Replication et Durabilité dans les systèmes répartis

Lambert SONNA MOMO

19 février 2001

0.1 Résumé

Les bases de données réparties et la replication des données sont reconnues aujourd'hui comme étant des moyens éfficaces pour augmenter la disponibilité et la fiabilié des bases de données. La replication offre aux utilisateurs de meilleures performances et une plus grande disponibilité des données. Toutefois, celle-ci introduit le problème de cohérence mutuelle des copies. La mise à jour des données doit prendre effet sur toutes les copies.

La notion de durabilié est capitale dans les bases de données. Elle assure que lorsqu'une base de données tombe en panne, les transactions qui ont réussi leur commit sont effectivement préservées sur la mémoire stable de la base de données. La durabilité est donc une technique de tolérance aux pannes .
Une autre technique de tolérance aux pannes est celle de la replication, des copies multiples assurent que, si une copie tombe en panne, les autres copies continuent de maintenir le service. Naturellement, ces deux techniques de tolérance aux pannes ont leur prix, que ce soit en terme de complicité ou de performance.

Le présent travail de semestre a pour but d'explorer comment les garanties de tolérance aux fautes liées á la durabilité et à la replication se complètent et voir dans quelle mesure les règles de durabilité peuvent être assouplies en présence de replication. L'idée est d'éviter que les deux approches offrent une trop de tolérance aux pannes à un prix prohibitif. Ce projet se situe à l'intersection des mondes des bases de données et des systèmes distribués. Il permet en outre de se familiariser avec des techniques de simulation.

0.2 Abstract

Distributed dats bases and data replication are recognised today as an efficient way to increase availability and fiability in data bases. The problem of mutual coherence of copies is however introduced. Updating data is to be done on all copies.

The notion of durability is capital in database. It assume that when a database crash, comited transactions are load on a stable database memory. Durability is a technique of fault tolerance.

Another technique of fault tolerance is replication, multiple copies assume that when one of them crash, the rest of copies continue to operate. This two techniques has their disavantages in terms of complexity and performance.

This semester work, look how durability and replication work together. The performance criteria of the simulation are expressed in terms of the generated traffic, response time and duration of unavailability of copies.

Table des matières

Table des figures

Chapitre 1

Introduction

Ce premier chapitre situe brièvement le contexte de ce travail de semestre, énonce les objectifs visés, présente les contributions apportées, et présente l'organisation de ce rapport.

1.1 Contexte

Le domaine des applications réparties, parfois appelé Informatique répartie, ne cesse de croître. Cette avancée de l'utilisation de l'Informatique, en tant qu'outil privilégié, dans des domaines de plus en plus divers (applications télématiques par exemple), est essentiellement le résultat du développement de la science et de la technique Informatique [MRL00].

La maîtrise des applications réparties et des outils qui permettent de les construire passe en effet par la connaissance des éléments fondamentaux de ce qu'il est convenu d'appeler un système réparti. Par rapport à un système traditionnel(généralement qualifié de : système opératoire centralisé) un système réparti présente une différence essentielle par l'échange des messages(il n'y a pas de mémoire centrale qui leur servirai de lieu d'échanges). Cette caractéristique, jointe d'une part à l' évolution de la technologie matérielle et d'autre part à celle de la méthodologie de conception et d'écriture du logiciel , suffit à montrer que la maîtrise des systèmes répartis passe par la connaissance de concepts, d'outils, d'algorithmes et de méthodes spécifiques. Le contexte de ce travail de semestre comporte deux volets : La replication et la durabilité dans les systèmes répartis.

1.2 Objectifs

Les objectifs de ce travail de semestre sont :

- Comprendre les techniques de la replication dans les bases de données.
- Comment les garanties de tolérance aux fautes liées à la replication et la durabilité se compètent.
- Comment les règles de durabilité peuvent être assouplies en présence de la replication.

Chapitre 2

La Tolérance aux fautes dans les systèmes répartis

2.1 Introduction

La tolérance aux fautes s'inscrit dans le contexte plus large de la sûreté de fonctionnement. La sûreté de fonctionnement (dependability) d'un système informatique est la propritété qui permet à ses utilisateurs de placer une confiance dans le service qu'il délivre. Le service délivré par un système est son comportement tel qu'il est perçu par ses utilisateurs[1]

2.2 Définitions

La terminologie présentée dans cette section est inspirée de [PUS00]. La traduction anglaise de chaque terme défini est donnée entre parenthèses pour permettre au lecteur de faire le lien avec la terminologie utilisée dans la communauté anglephone.

Attributs de la sureté de fonctionnement.

La sûreté de fonctionnement(dependability) d'un système peut être abordé sous des angles différents, selon les fonctions que remplit le système et selon

[1]Un utilisateur est un autre système(humain ou physique) qui interagit avec le système considéré.

le domaine d'applications auquel il est destiné. Ces points de vue correspondent à des attributs de la sûreté de fonctionnement sur lesquels les auteurs peuvent mettre plus ou moins l'accent.

- la **disponibilité**(availability) définit le fait d'être prêt à l'utilisation.
- la **fiabilité**(reliability) d'un système est definit comme une fonction R(t) du temps qui répresente la probabilité que le système survive jusqu'au temps t.
- la **sûreté**(safety) respecte la non occurence de défaillance catastrophique.
- la **sécurité-confidentialité**(security) prevoit la non occurence des accès non autorisés ou l'acquisition non autorisé des informations.
- la **maintenabilité**(maintenability) définit l'aptitude aux réparations et aux évolutions.

La qualité du service delivré par le système est determiné par l'efficacité des moyens assurant, la sûreté de fonctionnement sur les entraves de la sûreté de fonctionnement.

Entraves à la sureté de fonctionnement.

Mettre en oeuvre la sûreté de fonctionnement d'un système correspond à lutter contre les défaillances du système. Une défaillance(failure) survient losque le service delivré par le système ne correspond plus à sa specification. La spécification du service correspond à la description du service que les utilisateurs sont à mesure d'attendre du système. Une erreur(Error) est une anomalie de l'état du système et susceptible d'entrainer une défaillance. Une erreur ne provoque pas systèmatiquement une defaillance. Le sytème peut continuer à délivrer un service correct malgrè un certain nombres d'erreurs affectant son état. Une faute(fault) est la cause d'une erreur. Une erreur peut provoquer une défaillance : Une altération du service delivré par le système.

faute(fault)->erreur dans l'état du système -> défaillance dans le système.

La distinction entre faute, erreur et défaillance n'est pas absolue. Elle dépend de la position du point d'observation par rapport aux frontières du système consideré. La défaillance d'un sous système constitue une faute pour le système englobant. La figure 2.1illustre cet exemple. Considerons un composant C1 relialisé à

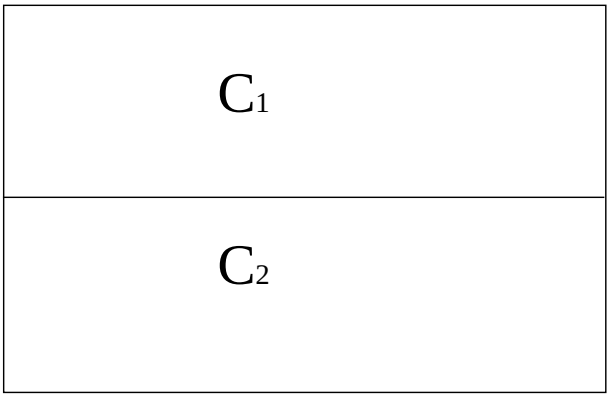


FIG. 2.1: Faute, erreur, et défaillance

l'aide d'un autre composant C2(C2 offre un service à C1). On peut constater :

- Une défaillance de C2 peut causer une faute de C1.
- Une faute de C1 peut causer une erreur de C1.
- Une erreur de C1 peut causer une défaillance de C1.

Ceci nous donne la chaine suivante :

... -> défaillance -> faute -> erreur -> defaillance ->...

Les défaillances pouvant affecter un système sont variées. L'élaboration d'une stratégie éfficace de lutte contre les défaillances nécessite une caractérisation précise des défaillances à combatre. Les lignes suivantes sont inspirées de [PUS00]. La figure 2.2 présente les classes de défaillance les plus fréquentes. Elles sont classifiées en quatre catégories, des moins graves aux plus graves. Lorsque le système omet de délivrer le service demandé, il exhybe une défaillance par **omission**(omission fault). Si l'omission devient permanente (le système ne repond plus), le système exhybe une défaillance par **arrêt**(Crash fault). Lorsque les conditions temporelles ne sont plus satisfaites (le système repond trop vite ou trop tard), le système exhybe une défaillance **temporelle**(Timing fault). Si le système exhybe chacun de ces comportements de manière imprévisible, il est affecté par des défaillances **arbitraires**(Byzantine fault).

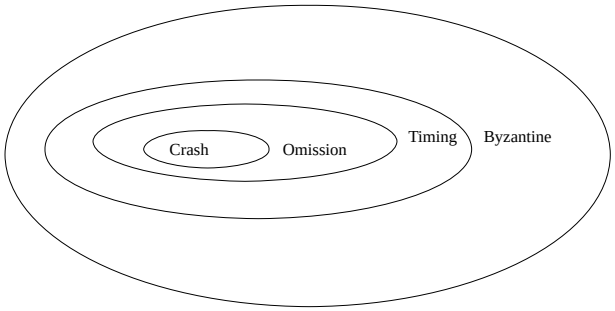


FIG. 2.2: Classes de défaillance

Moyens permettant d'assurer la sureté de fonctionnement.

Plusieurs moyens sont généralement combinés pour mettre en oeuvre la sûreté de fonctionnement d'un système :

- la **prevention des fautes**(fault prevention) consiste à empêcher l'ocurrence des fautes ;
- **la tolérance aux fautes**(fault tolerance) consiste à délivrer un service correct en dépit de l'ocurrence de fautes ;
- l'**élimination des fautes**(fault removal) consiste à reduire le nombre et la sévérité des fautes dans le but de les éliminer du système ;
- **prévision des fautes**(fault forecasting) consiste à estimer le nombre de fautes courantes et futures ainsi que leurs conséquences.

2.3 Techniques permettant d'atteindre la tolérance aux fautes

La tolérance aux fautes est mise en oeuvre par la combinaison de deux techniques.

- Le **traitement de la faute**(fault treatment) qui vise à éviter qu'une faute survenue ne se reproduise.
- Le **traitement d'erreur**(error processing) qui vise à éliminer une erreur avant qu'elle ne produise une défaillance [PUS00].

2.3.1 Traitement de faute

L'objectif du traitement de faute vise à éviter qu'une faute ne se reproduise. On procède au **diagnostic** de faute(fault diagnosis) qui vise à identifier la faute, puis, on procède à la **passivation**(fault passivation) de la faute qui consiste généralement à neutraliser les composants fautifs en les excluant du système.

2.3.2 Traitement de l'erreur

L'objectif du traitement de l'erreur est d'éliminer une erreur affectant le système afin qu'elle n'entraine de défaillance . Elle peut s'exprimer sur deux formes.

- Le **recouvrement d'erreur** consiste à remplacer l'état erroné du système par un état correct ;

- La **Le masquage d'erreur** consiste à compter sur la redondance présente dans le système pour que celui-ci continue à délivrer un service correct malgrè un état erroné.

2.3.3 Recouvrement d'erreur

Deux methodes sont possibles :

- la **reprise** consiste à remplacer l'état arroné par un état correct dans lequel le système était avant l'ocurrence de l'erreur.

- la **poursuite** consiste à remplacer l'état erroné par un nouvel état correct construit à partir de l'état erroné.

2.3.4 Compensation d'erreur

Elle peut prendre deux formes :

- la **détection et compensation d'erreur**.
- le **masquage d'erreur**.

La détection et compensation d'erreur consiste à remplacer le composant erroné par un composant correct. Le masquage d'erreur consiste en une compensation d'erreur. La reprise demande de faire régulierement de sauvegardes de l'état du système appelé **point de reprise**. Pour transformer un état erroné en un état correct, on rénitialise l'état du système à partir du dernier point de reprise. La poursuite consiste à construire un nouvel état correct à partir de l'état erroné.

2.4 Modélisation d'un système réparti

2.4.1 Modèle physique

Il consiste en un ensemble d'ordinateurs autonomes géographiquement dispersés et interconnectés par un réseau de lignes de communication. Chaque noeud est composé essentiellement par un processeur, une horloge, une memoire centrale(volatile), une memoire secondaire(non volatile) et une interface raccordée au réseau de communication. Les noeuds ne partagent aucun composant et ils communiquent exclusivement à travers le réseau de communication [KRM96].

2.4.2 Modèle logique

Ce modèle decrit un système réparti du point de vue programmes qui s'exécutent sur le système. Il consiste en un ensemble de processus s'exécutant de manière concurrente et coopérant pour réaliser une tâche commune. Un processus représente l'exécution d'un programme sur un noeud du système. Les processus coopèrent en échangeant des messages . Un message est un ensemble d'information qu'un processus desire communiquer à un ou plusieurs processus du système. Cet échange de message permet aux processus de prendre des decisions concernant l'état du système. Un message est transmis d'un processus à un autre par le biais d'un **canal de communication**. Un canal répresente une connection logique entre deux processus.

2.4.3 Modèle temporel

On distingue généralement deux approches : l'approche synchrone et l'approche asynchrone. L'approche **synchrone** consiste à supposer que [HTO93] :

– Il existe une borne supérieure connue comme délai de transmission d'un message. Ce délai comprend le temps nécessaire de l'émission, la transmission et la réception du message.

– Chaque processus a une horloge logique et la dérive de cette horloge par rapport au temps réel a une borne supérieure connue.

– Il existe une borne inférieure et une borne supérieure connues au temps nécessaire à un processus pour exécuter une instruction de son programme.

L'existence de ces bornes permettent de définir la notion de **délai de garde**(timeout), qui correspond au délai maximumn au bout duquel un message doit être acquité par le destinataire. Si au terme du délai de garde, l'acquitement ne parvient pas à l'expéditeur, une défaillance du destinataire ou du réseau est survenue.

2.4.4 Défaillance dans un système réparti

Un système distribué peut être vu comme un graphe dont les noeuds sont des sites et les arcs bidirectionnels répresentent les canaux de communications entre les noeuds [BHG87], ceci peut être illustré par la figure 2.3. Pour chaque noeud Il existe un chemin à tout autre noeud du graphe. Les sites peuvent communiquer soit directement, soit indirectement via les canaux de communications. La défaillance d'un canal ou d'un processus modélise une défaillance affectant un noeud ou une ligne du réseau. Cette défaillance d'un composant physique est causée par une faute matérielle ou logicielle. Pour clarifier la notion de défaillance pour un processus et pour un canal, il est utile de définir précisement, pour chaque classe de défaillance, la manifestation d'un composant défaillant.

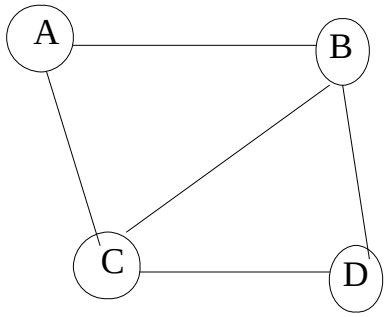


FIG. 2.3: Système réparti

Un processus exhibe :

- Une défaillance par arrêt s'il s'arrête prématurément de façon définitive ;
- Une défaillance par omission s'il omet d'envoyer ou de recevoir un message ;
- Une défaillance arbitraire s'il a un comportement imprévisible(une action non prevue) ;

Un canal exhibe :

- Une défaillance par arrêt s'il cesse définitivement de transmettre les messages qui lui sont confiés ;
- Une défaillance par omission s'il omet de transporter un message.
- Une défaillance temporelle si le temps de transmission d'un message ne respecte pas les bornes temporelles ;
- Une défaillance arbitraire s'il a un comportement imprévisible(génération d'un message erroné).

Toute défaillance par omission peut être vu comme un cas particulier de défaillance temporelle. Les défaillances temporelles n'ont de sens que dans un système synchrone.

2.4.5 Mecanismes de base pour la tolérance aux fautes

Cette section présente quelques mecanismes particulièrement utiles pour la mise en oeuvre de la tolérance aux fautes dans un système réparti.

Détecteur de défaillance

Un système tolérant aux fautes comprend un mecanisme chargé de détecter les défaillances des processus et des canaux. La notion de détecteur de défaillance fut formalisé par Chandra et Toueg [CTO96] dans le contexte de faute de crash.

Approche synchrone

Dans un système synchrone, les défaillances sont détectées à l'aide d'un délai de garde. Lorsqu'un processus P1 envoie un message m à un processus P2, le processus P1 s'attend à recevoir, de la part de P2, une confirmation de la réception du message m, avant l'écoulement d'un certain délai précis. Si P1 ne reçoit pas cette confirmation alors que le délai est écoulé, une défaillance temporelle est survenue.

Approche asynchrone

Dans un système asynchrone, il n'est pas possible de détecter les défaillances avec un délai de garde. Un détecteur de défaillance est chargé d'informer les processus corrects(correct process) des défaillances de leurs pairs. Les canaux sont supposés fiables. Cette hypothèse revient à considerer que les canaux peuvent perdrent des messages (défaillance par omission) mais qu'un nombre de retransmission du message suffisent à le faire parvenir à destination.
Briévement un détecteur de défaillance(faillure detector) est un service réparti sur tous les noeuds du système. Chaque module du détecteur de défaillance FDi(faillure detector module) est lié à un processus Pi .
Sur chaque noeud, un module du détecteur de défaillance informe les processus corrects s'exécutant sur ce noeud, de la liste des processus du système qu'il suspecte être défaillants. Chaque module du détecteur de défaillance peut faire des suspicions érronées. Un processus suspecté par un module à un instant donné comme incorect peut être vu par un autre module comme correct. ces erreurs ne doivent en aucun cas perturber le bon fonctionnement des processus corrects du systèmes . Les auteurs de [CTO96] ont défini des propriétés permetant de caractériser le comportement des détecteurs de défaillances, ces proprietés sont des pro-

prietés de complétude et de justesse. La compétude décrit l'aptitude du détecteur à finalement suspecter tous les processus défaillants, tandis que la jutesse restreint le nombre de suspicions erronés.

- **compétude**(completeness)
  1. complétude **forte**(strong completeness) : il esxiste un instant à partir duquel tout processus défaillant est définitivement suspecté par tout processus correct ;

  2. complétude **faible**(weak completeness) : il exite un instant à partir duquel tout processus défaillant est définitivement suspecté par au moins un processus correct ;

- **justesse**(accuracy)
  1. justesse **forte**(strong accuracy) : aucun processus correct n'est jamais suspecté ;

  2. justesse **faible**(weak accuracy) : il existe au moins un processus correct qui n'est jamais suspecté ;

  3. justesse **finalement forte**(eventual strong accuracy) : il existe un instant à partir duquel tout processus correct n'est suspecté par aucun processus correct ;

  4. justesse **finalement faible**(eventual weak accuracy) : il existe un instant à partir duquel au moins un processus correct n'est suspecté par aucun processus correct.

Ces proprietés permetent de définir huit classes de détecteurs de défaillance qui sont résumés par la figure 2.4. Si un détecteur satisfait la complétude forte et la justesse forte alors il appartient á la classe des détecteurs parfaits, cette classe notée P représente le détecteur idéal qui détecte toutes les défaillances(complétude forte) et qui ne fait aucune suspicion erronée(justesse forte). La notion de détecteur de défaillance peut être utilisée aussi bien dans un système synchrone que

complétude	**justesse**			
	forte	faible	finalement forte	finalement faible
forte	parfait P	fort S	finalement parfait ◇ P	finalement fort ◇ S
faible	quasiment parfait Q	faible W	finalement quasiment parfait ◇ Q	finalement faible ◇ W

FIG. 2.4: Huits classes de détecteurs de défaillances

dans un système asynchrone.

Groupe de processus

Un **groupe de processus**(process group) est un ensemble de processus coopérant pour atteindre un objectif commun. Un groupe est désigné par un nom unique et indépendant de la composition du groupe. A tout moment un processus peut quitter ou rejoindre le groupe. En particulier lorsqu'un processus appartenant à un groupe se termine, il quitte le groupe. A tout moment les membres d'un groupe g ont tous la même vue de la composition du groupe. La **vue**(group view) à l'instant logique i, notée vi(g), est la liste des processus appartenant à g, à l'instant i. Tous les membres du groupe g reçoivent la même séquence de vue. Cette sémantique est assurée par le **sevive de gestion de groupe**(group membership group), un service réparti qui s'appuie sur les informations fournies par le détecteur de défaillances. Comme ce dernier le service de gestion de groupe est constitué par des modules, localisé chacun sur un noeud lorsqu'un module détecte un évenement susceptible de provoquer un changement de vue (par ex . un processus desirant rejoindre le groupe, un membre du groupe est suspecté), il communique avec ses pairs de façon à établir un concensus sur la vue du groupe.

Envoi d'un message à un groupe de processus

La notion de groupe de processus s'avère utile pour envoyer un message à un groupe de processus. Le nom de groupe permet d'adresser un message à un groupe sans connaitre l'identité et la localisation de chaque destinnataire. Une pri-

mitive réalisant l'envoi d'un message à ungroupe est appelé **multicast**(multicast). On distingue un multicast fiable(reliable multicast) qui garantit qu'un message m envoyé à un groupe g est, reçu par tous les membres corrects du groupe, ou par aucun. Un multicast vue synchrone(view-synchronous multicast) garantit la propriété suivante : Soit les processus appartenant à vi(g), si p a délivré m dans vi(g) avant d'installer vi+1(g), alors tous les processus appartenant à vi(g) ayant installé vi+1(g) ont délivré m avant d'installer vi+1(g). La sémantique vue synchronne peut être augmentée avec un ordre de délivrance des messages. cet ordre est FIFO[2].

- ordre **FIFO** : si un processus p envoie un message m au groupe g avant d'envoyer un message m' au même groupe, alors tous les processus de g ne délivrent m'qu'après avoir délivré m.

- ordre **causal** : si l'envoi d'un message m à un groupe g, précède causalement l'envoi d'un message m' à g, alors tous les processus de g ne délivrent m' qu'après avoir délivré m. On dit qu'un évenement e précède un evenement e'(noté e -> e') si et seulement si :

  1. un processus exécute e puis e', ou
  2. e correspond à l'envoi d'un message m et e' à la délivrance de m, ou
  3. il existe un évenement e" tel que e ->e" et e" -> e' ;

- ordre **total** : si un processus p et un processus q appartenant à un groupe g, délivrent les messages m et m', alors p et q délivrent m et m' dans le même ordre.

2.5 Conclusion

La tolérance aux fautes est un moyen d'assurer la sûreté de fonctionnement d'un système informatique. Ce moyen consiste à faire en sorte que le système délivre un service correct malgrè l'ocurrence des fautes. Une faute est une anomalie affectant le matériel ou le logociel, susceptible de provoquer la défaillance du système. Un système est défaillant lorsqu'il ne délivre plus un service conforme à sa

[2] firts in firts out

spécification.
Un système réparti est un système informatique dont les composants logiciels s’exécutent sur des ordinateurs(ou noeuds) interconnectés par un réseau.

Chapitre 3

La Replication dans les systèmes répartis

3.1 Introduction

Les bases de données réparties et la replication des données sont reconnues aujoud'hui comme moyens efficaces pour augmenter la disponibilité et la fiabilié des bases de données. De plus la replication peut contribuer favorablement à l'amelioration des performances en utilisant les copies locales voire les copies plus proches.

Toutefois, ces avantages sont contraints par un problème majeur de cohérence mutuelle des copies. La gestion des copies en terme de propagation des mises à jour est ainsi nécessaire. La charge induite peut entraîner un impact significatif sur le système. Celle-ci ne doit pas altérer de façon excessive le temps de réponse global. En d'autres termes il s'agit de garantir la cohérence mutuelle dans les délais acceptables.

De plus, on peut noter que le temps nécessaire, pour qu'une mise à jour prenne effet sur toutes les copies, peut varier selon les methodes de gestion. Les copies peuvent ainsi présenter un décalage les unes par rapport aux autres. Ce retard, appelé **temps de latence**, définit la periode ou une donnée peut être utilisée(lecture) alors qu'elle ne reflète pas toutes les modifications antérieures de la base de données.

L'importance de la durée moyenne de ce temps de latence varie d'une application à l'autre. Certaines applications peuvent tolérer un temps de latence de quelques minutes, alors que d'autres exigent un temps de latence d'environ quelques secondes voire millisecondes.

Dans ce chapitre nous verrons comment la replication est utiliée dans les systèmes répartis pour mettre en oeuvre la tolérence aux fautes. Il rappelle quelques résultats dans la littérature concernant la replication.

3.2 Stratégies de replication

Cette section présente la notion de stratégie de replication. Ces stratégies visent à garantir une **cohérence forte**(strong consistency) entre les copies d'un objet repliqué. Informellement ceci revient à assurer que l'état de chaque copie soit identique. La replication passive et la replication active sont deux stratégis de références.

3.3 Replication active

La replication active(active replication ou state machine approach)se définie [WPS99] par la symétrie des comportements des copies d'un composant repliqué. Chaque copie joue un rôle identique à celui des autres.

Principe

La replication active est définie ainsi [WPS99] :

- réception des requêttes : toutes les copies reçoivent la même sequence [1]
- traitement des requêtes : toutes les copies traitent les requêtes de manière déterministe [2]
- émission des reponses : toutes les copies émettent la même séquence de réponses.

[1] Une séquence de requêtes est un ensemble totalement ordonné. Les copies reçoivent les mêmes requêtes dans le même ordre.

[2] Un traitement déterministe produit toujours le même résultat, par conséquent pour une requête donnée, toutes les copies produisent le même résultat.

La figure 3.1 illustre ce principe à l'aide d'un diagramme temporel.
Les flèches horizontales représentent l'exécution de quatre composants : Client,S1,S2,S3.
Les Si sont les copies du composant repliqués S. Elles appartiennent à un même
groupe. Lorsque le client invoque S, il envoie une requête(traits en flèeche continu)
à tous les Si à l'aide d'un ABCAST assurant l'ordre total et la propriété d'atomi-
cité. Chaque Si traite la requête(petit rectangle horizontal) et renvoie une réponse
au client.

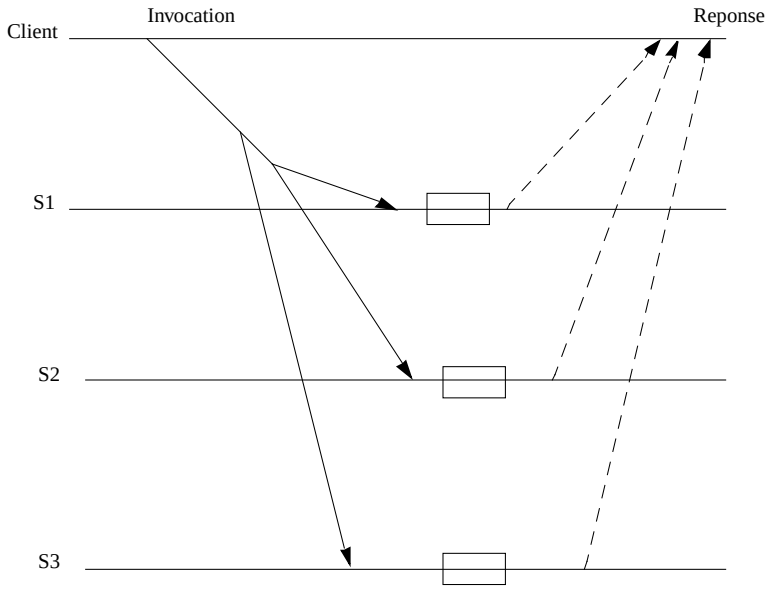


FIG. 3.1: Principe de la replication active

3.3.1 Tolérance aux fautes

La replication active [SOL99] où chaque copie est repliquée et exécutée simultanément sur n machines distinctes. Les repliques doivent synchroniser leurs exécutions à chaque fois qu'un message est reçu ou envoyé afin d'assurer que toute replique réalisant un même calcul reçoive les messages provenant des autres processus dans le même ordre. Plusieurs modes de défaillance d'une replique existent, allant de la défaillance par arrêt(défaillance la plus simple) à la défaillance byzantine(défaillance la plus complexe). Dans le cas de défaillance par arrêt, une replique défaillante ne produit plus aucun résultat. Dans le cas de défaillances byzantine, une replique défaillante continue de produire les résultats mais ceux-ci sont erronés. La tolérance aux fautes est assurée par masquage d'erreur. La défaillance d'une copie est masquée par le comportement des copies non défaillantes. Comme chaque copie joue un rôle identique. La défaillance de l'une d'entre elle ne perturbe pas le service fourni par le composant .

3.4 Replication passive

La replication **passive** distingue deux comportements d'un composant repliqué : la copie **primaire**(primary copy) et les copies **secondaires**(backups). La copie primaire est la seule à effectuer tous les traitements. Les copies secondaires, oasives, surveillent la copie primaire. En cas de défaillance de la copie primaire, une copie secondaire devient la nouvelle copie primaire.

Principe

La replication passive est définie ainsi [WPS99] :

- réception des requêtes : la copie primaire est la seule à recevoir les requêtes ;
- traitement des requêtes : la copie primaire est la seule à traiter les requêtes ;
- émission des réponses : la copie primaire est la seule à émettre les reponses ;

La figure 3.2 illustre ce principe. Le client envoie la requête uniquement à la copie primaire S1. Celle-ci traite la requête, construit un point de reprise et l'envoie à l'aide d'un multicast fiable assurant l'ordre FIFO, aux copies secondaires S2 et S3 en précisant le changement de son état(message update). Après la mise à jour de leurs états, les copies secondaire envoie un ack à la copie primaire. La copie primaire envoie la réponse au client après réception des "ack" de toutes les copies sécondaires. Le point de reprise permet de synchroniser l'état des copies secondaires avec celui de la copie primaire puisque celle-ci est la seule qui communique avec le reste du système.

3.4.1 Tolérance aux fautes

La replication passive où un composant logiciel est repliqué en n exemplaires, mais une seule des n repliques éffectue le calcul. Les n-1 autres repliques sont passives et ne prenent la relève que si la replique active est défaillante. Pour que cette stratégie fonctionne, il est nécessaire que la replique active transmette aux repliques passives, à intervalles réguliers son état d'exécution. Cet état d'exécution composé d'une pile, de données et de registres est stocké par chacune des repliques passives et constitue un **point de reprise**. Si la replique active est défaillante, une des repliques passive est activée et reprend l'exécution du calcul à

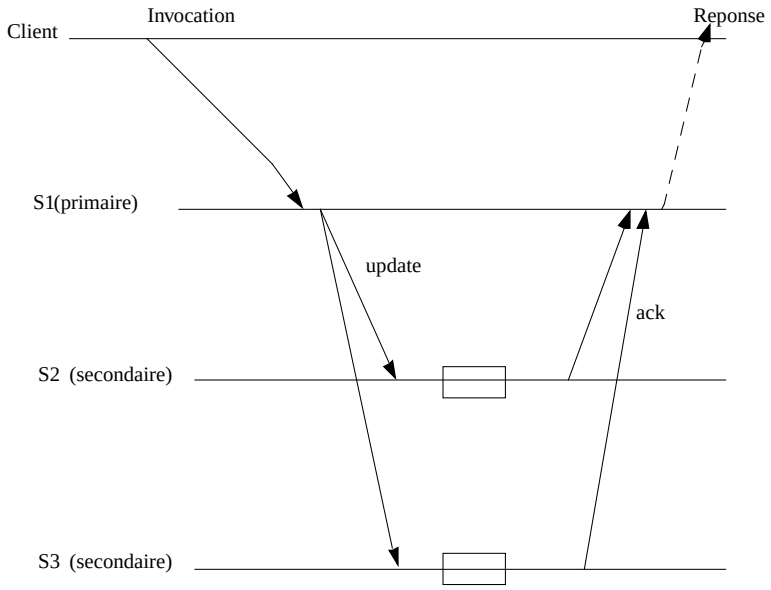


FIG. 3.2: Principe de la replication passive

partir du dernier point de reprise enregistré. On dit que le processus éffectue un retour arrière. Recréer un état d'exécution simule une remontée dans le temps en ramenant le calcul dans l'état qu'il occupait avant la manifestation de la défaillance. La tolérance aux fautes est realisée par détection et compensation de l'erreur. La défaillance d'une copie sécondaire ne nécessite aucun traitement particulier. Par contre, la défaillance de la copie S1 implique que les copies secondaires designent l'une d'entre elle (par exemple S2) comme la nouvelle copie primaire. Les cas suivants peuvent se presenter.

– La défaillance de la copie primaire avant l'envoie du message "update" a pour conséquence que le client n'obtienne aucune réponse à la requête. Tout le traitement de la requête éffectué par la copie primaire S1 est perdue. Le client doit alors reémettre sa requête en l'adressant à la nouvelle copie primaire S2. Cette dernière doit être en mesure de construire la réponse que le client attend.

– Deux situations peuvent se présenter selon que la défaillance de S1 ait été détectée pendant l'invoie du message "update" aux copies sécondaires ou

avant l'envoie de la reponse au client. Ceci est le cas le plus diffile à apprehender :

1. L'Atomicité doit être guarantie : le message " update" doit être reçu par tous ou par aucune des copies sécondaires.

2. Le client doit reémettre sa requête au nouveau primaire.

– La défaillence du primaire S1 a lieu après avoir envoyé la réponse. Dans ce cas un nouveau primaire doit être designé.

Pour la désignation du primaire et l'assurance de l'atomicité, nous renvoyons le lecteur à [PUS00].

3.5 Replication semi-active

La replication semi-active(semi-active replication) [WPS+00a] se situe à michemin entre la replication active et la replication passive. Contrairement à la replication passive, les copies sécondaires ne sont pas oasives. La copie primaire est appelée leader et les copies sécondaires sont appelées suiveurs.

Principe

La replication semi-active est défini ainsi [WPS+00a]

– réception des requêtes : toutes les copies reçoivent le même ensemble[3] de requêtes.

– traitement des requêtes : toutes les copies traitent toutes les requêtes. La copie primaire traite une requête dès qu'elle la reçoit . Par contre, une copie sécondaire doit attendre une notification de la copie primaire pour pourvoir traiter une requête ;

[3]Les requêtes ne sont pas totalement ordonnés.

– émission des réponses : la copie primaire est la seule à émettre les réponses.

La figure 3.3 illustre ce principe. Le client envoie une requête à tous les Si. La copie primaire S1 envoie une notification nitify aux copies sécondaires et commencent le traitement de la requête. Les copies sécondaires S2 et S3 ne commencent à traiter la requête qu'après avoir reçu la notification de la copie primaire. Sitôt le traitement terminé, S1 envoie la réponse au client.

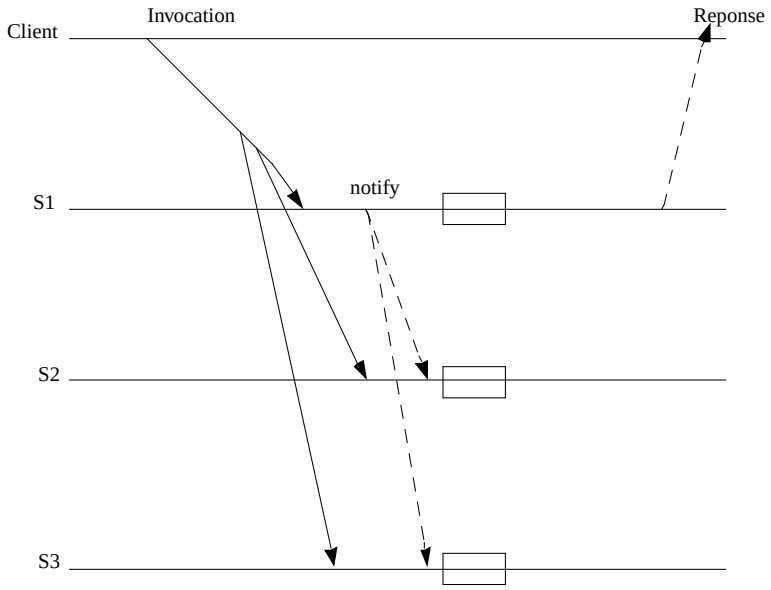


FIG. 3.3: Principe de la replication semi-active

3.6 Tolérance aux fautes

La tolérance aux fautes est réalisée par détection et compensation de l'erreur, comme dans le cas de la replication passive. Cependant comme toutes les copies reçoivent la requête, le client n'a pas besoin de reémettre la requêtte lorsque le primaire S1 défaille. La nouvelle copie primaire S2 envoie automatiquement la réponse au client. Deux situations peuvent se présenter suivant la défaillance du primaire S1 est détectée par les copies sécondaires avant ou après la notification. Si la défaillance de la copie primaire est détectée avant la réception de la notification, la nouvelle copie primaire S2 envoie une notification concernant la première requêtte présente dans sa queue et la traite normalement. Si la défaillance de S1 est détectée après la reception de la notification, S2 traite la requêtte correspondante sans envoyer de notification et envoie la réponse au client.

3.7 Conclusion

La replication est une technique permettant de mettre en oeuvre la tolérance aux fautes dans un système réparti. Repliquer une base de données BD consiste à faire en sorte qu'il existe plusieurs copies BDI de cette base dans le système. Chaque copie BDI est localisée sur un noeud distinct. Si une copie BDi tombe en panne, il existe une probabilité non nulle qu'une autre copie BDj soit toujours opérationnelle.

Commentaires

Ce chapitre a présenté le premier volet du contexte dans lequel s'inscrit ce travail de semestre, à savoir la tolérance aux fautes par replication dans les systèmes répartis. L'objectif de cette présentation était de faire le point sur la notion de replication et son utilisation pour mettre en oeuvre la tolérance aux fautes. Une autre technique de tolérance aux fautes est la durabilité. Ce second volet fera l'objet du prochain chapitre

Chapitre 4

Systèmes transactionnels

4.1 Introduction

Considerons le contexte [BHG87]. Une application cliente se connecte sur un serveur sur lequel s'exécute une base de données. Un client peut soumettre au serveur des opérations de lecture, écriture dans l'ordre de récupérer des données ou de sauvegarder des données. Ces opérations de lecture et d'écriture sont soumises dans le contexte des transactions. Ce modèle peut être vu comme un modèle client-serveur.

4.1.1 Processus client et processus serveur

Dans le système, on peut distinguer deux sortes de processus : processus serveur et processus client. Chaque processus serveur a son propre espace de stockage. Un processus client soumet des opérations au processus serveur en utilisant des primitives de communications. La figure 4.1 illustre un modèle de base.

Données

Sur le disque de chaque serveur, un nombre constant de données sont sauvegardées. La base de donnée peut être vue comme un tableau à une dimenssion contenant un nombre constant de valeurs. Le opérations possibles sont : lecture et écriture. Notation : Dans la plupart des exemples r et w sont utilisés pour spécifier la lecture et l'écriture.

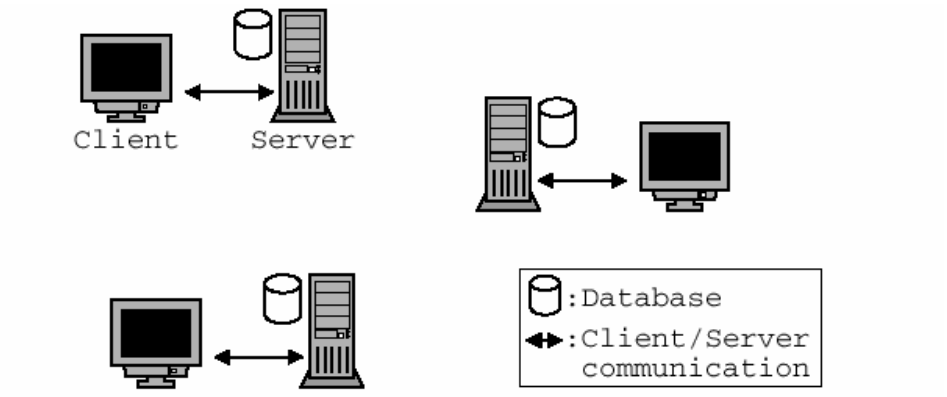


FIG. 4.1: Un modèle client serveur

Objectifs Système transactionnel

Un système transactionnel doit maintenir la cohérence d'un système d'information lors d'accès concurrents ou lorsqu'une défaillance se produit. Par exemple, si une personne désire faire un virement d'un compte A sur un compte B pendant qu'une autre veut consulter l'état des comptes A et B, il peut s'en suivre un résultat inconsistent. En effet, si la consultation se fait après que le compte A ait été débité mais avant que B n'ait été crédité, la deuxième personne n'aura pas une vue consistente de l'état du système. De même, si une défaillance se produit après que A ait été débité mais avant que B ne soit crédité, une incohérence apparaît dans le système d'information.

Transactions

Une transaction [BHG87] est une séquence d'opérations(lectures et écritures) invoquées sur des objets partagés. Le but du contrôle de concurrence est de s'assurer que les transactions s'exécutent de manière atomique[1]. , reprenons l' exemple du retrait d'argent dans un bancomat. La valeur du compte doit être lue, la valeur à rétirer doit être soustraite de ce montant, et la valeur résultante doit être sauvergardée dans le système. Ce retrait consiste en deux opérations read(Account) et write(Account,value), ces opérations vont de paire : soit elles sont exécutées ensemble, soit elles ne le sont pas. Une transaction commence par le mot reservé

[1]Elle doit s'éxecuter intégralement ou pas du tout

begin, suivi d'une suite d'opérations et se termine soit par l'opération abort, soit l'opération commit. Une opération commit signifit qu'une transaction sera soumise au système[2] La figure 4.2 montre un exemple de transaction. Le client qui soumet cette transaction doit le faire de la gauche vers la droite. Cette transaction est constituée de cinq opérations, dont trois opérations accédant aux données. Une transaction est dite de lecture si ses opérations sont constituées que des reads. Deux opérations sont en conflit si elles opèrent sur les mêmes données en provenant des transactions différentes et l'une d'entre elle est une opération de lecture. Deux transactions interfèrent quand l'une ou plusieurs de leurs opérations sont en conflit.

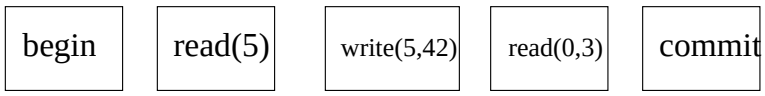


FIG. 4.2: Un exemple de transaction

4.2 Définitions

D'après [RGU00] :

- **Base de Données** : Ensemble de données persistantes.
- **Contraintes d'Intégrité** : Ensemble de règles (Contraintes référentielles, Contraintes de domaines, Dépendances fonctionnelles, etc..)
- **Base de Données Cohérente** : Base de données où toutes les contraintes d'intégrité définies sont vérifiées.

[2]collection de module logicliel et materiel permettant les commandes d'accès aux données

4.3 Propriétés ACID

Le but d'un système d'information est de regrouper ensemble les quatres propriétés suivantes [RGU00] :

- **Atomicité** : C'est le principe du tout ou rien : Toutes les actions sont validées ou aucune n'est validée.

- **Cohérence ou Consistence** : Il s'agit de prendre les données dans un état cohérent et les rendre dans un état cohérent[3]. La figure 4.3 illustre le principe de la cohérence.

- **Isolation** : Les modifications d'une transaction sont invisibles pour les autres transactions.

- **Durabilité** : Les actions d'une transaction ne peuvent être perdues si la transaction est validée.

Quand ces propriétés sont satisfaites, la cohérence du système d'information est assurée malgrè les exécutions concurrentes et les défaillances.
Un système d'information est dit transactionnel s'il garantit les propriétés d'atomicité, d'isolation et de durabilité.

4.4 Histoire et serialisabilité

Une transaction Ti est ordre partiel (Ei,<i) [RGU00] :

- Ei étant le nombre d'opérations de Ti, inclue les invocations commit et abort. L'invocation d'une opération op par la transaction Ti sur un object X est noté OPi[X], l'opération commit est noté Ci et l'opération abort Ai.

- <i est un ordre partiel sur Ei et un ordre total sur tout sous ensemble de Ei concernant le même objet.

[3] un état qui satisfait les invariant du système

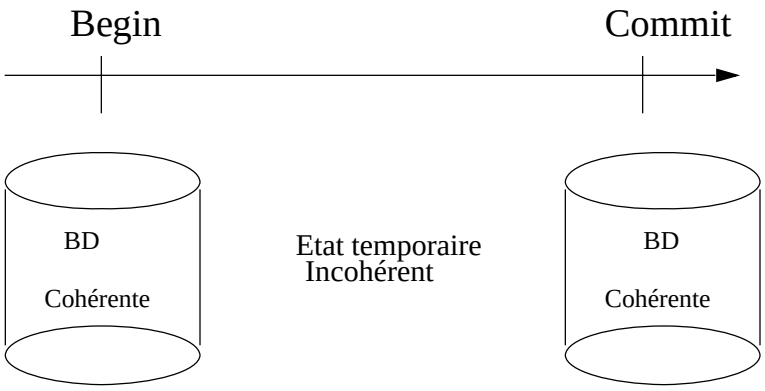


FIG. 4.3: Principe de la cohérence

Une histoire d'un ensemble de transactions est un ordre partiel H=(E, <),

- E est l'ensemble de toutes les opérations des transactions de T.
- < préserve l'ordre <i sur tout Ti de T.
- < est un ordre total sur tout sous ensemble de E concernant le même objet.

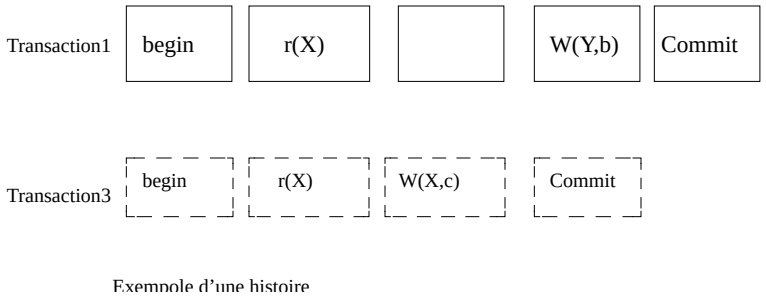
Pour un exemple d'histoire, voir la figure 4.4, nous avons deux transactions de cinq et quatre opérations.

Deux histoires H1 et H2 sont équivallentes si les conditions suivantes sont remplies :

- Elles comportent exactement les mêmes transactions.
- Si une opération de lecture r(X) d'une transaction T de H1, elle le fait également pour la transaction T de H2.
- Si w(X) est la dernière opération ayant écrit sur X dans H1, elle est aussi la dernière opération ayant écrit sur X dans H2.

Une histoire est **sérielle** [BHG87] si pour toute paire de transactions (T1,T2), toutes les opérations de T1 precèdent celles de T2(ie une histoire produite par un système exécutant les transactions séquentiellement). Une histoire est **sérialisable**

si son exécution est équivallente à une exécution séquentielle. Pour une histoire sérialisable voir la figure 4.4, pour une histoire non serialisable , voir la figure 4.5. Elle contient les mêmes transactions que que la figure préceédente, mais avec une légère différence. Le r(X) de la transaction2 est exécuté **entre** le r(X) et le w(X) de la transaction 1. Toute histoire sérialisable faite de transactions consistentes est consistente.



Exempole d'une histoire

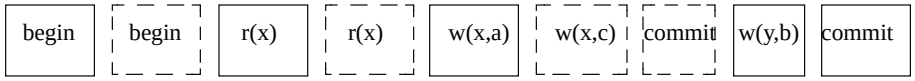


FIG. 4.5: Une histoire non serialisable

4.5 Conclusion

Le chapitre suivant, qui est consacré au Simulateur developpé au LSE, présentera la stratégie utilisée pour assurer la durabilité fera une proposition pour l'assouplir .

Chapitre 5

Le Simulation

5.1 Introduction

Il a fallu un bon moment pour comprendre le fonctionnement du simulateur de pouvoir scerner de manière précise la partie dont il faut apporter des modifications, pour assouplir les règles de la durabilité.

5.2 Description du simulateur

5.2.1 Aspects théoriques

Le simulateur a été développé en C++, tout site est modélisé comme un processeur avec des disques de données et un disque de sauvegarde. Le réseau est modélisé comme une ressource commune partagé par tous les sites. Chaque processeur est partagé par un ensemble de threads qui s'exécutent, un thread de terminaison et un thread générateur de données. Tous les threads ont la même priorité et les ressources sont allouées aux threads par un ordre FIFO. Tout thread s'exécutant, exécute une et une seule transaction et le thread de terminaison est chargé de faire la certification. Les threads de terminaison et d'exécution d'un site partagent les structures de données. Les transactions commitées sont délivrées par le thread de terminaison et sont certifiées. Si une transaction passe le test de certification, ses verrous d'écriture sont relâchés et ses mises à jour sont effectuées. Toute transaction terminée(commit ou abort) est remplacée par une nouvelle. Les transactions terminées par un abort sont renvoyées au générateur de transactions, qui seront resoumises plus tard au même processus d'origine. L'interblocage est détecté par un

mecanisme de timeout(transaction timeout). Un temps est nécessaire à une transaction pour s'exécuter, passé ce temps, les transactions n'ayant pas atteints leurs commits sont tout simplement avortées.

5.2.2 Quelques classes

Il existe plusieurs classes. Celles qui sont nécessaire pour mon projet sont :

- **Network** répresente un réseau abstrait.
- **AbortMessage** répresente un message pour un abort unilatéral.
- **Certifier** Un objet de cette classe peut decider si une transaction peut commiter
- **Constant** répresente toutes les constantes du système.
- **DBserver** répresente un serveur abstrait.
- **DeadLockCheck** utilisé pour détecter les interblocages.
- **Group** répresente un groupe de replicas.
- **GroupFactory** construit et maintient un groupe.
- **LockQueue** répresente une queue de verrous.
- **Member** répresente un membre du groupe.
- **Parameter** répresente l'ensemble des paramètres utilisés pour une simulation.
- **Server** répresente un serveur[1]
- **SimpleLink** répresente une liason dans le reseau.
- **Transaction** contient toutes les informartions et définitions pour les transactions , leurs statuts et les threads ou les processus qui leur sont associés.
- **TransactionList** répresente une liste de transactions, chacune est réferencée par un entier[2]
- **TrxOperation** répresente l'ensemble des opérations d'une transaction.

[1] un serveur est ensemble de cpus et de disques

[2] cet entier est appelé sequence number

5.2.3 Aspects pratiques

La figure 5.1 montre un exemple d'une base de données repliquée. On peut noter :

- Chaque site serveur a une copie entière de la base de données.
- Les updates se font de manière synchrone.
- Les transactions sont des séquences de reads et de writes suivies d'un commit ou d'un abort.
- Tolérance aux fautes : le système continue à fonctionner si un serveur crashe.

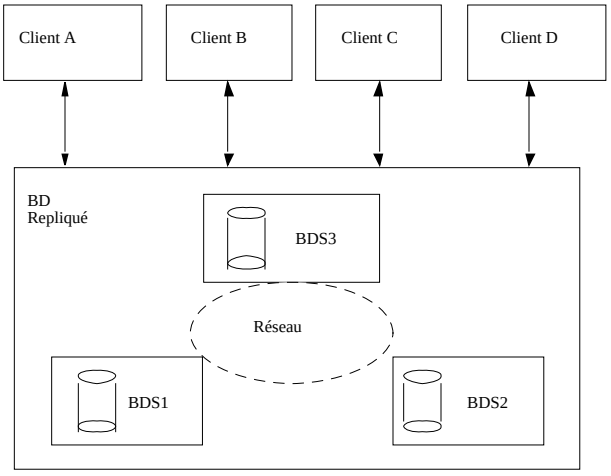


FIG. 5.1: Exemple d'une base de données repliquée

5.3 Assouplir la stratégie de la durabilité dans le simulateur

La gestion de la replication garantit que chaque modification prend effet sur toutes les copies de la base de données. La solution utilise le concept de maître/esclaves. Une transaction d'écriture est éffectuée à partir de la copie du maître. La durabilité quand à elle est assurée, par le fait que lorsqu'une transaction est validée, ses mises à jour ne doivent pas être perdues.

Idée

Dans le cadre de ce projet de semestre, l'idée est de garantir la durabilité par le groupe et non le disque. On n' aura plus besoin de faire des commits. Les updates se feront en mode **asynchrone** en dehors des transactions, pour cela il y'a des contraintes que doit respecter le système.

Principe des mises à jour en mode asynchrone.

Dans le contexte asynchrone, les copies pourront êtres mises à jour suivant différentes approches : On peut envisager de transmettre les données modifiées ou alors de transmettre les transactions de rafraîchissement. Nous nous intéresserons surtout à cette deuxième approche. L'étude de la mise à jour asynchrone montre que deux critères peuvent être utilisées pour la caractérisée.

- A l'initiative de qui faudrait-il ecrire ?
- A quel moment faudrait-il écrire ?

Dans notre contexte, les mises à jour se feront à l'initiative de l'esclave, c'est ce qui est appliqué dans le cadre de ce projet de semestre. Les copies sécondaires ne seront mises à jour que lorsqu'elles seront sollicitées. Par contre, il en decoule une augmentation du temps de latence et du temps de réponse global d'une interrogation, Nous appellerons :

- **Ecriture parallèle** : Une mise à jour pendant la transaction(stratégie actuellement utilisée dans le simulateur).

– **Ecriture séquentielle** : Une mise à jour après validation de la transaction(nouvelle stratégie proposée).

Description de l'Ecriture parallèle

Les mises à jour, la validation ou l'abadon sont exécutés et transmis aussitôt qu'ils prennent effet sur la copie primaire.

La transaction étant considérée comme une suite d'opérations, cette solution consiste à propager immédiatement chaque opération exécutée sur le maître vers les sites distants.

Dès la réception d'un message par un site esclave, un traitement particulier lui est reservé selon qu'il s'agisse d'une opération(lecture ou écriture), d'une validation ou d'un abandon.

1. Dans le cas d'une ecriture : si la données est disponible, l'opération est immédiatement exécutée, sinon elle est stockée dans une file d'attente. Afin de respecter le critère de **sérialisation**, les messages en attente sont traitées selon le principe FIFO.

2. Dans le cas d'un abandon : le traitement consiste à informer de l'échec de la transaction initiale à aboutir. Les opérations exécutées localement sont défaites(rollback).

Cette solution introduit un certain degré de parallélisme entre la modification de la copie primaire et celle des copies secondaires. La mise à jour de la copie primaire et des copies sécondaires sont exécutées simultanément. Une réduction du temps de réponse et du temps de latence peut aussi être obtenue. En fait, le temps de réponse et le temps de latence sont corrélés, car plus tôt une modification prendra effet sur toutes les copies, plus court sera le temps de latence.

Description de l'Ecriture séquentielle

– La mise à jour n'est effectuée que lorsque la transaction est validée sur la copie primaire. Le message transmis est ainsi une transaction identique à la première. La propagation de la transaction de rafraichissement se fait après

validation de la transaction.

– Dans le cas d'un abandon de la transaction initiale, celle-ci est annulée sur le maître, sans que le réseau et les esclaves n'en prennent connaissance.

– Par opposition à la méthode parallèle, cette méthode ne permet aucun degré de parallélisme.

L'utilisation de la méthode parallèle, grâce au parallélisme pourrait contribuer à reduire le temps de réponse. Toutefois, une analyse de ces deux méthodes montre que la méthode séquentielle présente deux avantages qui risquent contrecarrer l'avantage du parallélisme.

1. Une meilleure gestion des ressources : la méthode séquentielle ne transmet sur le réseau que les transactions de rafraîchissement qui sont sûres d'aboutir. La méthode parallèle, quant à elle, transmet toutes les opérations au risque d'envoyer une annulation par la suite en cas d'abandon.

2. Une charge réseau plus faible : la méthode séquentielle charge moins le reseau en raison de sa mise à jour après validation de la transation.

La problématique telle qu'elle est posée consiste à :

1. Déterminer, parmi ces deux méthodes, celle qui offre le meilleur temps de réponse, et par conséquent le meilleur temps de latence.

2. Evaluer le trafic généré et le taux d'indisponibilité moyen des copies pour chaque méthode.

Autrement dit, il s'agit d'une part de quantifier le surcoût en termes de temps de réponse, de trafic et ou d'indisponibilité pour chacune des méthodes et d'âutre part de verifier si la méthode parallèle peut dans tous les cas l'emporter face à une gestion de ressource ameliorée et une minimisation de la charge du réseau. De plus, si la réponse est négative, il faudra déterminer le seuil[3] à partir duquel(desquels) les performances de L'**Ecriture parallèle** chute en face de

[3]ou les seuils minima et maxima

l'**Ecriture séquentielle**.

Des réponses à ces interrogations ne pourront être apportées qu'après, implémentation de la méthode séquentielle et les résultats de simulation. D'ores et déjà, une proposition peut être faite sur la conception du modèle.

5.3.1 Propositions

Conception du modèle

L'objectif étant d'évaluer, dans un contexte particulier, des critères de performances tels que temps de réponse, durée d'indisponibilité etc... les paramètres à retenir doivent refléter aussi bien l'environnement[4].

Modèle

La structure générale du modèle est :

- collection de Nsites par un réseau de communication.
- un base de données repliqué[5]. Un site est considéré maître et Nc-1 comme esclaves.

La figure 5.2 illustre notre modèle.

Chaque site a deux composants principaux : source , processeur. La source génère les transactions. Le processeur simule l'accès à la copie locale de la base de données lorsqu'elle existe. Le site esclave dispose d'un selecteur qui implantera les stratégies de mise à jour et l'écrivain qui exécutera les mises à jour sur la copie secondaire.

Définitions

- **La source**, dans notre cas c'est le client, le composant responsable de la génération des transactions. Chaque transaction est caractérisée par un type : lecture ou mise à jour : Une transaction est dite de lecture si toutes ses opérations sont des lectures. Elle est dite de mise à jour si toutes les opérations sont des écritures.

[4] ie celui utilisé actuellement dans le simulateur

[5] Nc copies

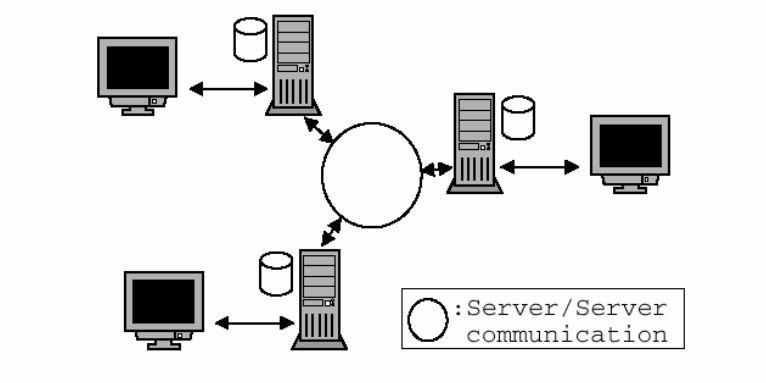


FIG. 5.2: Modèle de communication dans notre système repliqué

- **Le processeur** a pour mission d'exécuter les transactions qui lui sont transmises. Lorsqu'il s'agira d'une opération de mise à jour, le processeur avisera le selecteur qui sélectionnera[6] la méthode à utiliser et avisera l'ecrivain qui sera responsable de faire la mise à jour et de la propager.

- **Le selecteur** sélectionne la stratégie de mise à jour. Pour cela il achemine les opérations une par une dès leur exécution lorsque la méthode parallèle est utilisée. Dans le cas contraire, il garde une copie de la transaction qu'il transmetra aux sites distants via l'ecrtivain, dès qu'il recevra la validation locale.

- **L'écrivain** responsable de l'exécution des ecritures.

Classes concernées dans le simulateur

les classes les plus visées dans le simulateur pour l'implémentation de cette stratégie sont :

- **DBServer** : représente un serveur sur lequel se trouve une base de données.
- **Server** : représente un serveur formé de cpus et de disques.

[6]Ecriture parallèle ou Ecriture séquentielle

Une variable booléenne priority sera définie et declarée dans toutes les instances de la classe DBServer et Server. elle donne ls priorité[7] avec laquelle les opérations d'écriture seront exécutées. Cette variable sera transmise au selecteur pour lui permetre de faire la sélection. La figure 5.3 montre l'exécution d'un ordre d'ecriture par le serveur. on peut la resumer dans les lignes suivantes.

1. : Réception de l'opération write par le serveur.
2. : Exécution d'un thread le selecteur est informé et prend une decision selon la valeur de la variable **priority**.
3. : Priority = 0, la durabilité est garantie dans le simulateur par la methode "**Ecriture parallèle**"
4. : Priority = 1, la durabilité pourra être garantie par la methode "**Ecritue séquentielle**.

5.4 Conclusion

Lors de ce chapitre, la méthode de sauvegarde hors transaction a été proposée. Cette méthode diffère de celle actuelle dans le simulateur par le moment ou il faudrait faire les mises à jour. L'implémentation et l'étude des performances des deux stratégies fera partie de la suite de ce travail.

[7]Une priorité de 0 fera exécuter les opérations d'ecriture à l'interieur des transactions, une priorité de 1 fera exécuter les opérations d'ecriture hors des transactions

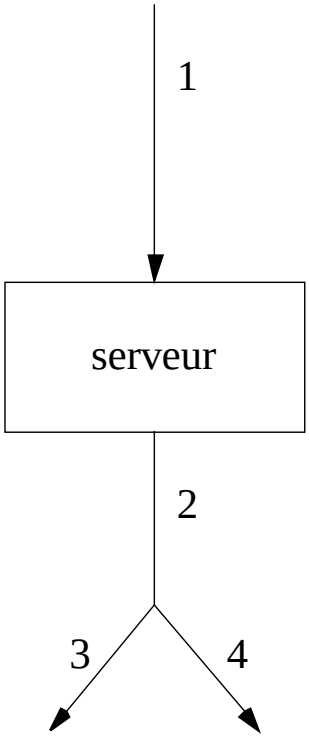


FIG. 5.3: Exécution d’un ordre d’écriture

Chapitre 6

Conclusion

Ce chapitre dresse un bilan de ce travail de semestre et en présente les perspectives.

6.1 Bilan

Lors du présent travail de semestre, j'ai fait une proposition pour assouplir la règle de la durabilité dans le simulateur. Cette deuxième méthode diffère de la première[1] par :

- Les updates pourront se faire de manière asynchrone .
- La tâche du serveur lorsqu'il réçoit un ordre d'écriture sur le disque :
    1. Dans la première, le serveur écrit pendant la transaction.
    2. Dans la deuxième, le serveur écrit hors de la transaction.

Les performances pourront toujours être décrits en terme de :

- temps moyen de réponse des transactions commitées.
- temps de latence du réseau.
- temps d'utilisation des processeurs en secondes.
- trafic généré.

Une implémentation de cette nouvelle stratégie, et l'exécution de quelques scénarios dans le simulateur permettra de repondre à toutes nos questions posées.

[1] celle actuellement appliquée dans le simulateur

6.2 Perspectives

La suite intéressante du travail consiste :

- Implémenter cette nouvelle stratégie dans le simulateur développé au LSE.
- Concevoir des scénarios et les appliquer en utilisant le simulateur.
- Comparer les performances des deux stratégies en terme :

    1. temps moyen de réponse des transactions commitées.
    2. temps de latence du réseau.
    3. temps d'utilisation des processeurs en secondes.
    4. trafic généré.
    5. temps d'indisponibilité des données.

En conclusion ce travail de semestre m'a permis d'acquerir de nouvelles connaissances en général sur les systèmes distribués et en particulier la tolérance aux fautes par la replication et la durabilité. Je me suis familliarisé avec les outils de simulation, je finirai en remerciant mon assistant Matthias Wiesmann du LSE, de sa disponibilité et de son secours pendant ce projet de semestre.

Bibliographie

[BHG87] P. Bernstein , V. Hadzilacos , and N. Goodman. Concurrency Control and Recovery in DatabaSE Systems. Addison-Wesley , 1987.

[CSIM18] ,Mesquite Sofware Inc, Simulation engine (C++ version), Est Braker Lane Austin Texas TX-78759-5321, 1994.

[CVS98] S.BORTZMEYER, CVS :Concurrent Versions System, WML ,02-08-98

[CTO96] T.D Chandra and S. Toueg, Unreliable faillure detectors for reliable distributed systems, journal of the ACM , 1996.

[DCA00] D. Chiadmi,A. Cohen, Gestion en mode Asynchrone de la Duplication dans les Bases de données Réparties ,Université Libre de Bruxelle, Service Télématique et Communication , 2000.

[DEF00] X. DEFAGO,Agreement-Related Problems : From Semi-Passive Replication to Total Ordered Broadcast,Thèse no 2229(2000) departement d'informatique Ecole Polytechnique Federale de Lausanne ,2000.

[HTO93] V. Hadzilacos and S . Toueg . Distributeds systems , chapter 5 : Fault Tolerant Broadcast and Related Problems , Addison-Wesley , 2nd edition.

[HAA99] J.Holliday, D. Agraval, A.El Abbadi, The Performance of Database Replication with Group Multicast, Seminar on Database Replication,1999.

[HDD] ,http ://www.mosx.net/logiciels/docdoc.shtml

[KRM96] K.R.MAZOUNI, Etude de l'invocation entre objets dupliqués dans un système tolérant aux fautes, thèse n0 1578 Departement d'informatique Ecole Polytechnique Fédérale de Lausanne , 1996.

[MRL00] M. Raynal,Systèmes Repartis et Reseaux : Concepts Outils et algorithmes,EYROLLES , 1987.

[PED99] F. Pedone, The Database State Machine and Group Communication Issues, thèse no 2090(1999)Departement d'informatique Ecole Polytechnique Fédérale de Lausanne.

[PUS00] P.Urban and A.Schiper, Fault tolerance in Distributed Systems , February 2000.

[PPV91b] .Powell and Verissimo. Delta4 : A generic architecture for dependable computing, chapter 6 : Distributed Fault-Tolerance, pages 89-123 .Springer-verlag, 1991.

[RVW00] Roel Vandewall, Database Replication Prototype,Masters thesis departement d'informatique Ecole Polytechnique Fédérale de Lausanne , 2000.

[RGU00] R. Guerraoui, Transactional Systems : The Consistency Contract, 2000

[SOL99] SOLIDOR, Tolérance aux fautes,1999.

[TSA95] A.S Tanenbaum : Distributed Operating systems, Prentice Hall Inc.1995

[WPS99] M Wiesmann,F. Pedonne and A .Schipper, A Systematic Classification of Replited Database Protocols based on Atomic Broadcast, In Proceedings of the 3th European Research Seminar on Advances in Distributed Systems(ERSADS99), Madeira Island()Portugal, April 23-28,1999.BROADCAST Esprit WG22455.

[WPS+00a] M.Wiesmann, F.Pedone, A. Schiper, B. Kemme , and G. Alonso. Database replication techniques : a threee parameter classification . In Proceedings of 19th IEEE Symposium on Reliable Distributed Systems(SRDS2000),Nüenberg , Germany , October 2000 . IEEE Computer Society.

[WPS+00b] M.Wiesmann, F.Pedone, A. Schiper, B. Kemme , and G. Alonso.Understanding replication in databases and distributed systems : . In Proceedings of 20th International Conference on Distributed Computing systemss(ICDCS'2000),pages 264-274, Taipei, Taiwan, R.O.C , April 2000 . IEEE Computer Society Los Alamitos California.