Rapport TP - Algorithmes de Broadcast Distribués

Auteur: [Votre Nom]

Date: 11 Novembre 2025

Module: Systèmes Distribués

Table des Matières

---

1. Introduction

Ce TP implémente plusieurs algorithmes de broadcast distribués en Go, dans un environnement simulé avec pannes de processus et pertes réseau. Les algorithmes implémentés sont :

- **Best-Effort Broadcast (BEB)** : Diffusion sans garantie
- **Reliable Broadcast (RB)** : Diffusion fiable avec eager relaying
- **Uniform Reliable Broadcast (URB)** : Diffusion uniforme avec acquittements
- **FIFO Broadcast** : Diffusion avec ordre FIFO par émetteur
- **Causal Broadcast** : Diffusion avec ordre causal global
- **Total Order Broadcast** : Diffusion avec ordre total (algorithme de Skeen)

Objectifs

- Comprendre les propriétés de chaque algorithme
- Gérer la concurrence et les pannes
- Mesurer les performances et les compromis

---

2. Choix d'Implémentation

2.1 Structures de Données

Message

```go
type Message struct {
    ID          string      // Identifiant unique
    Type        MessageType // Type pour le routage
    Sender      string      // Émetteur
    Content     interface{} // Contenu flexible
    SeqNum      int         // Pour FIFO
    VectorClock []int       // Pour Causal
    ProposedTS  float64     // Pour Total Order
}
```

Justification : Structure flexible permettant de supporter tous les algorithmes avec un seul type de message.

Process - Champs Principaux

- **delivered map[string]int** : Évite les duplications, compte les livraisons
- **received map[string]bool** : Pour RB, marque les messages déjà relayés
- ***pending map[string][]Message** : Buffer pour messages hors-ordre (FIFO/Causal)
- **ackTrack map[string]map[string]bool** : Pour URB, suit les acquittements
- **vectorClock []int** : Horloge vectorielle pour ordre causal
- **proposals map[string]map[string]float64** : Pour Total Order, collecte des timestamps

Justification : Séparation claire entre "déjà vu" (received) et "déjà délivré" (delivered) pour implémenter correctement le relaying.

2.2 Gestion de la Concurrence

Pattern utilisé :

```go
```

```
p.mu.Lock()
// Lecture des données
data := p.someData
p.mu.Unlock()

// Opérations externes (réseau, canaux)
p.network.Send(...)
```

Règles suivies :

1. Toujours libérer le verrou AVANT d'écrire sur un canal ou d'appeler le réseau
2. Minimiser la durée des sections critiques
3. Utiliser defer uniquement quand aucun appel externe n'est fait
4. Tester avec go test -race systématiquement

2.3 Choix Architecturaux

Relaying à la Réception vs à la Livraison

Choix fait : Relayer dès la réception (dans RBReceive), pas à la livraison.

Raison :

- Maximise la propagation même si la livraison est retardée
- Garantit que tous les processus corrects reçoivent le message
- Tolère mieux les pannes et les contraintes d'ordre

Séparation des Types de Messages

Chaque algorithme a son propre MessageType (BEB_DATA, RB_DATA, etc.) pour un routage clair dans Process.Receive().

---

3. Analyse de Complexité

3.1 Best-Effort Broadcast

Métrique	Valeur	Explication
Messages	O(n)	1 message par processus
Latence	O(1)	1 saut réseau
Mémoire	O(1)	Pas de buffer
Bande passante	Minimale	Pas de duplication

3.2 Reliable Broadcast

Métrique	Valeur	Explication
Messages	$O(n^2)$	Chaque processus relaye à n-1 voisins
Latence	O(1)	1 saut (avec relaying parallèle)
Mémoire	O(m)	m = nombre de messages en vol
Tolérance pannes	Élevée	Même si émetteur crashe après 1 envoi

Détail calcul messages :

- Émetteur vers n-1 processus
- Chaque receveur vers n-1 autres (relaying)
- Total : $n-1 + (n-1) \times (n-1) \approx n^2$

3.3 Uniform Reliable Broadcast

Métrique	Valeur	Explication
Messages	$O(n^2)$	DATA + ACKs à tous
Latence	O(1)	Parallèle si majorité disponible
Mémoire	$O(m \times n)$	Suivi des ACKs par message
Tolérance pannes	Jusqu'à $\lfloor (n-1)/2 \rfloor$	Jusqu'à $\lfloor (n-1)/2 \rfloor$ crashes

Avantage sur RB : Garantie uniforme - si un processus (même crashé) délivre, tous les corrects délivrent.

3.4 FIFO Broadcast

Métrique	Valeur	Explication
Messages	$O(n^2)$	Hérite de RB
Latence	O(1) + délai buffer	Attente si messages hors ordre
Mémoire	$O(n \times w)$	w = taille fenêtre par émetteur
Overhead	+4 bytes	Numéro de séquence (int)

Cas optimal : Messages arrivent dans l'ordre, livraison immédiate
Cas pire : Messages arrivent en ordre inverse, tous bufferisés

3.5 Causal Broadcast

Métrique	Valeur	Explication
Messages	$O(n^2 \times n)$	RB + horloge vectorielle (n ints)
Latence	O(1) + délai buffer	Attente si dépendances causales
Mémoire	$O(m \times n)$	Buffer + horloges

Métrique	Valeur	Explication
Overhead	+n×4 bytes	Horloge vectorielle complète

Condition de délivrance :

$$VC_{msg}[sender] = VC_{local}[sender] + 1 \quad ET$$
$$\forall k \neq sender : VC_{msg}[k] \leq VC_{local}[k]$$

3.6 Total Order Broadcast (Skeen)

Métrique	Valeur	Explication
Messages	$O(3n^2)$	DATA + PROPOSE + DECISION
Latence	O(3)	3 phases séquentielles
Mémoire	O(m × n)	Propositions + buffer
Synchronisation	Élevée	Coordination nécessaire

Phases :

1. Phase 1 : Émetteur vers DATA à tous (n messages)
2. Phase 2 : Tous vers PROPOSE à tous (n^2 messages)
3. Phase 3 : Coordinateur vers DECISION à tous (n messages)

Total : $n + n^2 + n \approx \mathbf{3n^2}$ messages

---

4. Impact des Pannes

4.1 Best-Effort Broadcast

Scénario : Émetteur crashe après avoir envoyé à $k < n$ processus

Résultat : Les n-k autres processus ne reçoivent jamais le message

Propriétés perdues :

- Pas d'agreement
- Pas de garantie de livraison

4.2 Reliable Broadcast

Scénario : Émetteur crashe après avoir envoyé à $k \geq 1$ processus

Résultat : Tous les processus corrects finissent par délivrer

Mécanisme : Le relaying eager propage le message même si l'émetteur disparaît.

Exemple avec 5 processus :

```
t=0 : P0 envoie à P1, P2
t=1 : P0 crashe
t=2 : P1 relaye à P2, P3, P4
t=3 : P2 relaye à P1, P3, P4
→ Tous ont reçu le message malgré le crash
```

4.3 Uniform Reliable Broadcast

Scénario : Un processus délivre puis crashe immédiatement

RB : Les autres pourraient ne jamais délivrer
URB : Tous les processus corrects délivreront (propriété uniforme)

Condition : Majorité d'ACKs garantit que le message survivra aux crashes.

4.4 FIFO/Causal/Total Order

Impact panne émetteur :

- Les messages déjà envoyés sont délivrés (grâce à RB sous-jacent)
- L'ordre est préservé parmi les messages délivrés
- Pas de nouveaux messages de cet émetteur

Impact panne receveur :

- Les autres processus continuent normalement
- Le processus crashé manque des messages

4.5 Total Order avec Crash du Coordinateur

Problème : Si le coordinateur crashe pendant la phase 2 (collecte des propositions), le protocole peut bloquer.

Solution non implémentée : Détection de timeout et réélection d'un coordinateur.

---

5. Résultats Expérimentaux

5.1 Configuration des Tests

```
go
```

Réseau : MinDelay=0.1s, MaxDelay=0.5s, LossRate=0-20%, DuplicationRate=0-100%
Processus : 3 à 5
Messages : 1 à 20 par test

5.2 Tests Passés

TestBEBValidity : L'émetteur se délivre son propre message

TestBEBNoDuplication : Pas de duplications malgré duplication réseau

TestRBValidity : Propriété de validité respectée

TestRBNoDuplication : Pas de duplications avec duplication réseau (100%)

TestRBWithLossDupl : Fonctionne avec pertes et duplications

TestRBWithCrash : Tous les processus délivrent même si émetteur crashe

TestFIFOOrder : Messages délivrés dans l'ordre FIFO

TestCausalOrder : Ordre causal respecté (m1 avant m2)

TestTotalOrder : Ordre total cohérent entre processus

TestFailureScenarios : Robustesse aux pannes à différents moments

TestWithNetworkLoss : Tolérance aux pertes réseau

TestMultipleSenders : Gestion de plusieurs émetteurs concurrents

5.3 Métriques Observées

BEB (5 processus, 10 messages)

Messages envoyés : 50 (10 × 5)
Messages délivrés : 50
Latence moyenne : ~0.3s

RB (5 processus, 1 message, perte 10%)

Messages envoyés : ~30-40 (relaying)
Messages perdus : ~3-4
Messages délivrés : 5 (tous)
Latence : ~0.4s

FIFO (4 processus, 10 messages séquentiels)

Messages en buffer (max) : 0-3
Ordre respecté : 100%
Latence : 0.3s - 1.5s (selon arrivée)

Causal (3 processus, 2 messages avec dépendance)

Total Order (4 processus, 3 messages)

```
Messages totaux : ~36 (DATA + PROPOSE + DECISION)
Phases : 3
Latence : ~1.5-2.0s
Ordre identique : 100% sur tous les processus
```

5.4 Observations

1. **BEB** : Le plus rapide mais aucune garantie - inutilisable en production
2. **RB** : Excellent compromis performance/fiabilité - 2-3× plus de messages que BEB
3. **URB** : Overhead limité par rapport à RB (~10-20%) mais garanties renforcées
4. **FIFO** : Performance proche de RB si messages arrivent dans l'ordre
5. **Causal** : Overhead significatif (horloge vectorielle) mais ordre causal crucial pour certaines applications
6. **Total Order** : Le plus coûteux (3× latence) mais indispensable pour cohérence forte

---

6. Difficultés Rencontrées

6.1 Data Races

Problème initial :

```go
// MAUVAIS : Verrou tenu pendant l'envoi réseau
p.mu.Lock()
for _, n := range p.neighbors {
    p.network.Send(...) // Peut bloquer
}
p.mu.Unlock()
```

Solution :

```go
```

```
// BON : Copier puis libérer
p.mu.Lock()
neighbors := p.neighbors
p.mu.Unlock()

for _, n := range neighbors {
    p.network.Send(...)
}
```

Détection : go test -race a été crucial.

6.2 Deadlocks sur deliverChan

Problème : Écrire sur deliverChan avec le verrou tenu pouvait bloquer si le canal était plein.

Solution : Toujours libérer le verrou AVANT d'écrire sur le canal :

```
go

p.mu.Unlock() // IMPORTANT : Libérer ici


p.deliverChan <- msg // Maintenant sûr
```

6.3 Condition Causale

Difficulté : Comprendre exactement quand un message peut être délivré.

Condition finale :

```
go

VC_msg[sender] == VC_local[sender] + 1  // Suivant attendu
ET ∀k ≠ sender : VC_msg[k] ≤ VC_local[k] // Pas de dépendances manquantes
```

Astuce : Tester avec seulement 2-3 processus et tracer les horloges à chaque étape.

6.4 Total Order - Coordinateur Unique

Problème : Algorithme de Skeen original nécessite un coordinateur, ce qui crée un point de défaillance unique.

Choix fait : Chaque processus peut initier un broadcast, mais le protocole peut bloquer si crash pendant phase 2.

Solution idéale (non implémentée) : Timeout et réélection de coordinateur.

6.5 Relaying vs Livraison

Confusion initiale : Quand relayer le message ?

Réponse : **Dès la réception** (dans RBReceive), pas à la livraison, pour garantir la propagation même si livraison retardée.

---

7. Améliorations Possibles

7.1 Optimisations de Performance

1. **Compression des horloges vectorielles**
   - Utiliser des deltas au lieu d'envoyer l'horloge complète
   - Réduction de O(n) à O(k) où k = nombre de changements
2. **Buffer avec priority queue**
   - Utiliser container/heap pour FIFO/Causal
   - Améliore de $O(n^2)$ à O(n log n) le tri des messages en attente
3. **Batching des ACKs**
   - Pour URB, envoyer des ACKs groupés
   - Réduit le nombre de messages

7.2 Fonctionnalités Supplémentaires

1. **Détection de pannes**
   - Timeouts + heartbeats
   - Permet de libérer les ressources pour processus crashés
2. **Membership dynamique**
   - Ajout/retrait de processus pendant l'exécution
   - Nécessite protocole de reconfiguration
3. **Persistance**
   - Sauvegarder l'état sur disque
   - Reprise après crash
4. **Métriques avancées**
   - Histogrammes de latence
   - Percentiles (p50, p99)
   - Grafana/Prometheus pour visualisation

7.3 Améliorations Théoriques

1. **URB optimisé**

- Utiliser des signatures de groupe pour réduire les ACKs
- Complexité messages : O(n) au lieu de $O(n^2)$

2. **Total Order sans coordinateur**
   - Algorithme de Lamport généralisé
   - Plus tolérant aux pannes
3. **Causal Broadcast optimisé**
   - Matrices causales creuses
   - Compression des dépendances

---

8. Conclusion

8.1 Synthèse des Apprentissages

Ce TP m'a permis de :

1. **Comprendre les propriétés fondamentales** des algorithmes de broadcast
   - Validité, Agreement, Intégrité
   - Ordre FIFO, Causal, Total
2. **Maîtriser la programmation concurrente en Go**
   - Gestion correcte des mutex
   - Éviter data races et deadlocks
   - Utilisation de canaux pour communication
3. **Analyser les compromis** entre différentes approches
   - Performance vs garanties
   - Complexité vs robustesse
   - Overhead vs propriétés
4. **Implémenter des algorithmes distribués classiques**
   - Eager relaying pour fiabilité
   - Horloges vectorielles pour causalité
   - Protocole de consensus (Skeen) pour ordre total

8.2 Applications Pratiques

Les connaissances acquises sont directement applicables à :

- Systèmes de messagerie distribués
- Réplication de bases de données
- Systèmes de coordination (Zookeeper, etcd)
- Applications blockchain
- Systèmes de gestion de configuration

8.3 Perspectives

Ce TP constitue une base solide pour explorer des sujets plus avancés comme :

- Consensus Byzantine (PBFT)
- Algorithmes de Paxos et Raft
- Protocoles de commit distribués (2PC, 3PC)
- Cohérence causale dans les systèmes géo-distribués