. Support de réseaux distribués réels

IP configurable au lieu de 127.0.0.1.

Chapter 6

Cas Particuliers

6.1 Perte de messages

Lamport suppose un canal fiable. Avec TCP cela reste rare mais possible en cas de :

- coupure réseau,
- fermeture intempestive,
- timeout.

Le système se bloque définitivement.

6.2 Deux processus envoyant REQ simultanément

Grâce aux horloges de Lamport :

- leurs timestamps augmentent,
- en cas d'égalité, le pid tranche.

C'est bien géré.

6.3 Processus lent

Un processus lent retarde :

tous les autres

car ils attendent son ACK.

Chapter 7

Conclusion

Le code fourni constitue une implémentation fidèle et fonctionnelle de l’algorithme de Lamport pour l’exclusion mutuelle distribuée. Il respecte les propriétés théoriques du modèle et fournit une base solide pour expérimenter.

Cependant :

- la scalabilité est limitée ($O(N)$ messages),
- la tolérance aux pannes est inexistante,
- la file pourrait être optimisée,
- la communication réseau doit être renforcée.

Pour un système de production, il serait conseillé de migrer vers :

- Ricart–Agrawala,
- Token Ring,
- ou un protocole de consensus (Paxos, Raft).