République algérienne démocratique et populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
Université A. Mira de Bejaia
Faculté des sciences exactes
Département d'informatique

Mémoire de fin d'étude pour obtention d'un master Professionnel en informatique

Option : administration et sécurité des réseaux

THÈME :

Conception d'une architecture pour la composition des services web byzantins

Réalisé par :
ISMAIL Adel
OUBIRA AMEZIANE

Encadré par :
Mr H. Moumen

Devant le jury composé de :
President: Mr K .KABYL
Examinateur 1 : Mr H kHENOUS
Examinateur 2 : Mr A. BAADACHE

Année 2011/2012

Sommaire

Liste des figures

REMERCIMENTS

On tient tout d'abord à remercier Mr H. MOUMEN pour l'honneur qu'il nous a fait en acceptant de nous encadrer. Ses conseils précieux ont permis une bonne orientation dans la réalisation de ce modeste travail.

On tient également à remercier les membres de jury d'avoir accepté de consacrer leur temps à la lecture et à la correction de ce mémoire.

Nos remerciements les plus vifs vont particulièrement à nos parents, nosfrères et sœurs.

Et enfin, merci à tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

DEDICACES

A tous ceux qui compte pour moi

A. OUBIRA

A toute ma famille

A mes amis

A.ISMAIL

Introduction generale

De nos jours, les entreprises expriment un grand besoin pour échanger des informationset des services. Ceci nécessite des langages communs de communication. Les effortsd'élaboration de ces langages ont donné lieu à un nouveau domaine de recherche connu sousle nom de ≪ business protocols≫. Une technologie émergeante dans ce domaine a permis detracer quelques pistes intéressantes pour la communication entre entreprises. Cettetechnologie est celle de services web.

Les services Web sont un paradigme naissant qui vise à la transposition des architecturespar composant dans le cadre du Web. Un service Web est un composant logiciel qui offre desservices à travers une interface standardisée. La particularité des services Web est l'utilisationde la technologie Internet comme infrastructure pour la communication entre eux. Lesservices web se basent sur un modèle de trois couches : un protocole de communicationpermettant de structurer les messages échanges entre les composants logiciels, unespécification de description des interfaces des services et enfin une spécification depublication et de localisation de services. Actuellement, ce modèle supporte principalementdes composants logiciels présentant des services sous forme d'une collection d'unités detraitement (opérations). Le paradigme des services Web est suffisant pour mettre en place descomposants interopérables et facilement intégrables. Toutefois, certains composants logicielsnécessitent l'exécution d'une interaction plus complexe qui obéit a un protocole spécifique envue de parvenir à la fonctionnalité attendue. Ce type de composant résulte d'une nécessiteconceptuelle de mise en place de services sophistiques ou encore d'une agrégation modulairede plusieurs services simples en services Web plus complexes. La composition des servicespermet donc de créer des applications et des processus à l'aide de services issusd'environnements hétérogènes, quels que soient les détails et les différences de cesenvironnements. La composition des services est bien plus facile à accomplir si les servicesen question sont construits avec des interfaces a granularité forte. Grace a un éventail deservices a granularité force conçus et composes de manière efficace, un expert en processusmétier peut composer de manière productive de nouveaux processus métier et de nouvellesapplications.

Il existe plusieurs travaux dans le domaine de développement et d'interaction de servicesweb mais il reste un grand besoin de simplification et d'automatisation de l'élaboration et dela composition des services. Le problème majeur dans ce domaine est que les services websont faiblement couples. En effet, ces services sont développes avec une indépendance totaledes uns par rapport des autres. En outre, les descriptions actuelles de services web ne sont passuffisantes pour développer des clients qui peuvent inter opérer correctement avec plusieursservices.

Dans ce travail nous allons présenter une étude de la composition des services web en présence de fautes byzantines et proposer des solutions de tolérances a ces dernières

Chapitre I : Généralités

1-Les services web

1.1- Définitions et infrastructures d'un service web

Avant de définir les servies web on doit passer par trois standards majeurs qui ont été conçus pour offrir aux services web une application internet robuste et fiable qui sont : SOAP, WSDL et UDDI

a- Le protocole SOAP :

SOAP est un protocole basé sur XML, indépendant de la plate-forme utilisée, permettant à plusieurs applications différentes de communiquer entre elles via le protocole de transfert HTTP. SOAP définit de quelle manière un document XML complexe peut être converti et expédié en HTML. SOAP est un format standard ; il permet l'échange de messages XML entre des plates-formes différentes, à partir de langages de programmation différents.

b- Le Langage WSDL :

WSDL est un langage de description des services XML basés sur le réseau. Il permet de définir le format des objets, quel que soit le protocole (SOAP, XML) ou le codage (MIME) utilisé par ces objets spécifiques à chaque système.

c- L'annuaire UDDI :

UDDI est une base de données de type XML, permettant de bâtir de véritables annuaires des entreprises e-business, de leurs produits et de leurs services ; UDDI contient également des informations sur les standards les mieux adaptés aux différentes procédures d'affaires.

Dans ces brèvesdéfinitions,on remarque que ces standard utilise tous Le XML, ce qui nous pousse à donner une brèvedéfinition De ce langage.

❖ **Le langage XML :**

XML (*Extensible MarkupLanguage*, ou Langage Extensible de Balisage) est le langage destiné à succéder à HTML. Comme HTML (*HypertextMarkupLanguage)* C'est un langage de balisage (*markup*) : il représente de l'information encadrée par des balises. XML est un métalangage, ce qui veut dire que contrairement à HTML qui possède un ensemble de balises de présentation prédéfinies, Il va permettre d'inventer de nouvelles balises d'isolement d'informations ou d'agrégats élémentaires que peut contenir une page Web.

1.1.1- Définitions

❖ Définition 1 :

Les Services Web sont « des applications modulaires basées sur l'Internet qui exécutent des tâches précises et qui respectent un format spécifique ». Plus clairement, cela consiste à permettre l'utilisation d'une application à distance. En fait, les services web facilitent l'invocation de certains traitements depuis Internet.

L'avantage de ce modèle tient à présenter ces services comme des boîtes noires.
En fait, les entrées-sorties d'un Service Web sont gérées au sein de messages dont on connaît le format grâce à des interfaces clairement exposées mais sur lesquels l'implémentation interne du traitement n'influe pas au niveau de structure. Ceci permet un haut niveau de modularité et d'interopérabilité.

L'avantage du modèle de message est qu'il permet de s'abstraire de l'architecture, du langage ou encore de la plate-forme qui va supporter le service. En fait, la principale Caractéristique des services web est d'être accessible depuis n'importe quelle machine par un programme écrit dans n'importe quel langage. Il suffit juste que le message respecte une structure donnée pour qu'il puisse être utilisé.

- Définition 2 :

Selon la définition du W3C (World Wide Web Consortium), un Web service (ou service Web) est une application appelable via Internet - par une autre application d'un autre site Internet -permettant l'échange de données (de manière textuelle) afin que l'application appelante puisse intégrer le résultat de l'échange à ses propres analyses. Les requêtes et les réponses sont soumises à des standards et normalisées à chacun de leurs échanges.
On trouve les services web de style RPC (RemoteProcedure Call) Et les services web de style Document.

Le style RPC utilise des appels de procédure à distance ; autrement dit, le message SOAP contient une syntaxe d'appel de fonctions, avec laquelle il est également possible de réaliser une surcharge de méthodes (methodoverloading). Les types de données sont alors codés selon les spécifications de SOAP. Le style RPC est le style par défaut dans la plupart des implémentations (à l'exception de .NET).

Le style Document s'appuie sur l'échange de documents XML. Les types de données sont alors définis en totalité selon le schéma XML. Ce style de services web est la variante par défaut des applications .NET.

1.1.2- Le modèle de base des Services web

Le modèle des services Web repose sur une architecture orientée service. Celle-ci fait intervenir trois catégories d'acteurs : les fournisseurs de services (i.e. les entités responsables du service Web), les clients qui servent d'intermédiaires aux utilisateurs de services et les annuaires qui offrent aux fournisseurs la capacité de publier leurs services et aux clients, le moyen de localiser leurs besoins en terme de services. La dynamique entre ces trois acteurs inclut donc les opérations de publication, de recherche et de liens (binding) d'opérations. Cette dynamique est normalisée à travers 3 standards : un protocole abstrait de description et de structuration des messages,
SOAP ,une spécification XML qui permet la publication et localisation des services dans les annuaires, UDDI et un format de description des services Web, WSDL . Un service WSDL est composé d'un ensemble d'opérations élémentaires, chacune décrite par un flux de messages échangés entre le client et le service.

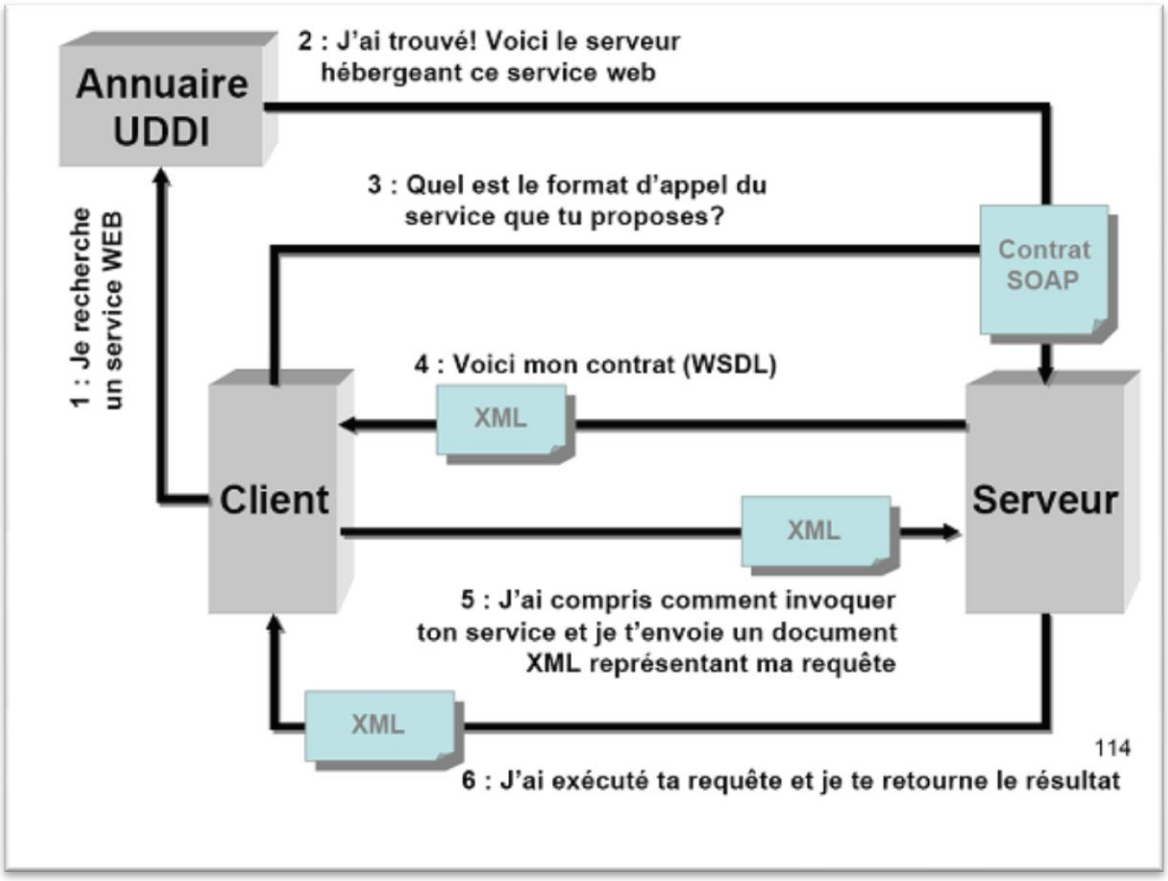

Figure 1: Model de base de service web

1. Le client envoie une requête à l'annuaire de Service pour trouver le service Web dont il a besoin.
2. L'annuaire cherche pour le client, trouve le service Web approprié et renvoie une réponse au client en lui indiquant quel serveur détient ce qu'il recherche.
3. Le client envoie une deuxième requête au serveur pour obtenir le contrat de normalisation de ses données.
4. Le serveur envoie sa réponse sous la forme établie par WSDL en langage XML.
5. Le client peut maintenant rédiger sa requête pour traiter les données dont il a besoin.
6. Le serveur fait les calculs nécessaires suite à la requête du client, et renvoie sa réponse sous la même forme normalisée.

1.2- Les Services Web Composites

1.2.1- Description et fonctionnement

Un service web est dit composé ou composite lorsque son exécution implique des interactions avec d'autres services web afin de faire appel à leurs fonctionnalités. La composition de services web spécifie quels services ont besoin d'être invoqués, dans quel ordre et comment gérer les conditions d'exception.

La composition des Services Web peut se faire de deux manières: orchestration et chorégraphie

I) Orchestration

L'orchestration décrit l'interaction des services au niveau de messages, incluant le logique métier et l'ordre d'exécution des interactions. Les services web n'ont pas de connaissance (et n'ont pas besoin de l'avoir) d´être mêlées dans une composition et d'être partie d'un processus métier. Seulement le coordinateur de l'orchestration a besoin de cette connaissance.

La figure suivante montre le workflow dans l'orchestration des services web. Un coordinateur prend le control de tous les services web impliqués et coordonne l'exécution des différentes opérations des services web qui participent dans le processus.

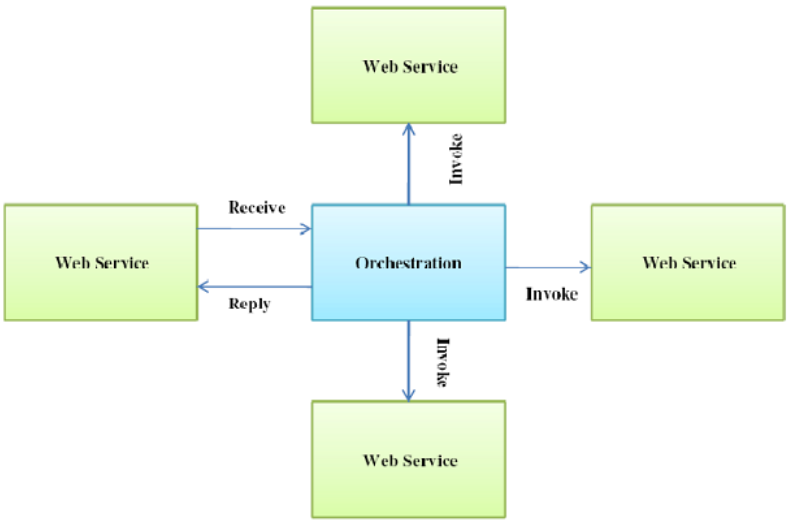

Figure 2:Orchestration des services web

II) Chorégraphie

Contrairement à l'orchestration, la chorégraphie n'a pas un coordinateur central.
Chaque service web mêlée dans la chorégraphie connaît exactement quand ses opérations doivent être exécutées et avec qui l'interaction doit avoir lieu.

La chorégraphie est un effort de collaboration dans lequel chaque participant du processus décrit l'itération qui l'appartient. Elle trace la séquence des messages qui peut impliquer plusieurs Services Web.

La collaboration dans la chorégraphie des services web peut être représentée de la manière suivante:

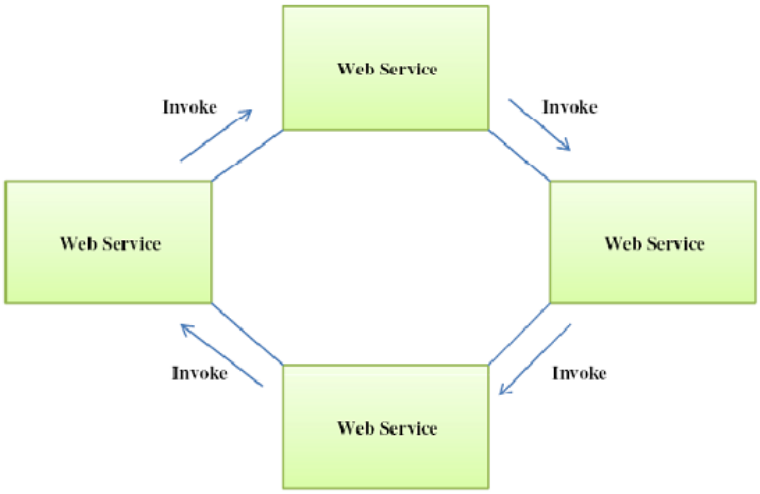

Figure 3:Chorégraphie des services web

Depuis la perspective de la composition des services web, l'orchestration est un approchement plus flexible que la chorégraphie:

- Le responsable ou coordinateur de tout le processus métier est connu.
- Les services web peuvent être incorporés sans soucis, parce qu'ils n'ont pas conscience d'appartenir à un processus métier.

1.2.2 Architecture générale des services web composites :

La figure suivante représente une structure générale pour la composition des services web.Elle a pour but de donner une abstraction de haut niveau des différentes méthodes de composition.

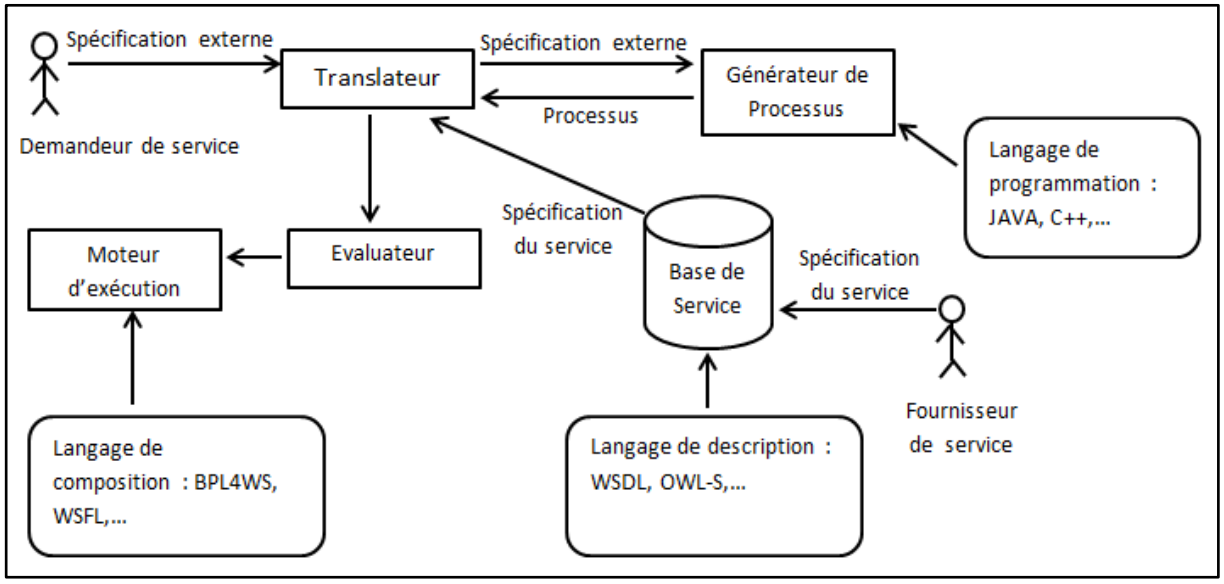

Figure 4: Architecture générale des services web composites

1.2.3-Types de Composition de services web :

Il existe plusieurs manières de classifier la composition des services web nous citons ci-dessous quelques une :

1.2.3.1-Classification basée sur la disponibilité du service composite :

- **Composition proactive :** le service n'est composé et compilé qu'à la demande du client. Ce type de composition est utilisé lorsque les composants ne sont pas stables ou le nombre de clients est petit.
- **Composition réactive :** le service composite est composé et compilé avant la demande du client. Ce type de composition est assigné au pour les applications stables.

1.2.3.2-Classification par le degré d'automatisation:

La composition peut s'accomplir depuis la perspective du dégrée d'automatisation. Elle peut être classifiée dans trois catégories :

I) Composition manuelle

La composition manuelle des services Web suppose que l'utilisateur génère la composition à la main via un éditeur de texte et sans l'aide d'outils dédiés.

II) Composition semi-automatique

Les techniques de composition semi-automatiques sont un pas en avant en comparaison avec la composition manuelle, dans le sens qu'ils font des suggestions sémantiques pour aider à la sélection des services Web dans le processus de composition.

III) Composition automatique La composition totalement automatisée prend en charge tout le processus de composition et le réalise automatiquement, sans qu'aucune intervention de l'utilisateur ne soit requise.

1.2.3.3-Classification basée sur la coordination :

- **Composition séquentielle :** l'exécution de chaque composant ne peut commencer qu'après que les services précèdent auront terminés leurs exécutions. Dans ce cas les flux de données et de contrôle se présentent sous une forme linéaire.
- **Composition concurrentielle :** dans ce cas on auraplusieurs services actifs qui exécutent en concurrence simultanément.

2- Les fautes

2.1- Définition d'une faute

La faute dans un système informatique représente soit un défaut d'un composant physique, ou soit un défaut d'un composant logiciel de ce système. Elle peut être crée de manière intentionnelle ou accidentelle, à cause des phémomènes physiques ou à cause des imperfections humaines. Durant l'exécutiondu système, la faute reste dormante jusqu'à ce qu'un événement intentionnel ou accidentel provoque son activation.

2.2-Classification des fautes

Les fautes et leurs sources sont extrêmement diverses. Les trois points de vue principaux selon lesquels elles peuvent être classées sont leur nature, leur origine et leur persistance. Ceci conduit à classer les fautes selon l'arbre suivant :

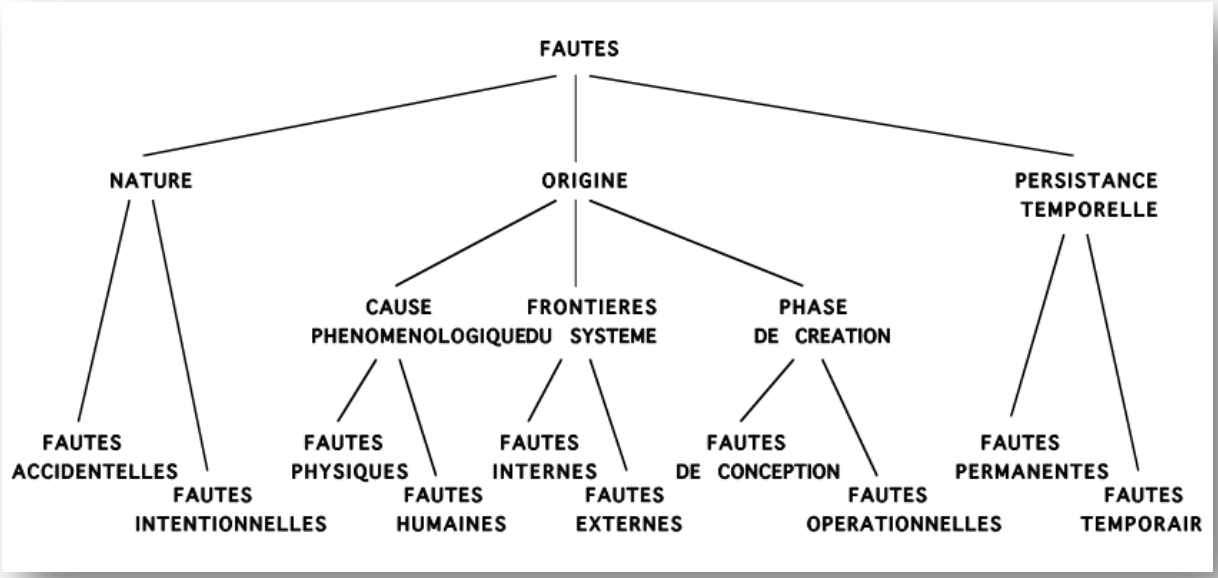

Figure 5:classification des fautes.

Suivant la persistance temporelle d'une faute, on distingue quatre types de fautes qui peuvent affecter un système :

- pannes franches.
- pannes transitoires.
- Pannes par omissions.
- Fautes byzantines.

2.2.1- Pannes franches

Une fois l'entité en panne franche il cesse immédiatement et de façon indéfinie de répondre à toute sollicitation ou de générer de nouvelles requêtes (jusqu'à une réparation).
Une panne franche est une panne permanente.

2.2.2- Pannes transitoires

L'entité cesse de fonctionner temporairement

2.2.3- Pannes par omission

A cause des liens de communications les messages sont perdus.

2.2.4- Fautes byzantines

Les entités fautives peuvent adopter un comportement arbitraire

3- Tolérance aux fautes

3.1- Définition de la tolérance aux fautes

La tolérance aux fautes est l'aptitude d'un système informatique à accomplir sa fonction malgré la présence ou l'occurrence de fautes, qu'il s'agisse de dégradations physiques du matériel, de défauts logiciels, d'attaques malveillantes, d'erreurs d'interaction homme-machine. Elle apparait comme un moyen de garantir une sureté de fonctionnement.

3.2- objectif de la tolérance aux fautes

L'objectif de la tolérance aux fautes est d'éviter les défaillances du système malgré la présence de fautes. La tolérance aux fautes est mise en œuvre par la détection d'erreur et le rétablissement du système.

3.2.1- Détection d'erreur

La détection d'erreur peut être réalisée lors d'une suspension de service. On dit alors qu'elle est préemptive. À l'opposé, on dit qu'elle est concomitante lorsqu'elle est réalisée lors de l'exécution normale du service.

Les techniques de détection concomitante utilisent la redondance au niveau information ou composant, ou la redondance temporelle ou algorithmique. Les formes les plus utilisées sont les suivantes.

• Les codes détecteurs d'erreur : ils introduisent une redondance dans la représentation de l'information.

• Le doublement et la comparaison : les unités de traitement sont dupliquées et leurs résultats sont comparés.

• Les contrôles temporels et d'exécution : un « chien de garde » (watchdog) contrôle les temps de réponse ou l'avancée de l'exécution.

• Les contrôles de vraisemblance ou de données structurées : des assertions sont insérées dans le code pour vérifier des types, des indices, des valeurs, etc.

3.2.2- Rétablissement du système

Le rétablissement du système vise à transformer l'état erroné en un état exempt d'erreur et de faute. Le traitement de la faute se fait en identifiant le composant fautif et en l'excluant. Le traitement de l'erreur peut se faire par trois techniques : la reprise, la poursuite et la compensation.

La reprise est la technique la plus couramment utilisée. L'état du système est sauvegardé régulièrement. Lorsqu'une erreur est détectée, le système est ramené à un état antérieur à l'occurrence de l'erreur. Cet état sauvegardé est appelé point de reprise.

La poursuite consiste à rechercher un nouvel état exempt d'erreur. Ceci peut par exemple être réalisé en associant un traitement exceptionnel lorsqu'une erreur est détectée. Le but de ce traitement est alors de corriger l'état erroné.

La compensation nécessite que l'état du système comporte suffisamment de redondance pour permettre sa transformation en un état exempt d'erreur. Elle est transparente
vis-à-vis de l'application car elle ne nécessite pas de réexécuter une partie de l'application (reprise), ni d'exécuter une procédure dédiée (poursuite). Elle peut par exemple être réalisée en

répliquant des composants et en effectuant un vote majoritaire sur les résultats. Une autre manière de procéder est d'utiliser les codes correcteurs d'erreurs ou plus généralement des algorithmes tolérants aux fautes.

On peut noter que la méthode de compensation ne nécessite pas de détection d'erreurs spécifique puisqu'elle effectue elle-même la détection d'erreur. Une méthode de compensation peut servir de détecteur d'erreur, tandis que l'inverse n'est pas vrai. En effet, la compensation nécessite une redondance plus importante pour pouvoir corriger l'erreur. Par exemple en termes de composants, deux composants suffisent à détecter une erreur, mais au moins trois seront nécessaires pour la corriger.

Chapitre II : Le problème des généraux byzantins

1. Introduction

Le problème des généraux Byzantins est une représentation abstraite d' une classe de programmes mettant en œuvre plusieurs intervenants comme des processeurs dans un ordinateur, des ordinateurs dans un réseau ou des robots dans une usine.

La tolérance aux pannes trouve ses origines dans les besoins militaires liés à la guerre froide : continuité des communications sur un réseau maillé (ARPA). L' explosion de l' informatique dans le monde industriel a soulevé d' autres questions comme la fiabilité des systèmes embarqués dans les avions et les fusées, la continuité de fonctionnement des outils de production ou la liaison des systèmes interconnectés.

Dans ce chapitre Nous allons détailler les Fautes byzantines et étudier les solutions possibles pour résoudre ces dernières.

2. Le problème des généraux byzantins

2.1 Présentation générale

Le problème de composants défectueux dans un système informatique peut être exprimé de façon abstraite en termes de généraux de l'armée Byzantine qui campent autour d'une cité ennemie. A l' aide de messagers, ils doivent se mettre d' accord sur un plan de bataille commun, sinon la défaite est assurée.

Mais certains généraux, des traîtres, essayent de semer la confusion parmi les autres. Le problème est donc de trouver un algorithme pour s' assurer que les généraux loyaux arrivent tout de même à se mettre d' accord sur un plan de bataille .Ils doivent trouver un « consensus ».

2.2. Analogie avec les systèmes informatiques

Dans cette description, un général est un composant informatique (un processeur ou un ordinateur), un messager est un canal de communication bidirectionnelle entre deux composants (Bus, câble Ethernet, réseau téléphonique,…) et un traître est un composant défectueux.

Si un canal de communication entre deux composants est défectueux, nous pouvons considérer que le composant émetteur est un traître. De cette manière, nous pouvons simplifier le problème en raisonnant uniquement sur les composants.

Un plan d'action, dans le cadre d' une gestion de pannes, peut se représenter comme le résultat d' une fonction mathématique (Ex : calcul de la trajectoire d' une fusée). L'évaluation de la fonction est réalisée par plusieurs calculateurs. Le plan d'action peut être le résultat à la majorité des calculateurs ou la moyenne des résultats obtenus. La victoire est le résultat attendu et la défaite est un mauvais résultat.

2.3. La problématique

- Plusieurs divisions de l'armée Byzantine campent autour de la cité ennemie, chacune d'entre elle étant dirigée par son propre général.
- Ils communiquent avec des messagers.

- Après avoir observé l'ennemi, ils doivent se mettre d'accord sur un plan d'action commun. Le problème est que certains de ces généraux peuvent être des traîtres, qui tentent d'empêcher les généraux loyaux de se mettre d'accord.

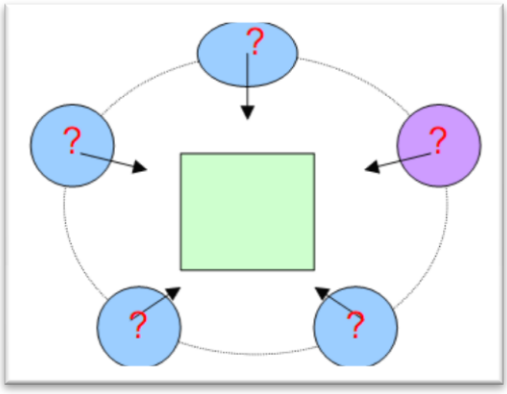

Figure 6: Problème des généraux byzantins

2.4 L'algorithme des généraux byzantins

Les généraux doivent disposer d'un algorithme pour garantir que :

A. Tous les généraux loyaux se mettent d'accord sur le même plan d'action.

Les généraux loyaux feront tous ce que l'algorithme leur a dits de faire, mais les traîtres peuvent faire ce qu'ils veulent. L'algorithme doit donc garantir la condition A, et ce, sans se préoccuper de ce que les traîtres choisissent de faire.

Les généraux loyaux ne doivent pas seulement trouver un accord, mais aussi trouver un plan raisonnable.

B. **Un petit nombre de traître ne peut pas faire que les généraux loyaux choisissent un mauvais plan**.

Chaque général observe l'ennemi et communique ces observations aux autres. Soit *v(i)* l'information communiqué par le $i^{ème}$ général. Chaque général doit donc utiliser une méthode pour combiner les valeurs *v*(1),..., *v*(*n*) en un plan d'action unique, avec *n* le nombre de généraux. La condition A peut être obtenue si tous les généraux utilisent la même méthode pour combiner les informations, et la condition B peut être obtenue en utilisant une méthode "robuste ". Par exemple, si la décision qui doit être prise est soit *Attaque* ou *Retraite*, alors *v(i)* peut être la décision du général *i* et la décision finale peut être basée sur la majorité de ces décisions. Des traîtres peuvent modifier cette décision seulement si les généraux loyaux étaient divisés de manière égale quant à la décision à prendre, auquel cas aucune des décisions ne peut être considérée comme mauvaise.

Bien que cette approche puissent ne pas être la seule façon de satisfaire les conditions A et B, c'est la seule connue. Elle suppose qu'il existe une méthode qui permette aux généraux de communiquer leurs valeurs *v(i)* aux autres. Une méthode évidente est, pour le $i^{ème}$ général, d 'envoyer *v(i)* par messager aux autres généraux. Malheureusement, cela ne fonctionne pas car pour que la condition A soit satisfaite, il faut que tous les généraux obtiennent le même ensemble de messages *v(1),..., v(n)*, et un traître peut très bien envoyer des messages différents

à chacun des autres généraux. Pour que la condition A soit satisfaite, il faut que la condition suivante soit vraie :

1. **Tous les généraux loyaux doivent obtenir les mêmes informations *v(1),..., v(n)*.**

   Cette condition implique qu'un général loyal, ne va pas forcément utiliser la valeur *v*(*i*) reçue de i, puisque si le $i^{ème}$ général est un traître, il a très bien pu envoyer des valeurs différentes à chacun. Cela signifie donc que, si l'on souhaite vérifier la condition 1 et qu'on ne fait pas attention, il est possible que les généraux utilisent une valeur de *v(i)* différente de celle envoyée par i – et ce même si le général i est loyal. Il ne faut pas permettre cela si nous souhaitons vérifier la condition B. Nous avons alors de plus la nécessité suivante :

2. **Si le $i^{ème}$ général est loyal, alors la valeur qu'il a envoyé doit être utilisée par tous les généraux loyaux comme étant *v(i)*.**

Nous pouvons alors réécrire la condition 1 comme ceci (que le $i^{ème}$ général soit loyal ou pas) :

1') Deux généraux loyaux quelconques utilisent la même valeur pour *v(i)*. Les conditions **1'** et **2** portent toutes deux sur une même valeur envoyée par le $i^{ème}$ général. On peut alors restreindre notre étude du problème à : **comment un seul des généraux envoie-t-il sa valeur aux autres ?**

On formulera cela en termes de commandant qui envoie un ordre à ces lieutenants, ce qui nous amène au problème suivant.

Définition du Problème des Généraux Byzantins

Un Général commandant doit envoyer un ordre à ses n-1 lieutenants, de manière à ce que :

IC1 : Tous les lieutenants loyaux obéissent au même ordre.

IC2 : Si le général est loyal, alors chaque lieutenant doit obéir à l'ordre qu'il a envoyé.

IC1 et IC2 sont connues comme les conditions de consistance interactive (**I**nteractive **C**onsistency conditions). Il faut remarquer que si le commandant est loyal, alors IC2 implique IC1. Mais bien sûr, le commandant peut très bien être un traître.

Ce problème, du point de vue informatique, peut être représenté de la façon suivante :

Soit un réseau de n processeurs qui peuvent communiquer les uns avec les autres seulement par le biais de messages, à travers des canaux de communications bidirectionnel, il faut s'assurer qu'un processeur envoie des données aux n-1 autres processus, de telle façon que :

IC1 : Les processeurs fiables reçoivent les mêmes données.

IC2 : Si le processeur émetteur est fiable, alors la valeur reçue est celle qui a été envoyée.

On voit bien que le problème de non-fiabilité que l'on cherche à résoudre peut alors provenir soit du processeur lui-même, soit de la liaison de donnée. Les deux cas n'ont donc pas à être différenciés. Si une liaison n'est pas fiable, alors le processeur sera considéré comme non fiable. Il faut de plus remarquer que dans le cadre de systèmes informatiques, il est plus probable que des données ne soient pas envoyées, plutôt que fausses.

Il s'agit donc plutôt d'erreurs par omissions, dues au crash d'un processeur par exemple.

2.5 Les messages signés

Avec les messages signés le consensus est toujours possible. Le principe est le suivant : lorsqu'un lieutenant relaie l'information, il ne peut en aucun cas l'altérer. En effet, les messages sont signés et chaque altération est détectée par le lieutenant receveur. Considérons l'exemple suivant avec un général et deux lieutenants. Un lieutenant loyal est toujours en mesure de détecter le traitre. Supposons qu'il reçoit un 'O' du général alors que le général prétend qu'il a envoyé '1' ou il n'a rien envoyé, alors à partir du message du deuxième lieutenant il peut conclure que le général est un traitre. Ceci dit, une valeur par défaut disons 'O' est affectée aux deux lieutenants. Si le lieutenant loyal ne reçoit rien du deuxième lieutenant, alors le lieutenant est un traitre et les commandes du général seront exécutées.

En pratique, les messages signés sont implantés avec une technique cryptographique d'authentification telle que RSA. Le nombre de messages reste le même que celui de l'algorithme des messages oraux, bien que le nombre de fautifs peut dépasser le tiers du nombre total de processeurs. Comme ce nombre de messages est exponentiel en n, il sera plus bénéfique d'avoir un algorithme avec un nombre moins important de messages.

3. Tolérance aux fautes byzantines

La tolérance aux fautes byzantines peut être accomplie en reproduisant le serveur et assurant que toutes les copies exactes du serveur atteignent un accord sur l'entrée en dépit de copies défectueuses (erreurs byzantines). Un tel accord est souvent connu sous le nom de l'accord byzantin.

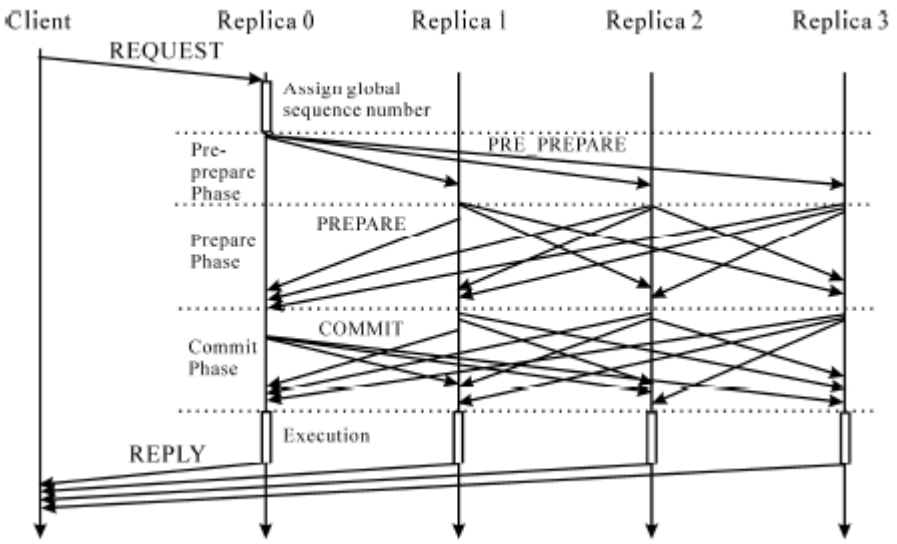

Figure 7 : Tolérance aux fautes byzantines

Le plus d'efficace algorithme concernant l'accord byzantin rapporté ci-dessus est du à Castro et Liskov (connu sous le nom de l'algorithme BFT). L'algorithme BFT est exécuté par au moins 3f + 1 copies exactes du serveur qui toléré f fautes byzantines. Une des copies est désigné comme le 'primaire' pendant que le reste sont des soutiens. L'exécution de l'algorithme BFT implique trois phases, La phase de l'appelle à la préparation, la phase préparation et la phase d'engagement. Durant la première phase, la phase de l'appelle, le serveur primaire envois à tous les autres serveurs un message « préparez-vous » qui contient la requête du client, la vue courante et un numéro de séquence assigné à la requête et à toute les répliques. La réplique vérifie la requête et l'information demandé. Si ce dernier accepte la requête, il envoie à toutes les répliques un message « préparez-vous » contenant l'information du classement et l'abrégé de la requête commandée. Cela débute la deuxième phase, la phase de préparation. Une réplique attend jusqu'à ce qu'elle ait rassemblé 2f messages « préparez-vous » de la part des autres répliques avant d'envoyer un message « engagez-vous » aux autres répliques ce qui débute la troisième phase (la phase d'engagement). La phase d'engagement prend fins quand une réplique a reçu 2f +1 messages assortis « engagez-vous » des autres répliques. À ce point, le message de la demande a été totalement commandé et est prêt à être délivré à l'application du serveur.

Chapitre 3 :La tolérance aux fautes Byzantines dans laPratique

3.1 Introduction :

Une approche largement adoptée pour fournir un service tolérant aux défaillances consiste à dupliquer le service sur un ensemble de serveurs. Dans ce contexte, la réplication de machine à états offre un cadre général permettant de gérer la duplication tout en assurant la consistance du service dupliqué. Néanmoins, la réalisation d'un protocole assurant une duplication cohérente est une tache assez complexe, particulièrement pour les systèmes distribués asynchrone.

Dans un environnement hostile, tel que l'internet, la réplication de machine à états tolérant les fautes Byzantines est une technique importante pour offrir des services robustes. La dernière décennie a vu proliférer les protocoles de réplication de machine à états tolérant les fautes Byzantine, dits protocoles BFT.

Dans ce chapitre, on présente le premier algorithme pratique (protocole) qui a été fait afin de tolérer les fautes Byzantines dans un système distribués asynchrone.

3.2 Un algorithme pratique tolérant aux fautes Byzantines

Dans [CL99], Castro et Liskov ont proposé le premier algorithme qui peut être utilisé dans la pratique afin de tolérer les fautes Byzantines dans un système distribué asynchrone. Cet algorithme est une autre variante de réplication de machines à état, introduit dans [Sch90].

3.2.1 Modèle du système

Castro et Liskov [CL99], supposent un système distribué asynchrone où les nœuds sont connectés par un réseau. Ce réseau peut ne pas délivrer de messages, les retarder ou les délivrer désordonnés. Ils utilisent un modèle de défaillances où les nœuds défaillants, qui peuvent se comporter arbitrairement, sont seulement sujets aux restrictions mentionnées.

Les défaillances des nœuds sont supposées indépendantes les unes des autres. Pour que cette hypothèse soit vraie en présence des attaques malicieuses, quelques étapes doivent être prises en considération, par exemple, chaque nœud doit exécuter des implémentations différentes du code du service et du système d'exploitation, et doit avoir un mot de passe *root*différent et un administrateur différent.

Les auteurs supposent que le système est équipé d'un ensemble de techniques cryptographiques afin d'éviter le *spoofing*et la répétition ainsi que pour détecter les messages corrompus. Les messages contiennent des signatures à clé publique [RSA78], des codes d'authentification de messages [Tsu92], et des *digests* de messages produits par des fonctions de hachage résistantes aux collisions [Riv92]. On dénote un message *m* signé par le nœud *i* comme $<m>\sigma_i$et le *digest* du message *m* par D(*m*).

L'adversaire est supposé très fort, pouvant coordonner des nœuds défaillants, retarder la communication ou retarder des nœuds corrects afin de causer le dommage maximal au service répliqué. On suppose également que l'adversaire possède des capacités limités qui le rendent incapable de subvertir les techniques cryptographiques mentionnées précédemment. Par exemple, l'adversaire ne peut pas produire une signature valide d'un nœud non-défaillant, calculer l'information récapitulée par un digest à partir du digest, ou trouver deux messages avec le même digest.

3.2.2 Propriétés du service :

L'algorithme de Castro et Liskov [CL99], peut être utilisé pour implémenter n'importe quel service répliqué de façon déterministe avec un état et certaines opérations. Les opérations ne sont pas restreintes à de simples lectures ou écritures de portion de l'état du service ; ils peuvent effectuer arbitrairement des calculs déterministes en utilisant l'état et les arguments de l'opération. Les clients envoient des requêtes au service répliqué pour invoquer des opérations et se bloquent en attente d'une réponse. Le service répliqué est implémenté par *n* réplicas. Les clients et les réplicas sont non-défaillants s'ils suivent l'algorithme.

Cet algorithme garantit la *sureté* et la *vivacité* en supposant pas plus de (*n*-1)/3 réplicas défaillants. La sureté exige une borne sur le nombre de réplicas défaillants puisqu'un réplica défaillant peut se comporter de manière arbitraire, par exemple, il peut détruire son état.

La sûreté est fournie indépendamment du nombre de clients défaillants qui utilisent le service ; toutes les opérations effectuées par les clients défaillants sont observées par les clients corrects d'une manière cohérente, en particulier, si les opérations du service sont conçues pour préserver certains invariants sur l'état du service, les clients défaillants ne peuvent en aucun cas violer ces invariants.

Cependant, pour limiter le degré des dommages, le système est équipé par un mécanisme de contrôle d'accès afin d'empêcher un client défaillant d'exécuter des opérations non-légitimes (des clients doivent être authentifiés et l'accès doit être arrêté si le client envoie une requête qui n'a pas le droit d'invoquer l'opération). En outre, les services peuvent fournir des opérations pour changer les permissions d'accès à un client.

Pour satisfaire la *sureté*, l'algorithme ne se base pas sur la synchronie. Par conséquent, il doit se baser sur la synchronie pour garantir la *vivacité*. Par ailleurs, il pourrait être utilisé pour implémenter le consensus dans un système asynchrone, ce qui n'est pas possible [FLP85]. La *vivacité* est garantie, c.-à-d. : les clients reçoivent inéluctablement $n - (n\text{-}1)/3$ réponses à leurs requêtes puisqu'il existe au plus (*n*-1)/3 réplicas qui peuvent être défaillants. Ces réponses doivent être reçues dans un délai *Delay* (O) qui ne peut pas

dépasseré' indéfiniment. *Delay (* é*')* est le temps entre le moment où un message est envoyé pour la première fois et le moment de sa réception par sa destination. Cette hypothèse faible de synchronie est suffisante pour contourner le résultat d'impossibilité de [FLP85].

La résilience de l'Algorithme de Castro et Liskov [CL99] est en effet optimale. (3*t* + 1) est le nombre minimum de réplicas qui permettent à un système asynchrone de garantir les propriétés de *sureté* et de *vivacité* quand au plus *t* réplicas sont défaillants. Ce nombre de réplicas est nécessaires car il est possible de procéder par la suite en communication avec (*n* - *t*) réplicas puisque *t* réplicas peuvent être défaillants et ne pas répondre. Cependant, il est possible que les *t* réplicas qui n'ont pas répondu ne sont pas défaillants et, en conséquence, *t* de ceux qui ont répondu peuvent être défaillants. Néanmoins, il doit y avoir assez de réponses tel que le nombre de réplicas corrects dépasse le nombre des défaillants, c.-à-d.: (*n* >3*t*).

3.3 L'algorithme de Castro et Liskov

L'algorithme proposé par Castro et Liskov est une forme de réplication de machines à état. Le service est modélisé comme une machine à état répliquée sur les différents nœuds d'un système distribué. Chaque réplica de machine à état maintient l'état du service et implémente les opérations du service. On dénote l'ensemble des réplicas par R et on identifie chaque réplica en utilisant un entier dans {0,..., |R|-1}. Pour la simplicité, on suppose que |*R*| = 3*t*+1, où *t* et est le nombre maximum de réplicas qui peuvent être défaillants.

Le rôle de chaque réplica peut se changer d'une configuration à une autre. Chaque configuration est appelée une *vue*. Dans une vue, un réplica est le primaire et les autre sont des *Backup*. Le primaire d'une vue est le réplica p tel que p = *v* mod |R|, où *v* est le numéro de la vue. Des changements de vue sont effectués quand il apparait que le primaire est tombé en panne.

L'algorithme fonctionne généralement comme suit:

1. Un client envoie une requête au primaire pour invoquer une opération de service.
2. Le primaire multicaste la requête aux Backups.

3. Le client attend des réponses avec le même résultat à partir de $t + 1$ réplicas différents. Ces réponses sont le résultat de l'opération.

Comme toutes les techniques de réplication de machine à état [Sch90], on impose deux conditions sur les réplicas : elles doivent être déterministes (c.-à-d. ; l'exécution d'une opération dans un état donné et avec le même ensemble donné d'arguments doit toujours donner le même résultat) et elles doivent commencer dans le même état. Etant donné ces deux conditions, l'algorithme assure la propriété de sûreté en garantissant que tous les réplicas non-défaillants se mettent en accord sur un ordre total pour l'exécution des requêtes en dépit des défaillances.

3.3.1 Le client

Un client c demande l'exécution de l'opération de machine d'état o par l'envoi d'un message<REQUEST, o, 9, c >σ_c au primaire. L'estampillage est utilisé pour assurer exactement une sémantique unique pour l'exécution des requêtes du client. L'estampillage doit être la valeur de l'horloge locale du client quand la requête est envoyée. On utilise les estampilles pour les requêtes afin de les ordonner totalement.

Chaque message envoyé par les réplicas au client inclut le numéro de vue actuel permettant au client de garder une trace pour la vue et par conséquent le primaire actuel. Un client envoie une requête à celui qu'il croit être le primaire actuel en utilisant un mode de communication *point-à-point*. Le primaire envoie d'une façon atomique la requête à tous les Backups en utilisant le protocole décrit dans la prochaine section.

Un réplica envoie la réponse à une requête directement au client. La réponse est sous la forme <REPLY, v, 9, c, i, r >σ_i, où v est le numéro de la vue actuelle, 9 est l'estampillage de la requête correspondante, i est le numéro du réplica et r le résultat de l'exécution de l'opération demandée.

Le client attend des réponses avec des signatures valides à partir de $t + 1$ réplicas différents et avec le mêmes 9 et r avant d'accepter le résultat r. Ceci assure que le résultat est valide, puisqu'au plus t réplicas peuvent être défaillants.Si le client ne reçoit pas des réponses dans les délais, il broadcaste la requête à tous les réplicas. Si la requête a déjà été traitée, les réplicas renvoient simplement la réponse ; les réplicas se souviennent de la dernière réponse envoyée à chaque client. Autrement, si le réplica n'est pas le primaire, il relaye la requête au primaire. Si le primaire ne multicast pas la requête au groupe, il sera

éventuellement suspecté d'être défaillant par un nombre suffisant de réplicas pour provoquer un changement de vue.

3.3.2 Opération de cas normal

L'état de chaque réplica inclut l'état du service, un *message log* contenant les messages que le réplica a accepté, et un entier qui dénote le numéro de vue actuel du réplica.

Quand le primaire *p* reçoit une requête *m* d'un client, il commence l'exécution d'un protocole de trois phases pour multicaster, de manière atomique, la requête aux réplicas. Le primaire commence l'exécution du protocole immédiatement, à moins que le nombre de messages liés au protocole excède un maximum donné. Dans ce cas, il stocke la requête dans un tampon. Les requêtes stockées sont multicastées plus tard comme un groupe afin de réduire le trafic de messages et les overheads du CPU en dessous d'une charge lourde. Pour la simplicité, on ignore cette optimisation dans la description ci-dessous.

Les trois phases sont ; *pre-prepare, prepare et commit*. Les deux premières phases sont utilisées pour ordonner totalement les requêtes envoyées dans la même vue même quand le primaire est défaillant. Les deux dernières phases sont utilisées pour assurer que les requêtes qui exécutent l'opération *commit* sont totalement ordonnées à travers les vues.

Dans la première phase *pre-prepare*, le primaire attribue un numéro de séquence, *n*, à la requête, multicaste un message *pre-prepare*contenant la requête *m* du client (*m* est joint dans *pre-prepare*) à tous les Backups, et joint le message à son log. Le message a la forme <<PRE-PREPARE, *v*, *n*, *d*>σ_p ,*m*> où *v* indique la vue dans laquelle le message a été envoyé, *m* est la requête du client *d* est le digest de *m*.

Des requêtes ne sont pas incluses dans les messages *pre-prepare*pour les maintenir petits. C'est important parce que les messages *pre-prepare*qui portent le numéro de séquence *n* et le numéro de vue *v* sont utilisés comme une preuve pour les changements de vue.

Un backup accepte un message *pre-prepare*fournissant:

- Les signatures dans la requête et le message *pre-prepare*sont correctes et *d* est le digest pour *m*.
- Il est dans la vue *v*
- Il n'a pas accepté un message *pre-prepare*pour la vue *v* et le numéro de séquence *n* et contenant un digest différent.
- Le numéro de séquence *n* dans le message *pre-prepare*est entre une limite basse *h*, et une limite haute, *H*. c.-à-d. : *n* est dans l'intervalle [*h*, *H*].

La dernière condition empêche un primaire défaillant d'épuiser l'espace du numéro de séquence en sélectionnant un espace très grand.

Si le Backup *i* accepte le message <<PRE-PREPARE, *v*, *n*, *d*>σ_p, m> , il entre dans la phase *prepare*en multicastant un message <PREPARE, *v*, *n*, *d*, *i*>σ_ià tous les autres réplicas et ajoute les deux messages à son log, autrement, il ne fait rien.

Un réplica (incluant le primaire) accepte les messages *prepare*et les ajoute à son log stipulant que leurs signatures sont correctes, leur numéro de vue est égal à celui de la vue actuelle et leur numéro de séquence est entre *h* et *H*.

On dit que l'attribut *prepared*(*m, v, n, i*) est vrai si et seulement si le réplica *i* a inséré dans son log : la requête *m*, une pré-préparation pour *m* dans la vue *v* avec un numéro de séquence *n* et 2*t* messages *prepare*des différents Backups liés à la phase *pre-prepare*. Les réplicas vérifient si la phase *prepare*s'accorde avec *pre-prepare*, en vérifiant qu'ils ont la même vue, le même numéro de séquence et le même digest.

Les deux phases *pre-prepare*et *prepare*de cet algorithme garantissent que les réplicas non-défaillants conviennent sur un ordre total pour les requêtes sans vue. Plus précisément, elles assurent les invariants suivants : si *prepared*(*m, v, n, i*) est vrai alors *prepared*(*mÇ v, n, j*) est faux pour n'importe quel réplica non-défaillant *j* (même si *i = j*) et n'importe quel *m′* tels que D(*m′*) ≠ D(*m*). Ceci est vrai puisque *prepared*(*m, v, n, i*) et |R| = 3*t* +1 implique qu'au moins *t* +1 réplicas non-défaillants ont envoyé un message *pre-prepare*ou *prepare*pour *m* dans la vue *v* avec le numéro de séquence *n*. Ainsi, pour que *prepared*(*mÇ v, n, j*) soit vrai, au moins un de ces réplicas nécessite d'avoir envoyé deux messages *prepare*conflictuels (ou *pre-prepare*s'il est le primaire pour *v*), c.-à-d. ;

deux messages *prepares*avec la même vue, le même numéro de séquence et un digest différent. Mais ceci n'est pas possible car le réplica n'est pas défaillant. Finalement, la supposition au sujet de la force des digests des messages assure que la probabilité que $m \neq m'$ et D(m)=D(m') est négligeable.

Une fois *prepared*(*m, v, n,* i) est vrai, le réplica identifié par *i* multicaste un message sous la forme <COMMIT, *v, n,* D(*m*), i>$\sigma_i$aux autres réplicas. C'est le début de la phase *commit*. Les réplicas acceptent les messages *commit* et les insèrent dans leur *logs* si : ils sont correctement signées, le numéro de vue dans le message est égal à la vue courante du réplica, et le numéro de séquence est entre *h* et *H*.

On définit les deux attributs *committed*et *committed-local* comme suit: *committed*(*m, v, n*) est vrai si et seulement si *prepared*(*m, v, n, i*) est vrai pour tous *i* dans un certain ensemble de t +1 réplicas non-défaillants. *committed-local* **(***m, v, n, i***)** est vrai si et seulement si *prepared (m, v, n, i)* est vrai et *i* a accepté 2t +1 *commits*(probablement comprenant le sien) des différents réplicas qui relaye la *pre-prepare*pour *m* ; un *commit* relaye un *pre-prepared*s'ils ont la même vue, même numéro de séquence, et le même digest.

La phase *commit* assure l'invariante suivante: si *committed-local* (*m, v, n, i*) est vrai pour un certain *i* non-défaillant, alors *committed*(*m, v, n*) est vrai. Cet invariant et le protocole de changement de vue assurent que les réplicas non-défaillants consentent sur les numéros de séquence des requêtes qui valident localement même si elles valident dans différentes vues à chaque réplica. En outre, elle garantie que n'importe quelle requête qui est localement valide au niveau d'un réplica non-défaillant sera valide au niveau de (t + 1) réplicas non défaillants ou plus.

Chaque réplica *i* exécute l'opération demandée par *m* après que *committed-local* (*m, v, n, i*) soit vrai et l'état du *i* reflète l'exécution séquentielle de toutes les requêtes avec des numéros de séquence inférieurs. Ceci assure que tous les réplicas non-défaillants exécutent des requêtes dans le même ordre en garantissant la propriété de sûreté. Après l'exécution de l'opération demandée, les réplicas envoient une réponse au client. Les réplicas jettent les requêtes dont l'estampillage est inférieur à l'estampillage dans la dernière réponse qu'ils ont envoyé au client pour garantir les sémantiques.

On ne compte pas sur une bonne livraison de messages, et donc, il est possible qu'un réplica valide des requêtes en panne. Ceci n'importe pas puisqu'il garde les messages *pre-prepared, prepared, et commit* notés jusqu'à ce que la requête correspondante puisse être exécutée.

La *figure 3.1* montre l'opération de l'algorithme dans le cas normal sans défaillances primaires. Le réplica 0 est le primaire, le réplica 3 est défaillant, et *C* est le client.

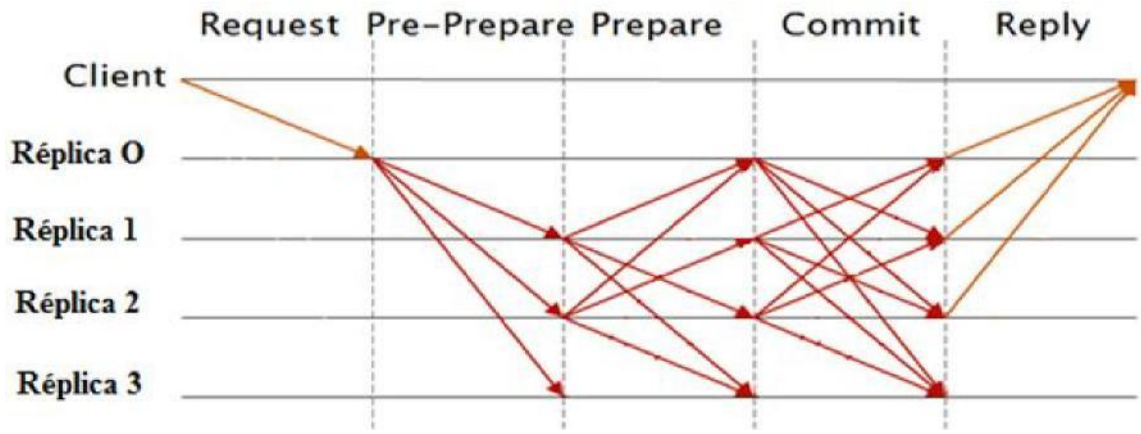

Figure 8: Le protocole PBFT de Castro et Liskov

3.3.3 Collection de miettes

Cette section discute le mécanisme utilisé pour jeter des messages du log. Pour que la condition de sûreté tienne, les messages doivent être maintenus dans le log d'un réplica jusqu'à ce qu'il sache que les requêtes qu'elles concernent ont été exécutées par au moins *t+1* réplicas non-défaillants et il peut prouver ceci aux autres dans des changements de vue. En outre, si un réplica manque les messages qui ont été jetés par touts les réplicas non-défaillants, il devra être mis à jour en transférant tout ou une partie de l'état du service. Par conséquent, les réplicas ont également besoin d'une preuve que l'état est correct.

Générer ces preuves après exécution de chaque opération serait cher. Au lieu de cela, elles sont générées périodiquement, quand une requête avec un numéro de séquence divisible par une certaine constante (par exemple 100) est exécutée. Nous nous référerons aux états produits par l'exécution de ces requêtes en tant que points de contrôle et nous indiquerons qu'un point de contrôle avec une preuve est un point de contrôle stable.

Un réplica maintient plusieurs copies logiques de l'état du service: le dernier point de contrôle stable, zéro ou plus de points de contrôle qui ne sont pas stables et un état actuel. Les techniques Copy-on-write peuvent être utilisées pour réduire les frais généraux de l'espace pour stocker les copies supplémentaires de l'état.

La preuve de l'exactitude pour un point de contrôle est produite comme suit. Quand un réplica produit un point de contrôle, il multicaste un message <CHECKPOINT, *n*, *d*, *i* >$\sigma_i$aux autres réplicas, où *n* est le numéro de séquence de la dernière requête dont l'exécution est reflétée dans l'état, et *d* est le digest de l'état. Chaque réplica rassemble des messages de points de contrôle dans son log jusqu'à ce qu'il ait 2*t* +1 d'eux pour le numéro de séquence *n* avec le même digest *d* signé par différents réplicas (y compris probablement son propre message). Ces 2*t* +1 messages sont la preuve de l'exactitude pour le point de contrôle.

Un point de contrôle avec une preuve devient stable et le réplica jette tous les messages (pre-prepare, prepare, et commit) avec un numéro de séquence inférieur ou égal à *n* de son log; elle jette également tous les anciens points de contrôle et messages de point de contrôle.

Le calcul des preuves est efficace, car le digest peut être calculé en utilisant la cryptographie progressive [BM97], et les preuves sont produites rarement.

Le protocole de contrôle est utilisé pour faire avancer les marks les limites ***h*** et ***H*** (qui limitent quels messages seront acceptés). La limite ***h*** est égal au numéro de séquence du dernier point de contrôle stable. La limite ***H*** = ***h*** + *K*, où *K* est assez grand pour que les réplicas ne restent pas en attente d'un point de contrôle pour devenir stable. Par exemple, si les points de contrôle sont pris toutes les 100 requêtes, *K* peut être 200.

3.3.4 Changement de vue

Le protocole de changement de vue garantit la *vivacité* en permettant au système de faire une progression quand le primaire est défaillant. Les changements de vue sont déclenchés par des *timeouts* qui empêchent les backups d'attendre indéfiniment des requêtes à exécuter. Un backup est en attente d'une requête s'il a reçu une requête valide et il ne l'a pas exécuté. Un backup démarre un *timer*quand il reçoit une requête et le *timer*n'est pas

encore en cours d'exécution. Il arrête le *timer*quand il n'attend pas pour exécuter une requête, mais il redémarre si, à ce stade, il est en attente pour exécuter une autre requête.

Si le *timer*du backup *i* expire dans la vue *v*, le backup démarre un changement de vue pour déplacer le système vers la vue *v* + 1. Il arrête d'accepter des messages (autres qu'un point de contrôle, un changement de vue et les messages de la nouvelle vue) et multicaste un message <VIEW-CHANGE, *v*+1, *n, C, P, i* >a_i à tous les réplicas. Ici *n* est le numéro de séquence du dernier point de contrôle stable, *s*, connu pour *i*, *C* est un ensemble de 2*t* + 1 messages de point de contrôle valides prouvant l'exactitude de *s* et *P* est un ensemble contenant un ensemble *Pm* pour chaque requête *m* (requête *prepare*) envoyé par *i* à avec un numéro de séquence supérieur à *n*. Chaque ensemble *Pm* contient un message *pre-prepare*valide (sans le message du client correspondant) et 2*t* relais, des messages *prepare*valides signés par différents backups avec la même vue, le même numéro de séquence et le même digest de *m*.

Lorsque le primaire *p* de la vue *v* + 1 reçoit 2*t* messages de changement de vue valides pour la vue *v* + 1 provenant d'autres réplicas, il multicaste un message <NEW-VIEW, *v* + 1, *V*, *O*>a_p à tous les autres réplicas, où *V* est un ensemble contenant les messages de changement de vue valides reçues par le primaire ainsi que le message de changement de vue pour *v* + 1 que le primaire a envoyé (ou aurait envoyé), et *O* est un ensemble de messages *pre-repare*(sans la requête superposée). *O* est calculé comme suit:

1) Le primaire détermine le numéro de séquence *min-s*du dernier point de contrôle stable en *V* et le numéro de séquence le plus élevé *max-s* dans un message *prepare*en *V*.

2) Le primaire crée un nouveau message *pre-prepare*pour la vue *v* +1 pour chaque numéro de séquence *n* entre *min-s*et *max-s*. Il ya deux cas:
    a) il ya au moins un ensemble dans la composante P de quelque message de changement de vue en *V* avec le numéro de séquence *n*.
    b) il n'y a pas un tel ensemble.

Dans le premier cas, le primaire crée un nouveau message <PRE-PREPARE, *v* +1, *n*, *d*>a_p, où *d* est le digest de la requête dans le message *pre-prepare*pour le numéro de séquence *n*

avec le numéro de vue le plus élevé en *V*. Dans le second cas, il crée un nouveau message *pre-prepare*<PRE-PREPARE, *v* +1, *n*, d^{null}>σ_p, où ^{dnull} est le digest d'une requête nulle spéciale; une requête nulle passe par le protocole comme les autres requêtes, mais son exécution est un no-op. (Paxos [Lam89] a utilisé une technique similaire pour combler les lacunes.)

Ensuite, le primaire ajoute des messages en *O* à son log. Si *min-s*est supérieur au numéro de séquence de son point de contrôle stable, le primaire insère également la preuve de stabilité pour le point de contrôle avec le numéro de séquence *min-s*dans son log, et rejet l'information de son log, tel que discuté dans la section précédente. Puis il entre dans la vue *v* +1: à ce stade, il peut accepter des messages pour la vue *v* +1.

Un backup accepte un message d'une nouvelle vue pour la vue *v* +1 s'il est correctement signé, si les messages de changement vue qu'il contient sont valides pour la vue *v* +1, et si l'ensemble *O* est correct; il vérifie l'exactitude de *O* en effectuant un calcul similaire à celle utilisée par le primaire pour créer *O*. Ensuite, il ajoute les nouvelles informations à son log comme décrit pour le primaire, multicaste un *prepare*pour chaque message en *O* à tous les autres réplicas, ajoute ces *prepares*à son log et entre dans la vue v +1.

Par la suite, le protocole se déroule comme décrit dans les sections précédentes. Les réplicas ré-exécutent le protocole pour les messages entre *min-s*et *max-s*, mais ils évitent les requêtes des clients en réexécution (en utilisant leurs informations stockées sur la dernière réponse envoyée à chaque client).

Un réplica peut perdre une requête *m* ou un point de contrôle stable (puisque ceux-ci ne sont pas envoyés dans un message de nouvelle vue). Il peut obtenir les informations manquantes à partir d'un autre réplica. Par exemple, le réplica *i* peut obtenir un point de contrôle d'état manquant, *s*, de l'un des réplicas dont les messages de point de contrôle ont certifié sa justesse en *V*. Comme *t* +1 de ces réplicas sont corrects, le réplica *i*aura toujours obtenu *s* ou un dernier point de contrôle stable certifié. On peut éviter d'envoyer entièrement le point de contrôle en partitionnant l'état et estampillant chaque partition avec le numéro de séquence de la dernière requête qui l'a modifié. Pour mettre un réplica à jour, il est seulement nécessaire de lui envoyer les partitions où il n'est pas à jour, plutôt que le point de contrôle entier.

3.4 Conclusion

Bien que l'algorithme de Castro et Liskov, qui est le premier protocole pratique de réplication Byzantine, donne les meilleurs résultats dans un système distribué asynchrone sans attaques, il est cependant dégradé s'il est appliqué dans un système sujet à des attaques. Après avoir décrit en détails le protocole BFTP, et avoir remarqués ses dégradations, notre objectif est de minimiser au maximum les dégradations qui peuvent être causées par un client malicieux ou un primaire qui est défaillant.

Dans le chapitre suivant, nous proposons un protocole de réplication Byzantines qui traitera dans ce contexte.

Chapitre 4 : Un protocole de réplicationByzantine sousAttaques

4.1 Introduction :

Les protocoles existants de réplication de machine à état tolérant aux défaillances Byzantines sont évalués par rapport à deux critères standards de correction : la sûreté et la vivacité. La sûreté signifie que les serveurs corrects ne prennent pas de décisions contradictoires, tandis que la vivacité signifie que chaque update à l'état répliqué est inéluctablement exécuté. La plupart des protocoles de réplication sont conçus pour maintenir la sûreté dans toutes les exécutions, même lorsque le réseau délivre des messages avec des délais arbitraires. Cependant, le résultat d'impossibilité de FLP [FLP85] implique qu'aucun protocole d'accord Byzantin, dans le contexte de l'asynchronisme, ne peut toujours être sûr et vivace à la fois, et ces systèmes assurent seulement la vivacité pour une synchronie et une connectivité suffisante [DLS88]. Lorsque le réseau est suffisamment stable et il n'ya pas de défaillances Byzantines, les systèmes de réplication tolérant aux défaillances Byzantines peuvent satisfaire des garanties de performance beaucoup plus fortes que la vivacité. Dans la littérature, de nombreux exemples de systèmes ont été évalués dans ces exécutions bénignes et qui atteignent des débits de milliers d'opérations update par seconde [CL02], [KADCW09]. Il a été une pratique moins courante d'évaluer les performances des systèmes de réplication tolérant aux défaillances Byzantines lorsque certains des processus exhibent un comportement Byzantin.

Dans ce chapitre, nous remarquons que dans de nombreux systèmes, un petit nombre de processus Byzantins peut dégrader les performances à un niveau très bas par rapport à ce qui serait réalisable avec seulement des processus corrects. Plus précisément, un processus Byzantin peut provoquer le système à faire des progrès à un rythme extrêmement lent, même lorsque le réseau est stable et il pourra supporter un débit beaucoup plus élevé.

En effet, les systèmes qui sont vulnérables à une telle dégradation d'exécution sont très peu utilisés dans la pratique dans les environnements où des adversaires sont présents.
Dans ce chapitre, nous allons présenter certaines attaques possibles pour BFTP de Castro et Liskov [CL99].

4.2 BFT sous attaques

Cette section présente une analyse de certaines attaques possibles pour le BFT protocole de Castro et Liskov [CL99], un protocole tolérant aux défaillances Byzantines basé sur le leader. Nous avons choisi BFT puisque :

1. C'est un protocole largement étudié auquel d'autres protocoles tolérant aux défaillances Byzantines sont souvent comparés.

2. Plusieurs attaques qui peuvent être appliquées à BFT sont également appliquée à d'autres protocoles basés sur un leader.

3. Son implémentation est publiquement disponible.

Le BFT donne de bon résultats dans le cas d'absence de défaillances ou quand les serveurs exhibent des fautes simples. Dans la suite nous décrivons quelques attaques qui peuvent dégrader les performances de BFT.

4.2.1 Attaque par un client Byzantin :

Un client malicieux peut introduire une latence très importante. Si le client n'envoie pas une opération mais qu'il arme son timeout, alors ce délai va s'expirer sans qu'il reçoive une réponse. Par la suite, le client broadcaste l'opération à tous les serveurs. Les serveurs qui ne sont pas primaires relayent l'opération du client au primaire dès sa réception. Le protocole BFT ne fournit pas de mécanisme pour réduire les dégâts de cette attaque non-détectable (nous n'avons pas une façon pour observer les messages envoyés par un client si ses messages ne sont pas reçus par au moins deux serveurs), surtout si le nombre de clients Byzantins est non limité. Dans ce scenario, le nombre de messages va augmenter de 1 à (2N-1).

4.2.2 Attaque par un primaire Byzantin

Un primaire malicieux peut ignorer des opérations envoyées directement par des clients. Si le timeout d'un client sur le primaire s'expire avant de recevoir une réponse pour son opération, alors il broadcaste l'opération à tous les serveurs. Les serveurs qui ne sont pas primaires relayent l'opération du client au primaire dès sa réception. Dans ce cas, même si le primaire va se comporter correctement dans les prochaines phases, une latence très importante sera introduite avant de recevoir la réponse par le client. En plus, le nombre de messages va augmenter de 1 à (2N -1). Cette attaque aussi est non observable car il existe plusieurs possibilités.

4.3 Modèle du système et propriétés du service

Nous considérons un système composé de *N* serveurs et de *M* clients (collectivement appelés processus). Les processus communiquent entre eux par échange de messages. Chaque serveur a une identité unique dans l'ensemble R = {1, 2,...N}, et chaque client a une identité unique dans l'ensemble *S* = {N+1, N+2,...N+M}. Nous supposons un modèle de défaillance Byzantines dans lequel les processus sont corrects ou défaillants. Les processus corrects suivent exactement les spécifications des protocoles, tandis que les processus défaillants peuvent dévier arbitrairement sur les spécifications du protocole par l'envoi de n'importe quel message à n'importe quel moment. Nous supposons que $N \geq (3t + 1)$, où t est la borne maximal de serveurs qui peuvent être défaillants. Pour la simplicité,nous décrivons le protocole pour le cas où $N = (3t + 1)$. Nous supposons aussi que le nombre de clients défaillants est non connu.

Nous supposons un système asynchrone, dans lequel la vitesse relative de transmission de n'importe quel message est non bornée. Le système satisfait notre propriété de *sureté* dans toutes les exécutions dans lesquels au plus *t* serveurs sont défaillants. Le système garantie notre propriété de *vivacité* et les propriétés de performances seulement dans des sous-ensembles d'exécution dans lesquels les vitesses relatives de transmission satisfont certaines contraintes. Tous les messages envoyés entre deux processus sont numériquement signés. Nous dénotons un message *m* signé par un processus *i* par $<m>\sigma_i$. Nous supposons que les signatures digitales et les signatures numériques sont non forgeables sans connaitre une clé privée d'un processus. Nous nous servons également d'une fonction de hachage D résistante aux collisions pour calculer les digests de messages. Nous dénotons le digest du message *m* par D(*m*).

Un client soumet une opération au système par son envoi à un ou plusieurs serveurs. Les opérations sont classifiées en deux catégories ; Read-Only (Queries) et Read-Write (Updates), chaque opération est signée. Chaque serveur produit une séquence d'opération comme sortie (Output), la sortie reflète l'ordre dans lequel le serveur exécute les opérations du client. Quand le serveur produit comme sortie une opération, il envoie une réponse contenant le résultat de l'opération au client.

Sureté : Les réponses du serveur pour des opérations soumises par des clients corrects sont correctes conformément à la linéarisabilité [HW90] modifiée pour faire face aux clients défaillants. Notre protocole établie un ordre total sur les opérations du client. Formellement, la propriété de sureté est définie comme suit :

Définition 4.1 (Sureté): Dans toutes les exécutions où au plus *t* serveurs sont défaillants, les séquences de sortie de deux serveurs corrects sont identiques, ou l'une des séquences de sortie est un préfixe de l'autre.

La propriété de vivacité : La *vivacité* est garantie car les clients reçoivent inéluctablement $2t+1$ réponses à leurs requêtes puisqu'il existe au plus *t* réplicas qui peuvent être défaillants. Ces réponses doivent être reçues dans des délai limité avec une borne supérieure O.

4.4 Un protocole de réplication Byzantine sous attaques :

4.4.1 Description du protocole

Pour demander un service, le client envoie une requête numériquement signée au primaire et à 2*t* réplicas choisis aléatoirement. Le format de la requête envoyée par le client est <REQUEST, *o*, o, *c*>σ_c ,où*o* est l'opération demandée, o l'estampillage de la requête et *c*est l'émetteur du message (le client). L'estampillage est utilisé afin d'ordonner totalement les requêtes des clients. L'estampillage doit être la valeur de l'horloge locale du client quand il envoie sa requête.

Après avoir reçu la requête par les 2t +1 réplicas, quatre autres phases du protocole doivent être respectées avant que la réponse soit envoyée au client. Ces différentes phases sont comme suit :

Dans la première phase, chacun des réplicas ayant reçu la requête *m* du client diffuse un message <*m*>σ_i, qui est la requête du client avec la signature du réplica qui l'envoie, à tous les différents réplicas (y compris le réplica qui n'a pas reçu cette requête directement du client). Chacun des réplicas attend au moins *t* +1 messages identiques. Cette phase permet de s'assurer que le primaire et tous les autres réplicas reçoivent la requête du client, et que cette requête est juste : si les différents réplicas ne reçoivent pas au moins *t* +1 messages identiques, alors la requête est rejetée, sinon le protocole valide la requête et entame la deuxième phase.

La deuxième phase est la phase *pre-prepare*. Dans cette phase, le primaire multicaste un message *pre-prepare*sous la forme <<PRE-PREPARE, *v*, *n*, *d*>σ_p ,*m*>, où *n* est le numéro de séquence, *v* est le numéro de la vue actuelle, *m* est la requête du client, et *d* le digest du message *m*. Le numéro de vue *v* et le numéro de séquence *n* sont utilisés comme preuve pour les changements de vue. Le message *pre-prepare*est accepté si les signatures de la requête sont correctes, ainsi que le numéro de séquence et le numéro de vue.

Cette phase permet de vérifier si le primaire suit correctement le protocole, sinon un changement de vue aura lieu et un nouveau primaire sera élu.

La troisième phase (*prepare*) est entamée si le message *pre-prepare*est accepté. Dans cette phase, les backups multicastent un message *prepare*à tous les autres réplicas (y compris le primaire) et l'ajoute à son log. Les messages *prepare*ont la forme <PREPARE, *v*, *n*, *d*, *i*>σ_i, où *v* est le numéro de vue, *n* le numéro de séquence, *d* le digest de m et *i* identifie l'émetteur.

Chacun des backups doit recevoir 2*t* messages *prepare*provenant des autres backups. Les réplicas vérifient alors si les deux phases *prepare*et *pre-prepare*s'accordent en vérifiant qu'ils ont la même vue, le même numéro de séquence et le même digest.

L'attribut *prepared*(*m, v, n, i*) est vrai si et seulement si le réplica identifié pas *i* a inséré dans son log la requête *m*, le message *pre-prepare*(pre-preparation de *m*) dans la vue *v* avec un numéro de séquence *n* et 2*t* messages *prepare*des différents Backups qui sont liés à la phase *pre-prepare*.

Les deux phases *prepare*et *pre-prepare*assurent que si *prepared*(*m, v, n, i*) est vrai alors *prepared*(*mÇ v, n, j*) est faux pour n'importe quel réplica non-défaillant *j* (même si *i = j*) et n'importe quel *m'* tels que D(*m'*) ≠ D(*m*).

La quatrième phase (*commit*) est entamée une fois l'attribut *prepared*(*m, v, n,* i) est vrai. Chaque réplica *i* multicaste un message *commit* sous la forme <COMMIT, *v*, *n*, D(*m*), i>σ_i. Si les messages *commit* sont correctement signés, le numéro de vue est celui de la vue courante et le numéro de séquence est correct, alors ils sont acceptés par les différents réplicas et ajoutés à leur logs.

L'attribut *committed*(*m, v, n*) est vrai si et seulement si *prepared*(*m, v, n, i*) est vrai pour tous *i* dans un certain ensemble de *t* +1 réplicas non-défaillants.

L'attribut *committed-local* **(***m, v, n, i***)** est vrai si et seulement si l'attribut *prepared (m, v, n, i)* est vrai et *i* a accepté 2*t* +1 *commits*(probablement comprenant le sien) des différents réplicas qui relaye le *pre-prepare*pour *m*.

La phase *commit* assure que n'importe quelle requête qui est localement valide au niveau d'un réplica non-défaillant sera valide au niveau de *t* +1 réplicas non défaillants ou plus.

Après que *committed-local* (*m, v, n, i*) soit vrai, chaque réplica *i* exécute l'opération demandée par *m*. Après l'exécution de l'opération demandée, les réplicas envoient une

réponse au client et jettent les requêtes dont l'estampillage est inférieur à l'estampillage dans la dernière réponse qu'ils ont envoyé au client, afin de garantir les sémantiques.

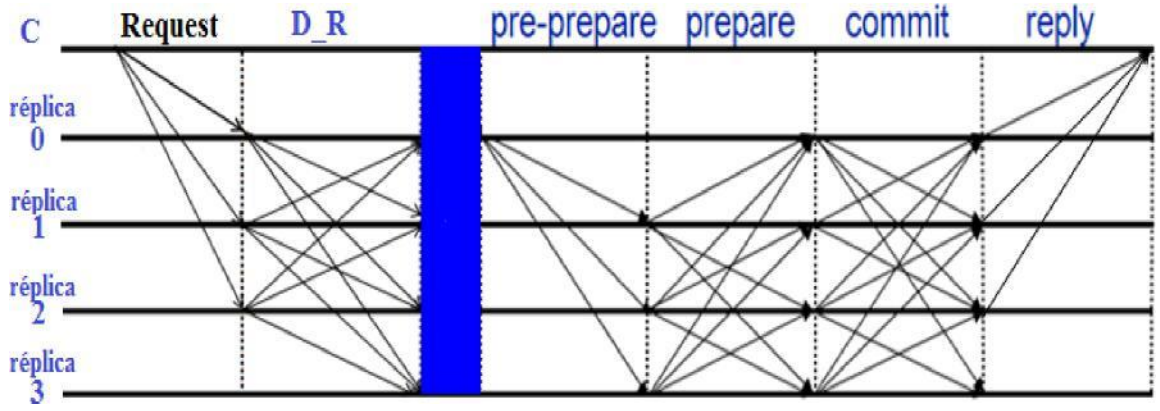

Figure 9:Un exemple d'exécution du protocole proposé dans le cas normal

4.4.2 Sur l'efficacité du protocole

Le protocole proposé donne de meilleurs résultats par rapport à BFTP dans les cas suivants :

Cas N° 1 : Dans le cas de BFTP, quand un client malicieux n'envoie pas une opération mais il arme son timeout, il broadcaste cette opération à tous les réplicas après l'expiration de son timeout. Par conséquence et pour dégrader les performances du système, il broadcaste cette opération pour tous les réplicas. Quand un réplica reçoit cette opération, il la relaye au reste des réplicas. Les performances vont se dégrader plus encore si ce client malicieux décide d'envoyer des opérations valides avec des contenus différents aux différents réplicas.

Dans le cas de notre protocole, nous pouvons éviter ce scénario grâce à un prix à payer au début de l'opération et non après l'expiration du timeout du client. Ce prix à payer est

d'augmenter le nombre de messages de 1 (un message est suffisant pour invoquer une opération dans le cas de BFTP en absence de défaillances) à $2t$ +1. Comparativement à BFTP et après l'expiration du timeout, N messages sont nécessaires pour invoquer une opération. Même dans le cas où un client malicieux envoie des opérations valides à des réplicas différents, notre protocole a la capacité de détecter ce comportement malicieuxgrâce à la deuxième phase (D-R), dont le rôle est de suspecter un client qui n'envoie pas les opérations conformément au protocole. La suspicion se fait grâce à une simple comparaison des opérations valides relayées par d'autres serveurs. Si au moins t +1 serveurs reçoivent deux opérations valides avec des contenus différents, ils informent le reste de serveurs et il abandonne l'opération. Par conséquence, ce client malicieux sera ajouté dans une liste noire pour éviter une autre éventuelle tentative de sa part.

Cas N° 2 : Dans le cas de BFTP, si le primaire ignore une opération valide d'un client correct, le client attend l'expiration de son timeout et s'il ne reçoit pas la réponse, alors il diffuse sa requête à la totalité des réplicas et chaque réplica qui n'est pas primaire relaye cette requête au réplica primaire dès sa réception. Même si le primaire se comporte correctement dans les prochaines phases, une latence très importante sera introduite avant la réception de la réponse.

Dans le cas de notre protocole, le primaire ne peut pas ignorer une requête valide d'un client correct, puisque la requête du client est envoyée au primaire et à $2t$ réplicas choisis aléatoirement. Les réplicas qui ont reçu cette opération la broadcastent à tous les autres réplicas. Par la suite, si le primaire n'envoie pas de message pre-prepare à tous les réplicas, alors il sera suspecté si t+1 réplicas réclament la non réception de message pre-prepare car ils ont la preuve que le primaire a reçu l'opération du client au moins de la part de t+1 réplicas (au moins un est correct). Contrairement à BFTP, un client n'envoie pas son opération une autre fois car il est sûr qu'il va recevoir une réponse. Dans le cas de suspicion du primaire, un changement de vue sera lancé.

4.5 Conclusion

Dans ce chapitre, nous nous somme basé sur le protocole décrit dans le chapitre 3 afin d'apporter des améliorations liées aux performances. Nous avons proposé un protocole de réplication Byzantines traitant des attaques qui peuvent dégrader les performances de BFTP que ce dernier a ignoré. On a cités ces éventuelles attaques avant de décrire notre protocole et on a fini par une discussion sur son efficacité dans un système distribué asynchrone sujets à des attaques que ce soit de la part d'un client malicieux ou bien celles du serveur primaire qui pourrait être défaillant.

Conclusion

Les systèmes distribués, ou répartis, tiennent une place de plus en plus importante dans le monde actuel, en particulier avec l'arrivée de l'Internet. Informellement, ils représentent une abstraction dans laquelle un ensemble d'entités coopèrent entre elles afin d'effectuer une tâche ou d'offrir un service donné.

Dans ce mémoire, nous avons commencé par un chapitre introductif sur les systèmes distribués afin pour que le lecteur de ce document soit en mesure de comprendre notre travail. Nous avons présenté également quelques définitions et résultats quant au problème du consensus qui représente la brique de base pour concevoir des solutions pour tous les autres problèmes d'accord. L'objectif de ceci est de montrer l'incertitude du contexte sur lequel notre travail se base.

Afin de donner une motivation pour notre travail, nous avons présenté en détails le premier protocole pratique de réplication Byzantine proposée par Castro et Liskov. Le choix de ce protocole est justifié par : la disponibilité publique de son implémentation, la majorité de protocoles de réplication Byzantine apportent des petites modifications sur ceci.

Enfin, nous avons donné deux attaques possibles dont BFTP peut être sujet. Ces attaques dégradent les performances de BFTP et ils peuvent causer des dégâts très remarquables. Par la suite, nous avons proposé un protocole pour minimiser le maximum possible les dégâts de ces attaques.

Perspectives

Le protocole proposé dans ce mémoire pour l'implémentation de réplication Byzantine montre sa robustesse d'un point de vue théorique dans le contexte d'un système distribué asynchrone en présence d'attaques. Cette robustesse est justifiée par la non utilisation du *timer*par le client et par le nombre de messages nécessaire pour invoquer un service, mais expérimenter dans un réseau internet qui fonctionne en mode best effort est un objectif très souhaitable. Pour cela, nous envisageons de réaliser cet objectif dans le futur ou nous proposons aux futures promotions de Master d'investir dans ce domaine car il est très important pour l'avenir. Cette importance est justifiée par le nombre explosif de hackers présents sur Internet.

BIBLIOGRAPHIE

Bibliographie

[Ben83] M. Ben-Or. Another advantage of free choice: *Completely asynchronous agreement protocols*. In 2nd ACM Symp. On Principles of Distributed Computing (PODC‟83), pages 27-30, 1983.

[BM97] M. Bellare and D. Micciancio. *A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost*. In Advances in Cryptology – Eurocrypt 97, 1997.

[Cas01] M. Castro, "Practical Byzantine Fault Tolerance," PhD dissertation, Massachusetts Inst. of Technology, pp. 29-31, 2001.

[CDK01] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed systems (3rd ed.) : concepts and design*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[CF99] F. Cristian and C. Fetzer. *The timed asynchronous distributed system model. IEEE Trans. On Parallel & Distributed Systems*, 10(6):642–657, June1999

[Cha90] S. Chaudhuri. Agreement is header than consensus Set consensus problems in totally asynchronous systems. In Proc. 9th ACM Symposium on Principles of Distributed Computing (PODC‟90), pages 311-324, 1990.

[CHT96] T. D. Chandra, V. Hadzilacos et S. Toueg :*The weakest failure detector for solving consensus*. Journal of the ACM, 43(4):685–722, 1996.

[CL02] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," ACM Trans. Computer Systems, vol. 20, no. 4, pp. 398-461, 2002.

[CL85] K. Mani Chandy and Leslie Lamport.*Distributed snapshots : determining global states of distributed systems*. ACM Trans. Comput.Syst., 3(1) :63–75, 1985.

[CL99] Miguel Castro and Barbara Liskov, *Practical Byzantine Fault Tolerance*. Laboratory

[CT96] Tushar Deepak Chandra and Sam Toueg.*Unreliable failure detectors for reliable distributed systems*.Journal of the ACM, 43(2) :225–267, 1996.

for Computer Science,Massachusetts Institute of Technology, 1999.

[CS00] B.Charron-Bost and A.Schiper.*Uniform consensus is harder than consensus (extended abstract).* Technical Report DSC/200/028, EPFL, 2000.

[Del07] Christian DELBÉ, *Tolérance aux pannes pour objets actifs asynchrones : protocole, modèle et expérimentations.* Thèse de Doctorat, Université de NICE - SOPHIA ANTIPOLIS, Janvier 2007

[DLS88] C. Dwork, N. Lynch et L. Stockmeyer : *Consensus in the presence of partial synchrony*. Journal of the ACM, 35(2):288–323, 1988.

[Dum97] Cedric Dumoulin, *Dream: Une mémoire partagée répartie à cohérence programmable*. Thèse de Doctorat, Université des Sciences et Technologies De LILLE. 1997

[FLP85] M. Fischer, N. Lynch, and M. Paterson. *Impossibility of Distributed Consensus With One Faulty Process*. Journal of the ACM, 32(2), 1985.

[HHM+00] Jean-Michel Hélary, Michel Hurfin, AchourMostefaoui, Michel Raynal, and FrédéricTronel.*Computing global functions in asynchronous distributed systems with perfect failure detectors*. IEEE Transactions on Parallel and Distributed Systems, 11(9) :897–909, September 2000.

[HT94] VassosHadzilacos and Sam Toueg.*A modular approach to fault-tolerant broadcasts and related problems*.Technical Report TR94-1425, CornellUniversity, Ithaca, NY, USA, 1994.

[Hug10] Kévin HUGUENIN, *Méthodes dissuasives contre les utilisateurs malhonnêtes dans les systèmes répartis*. Thèse de Doctorat. Université de Rennes 1, Décembre 2010.

[HW90] M.P. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," ACM Trans. Programming and Languages and Systems, vol. 12, no. 3, pp. 463-492, 1990.

[KADCW09] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," ACM Trans. Computer Systems, vol. 27, no. 4, pp. 7:1-7:39, 2009.

[Kef00] Mohamed Ridha Kefi, *Outil pour le masquage /démasquage des fautes byzantines.* Mémoire de maître ès sciences (MSc.), Université de SHERBROOKE, février 2000

[LAM04] L. LAMPORT, and M. MASSA. “Cheap paxos ”. In *DSN ’04 : Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN’04)*, page 307,Washington, DC, USA, 2004. IEEE Computer Society.

[Lam89] L. Lamport. *The Part-Time Parliament*. Technical Report 49, DEC Systems Research Center , 1989.

[LSP80] M. Pease, R. Shostack, and L. Lamport. *Reaching Agreement in the Presence of Faults*.*Journal of the* ACM, 27(2):228-234, 1980.

[LSP82] L. Lamport, R. Shostack, and M. Peaçe. *The Byzantine Generals Problem*.*ACM Transactions on Programming Languages and* Systems, 4(3):382-401, 1982.

[Mal07] Willy MALVAULT, *Protocoles de diffusion totalement ordonnée pour les systèmes distribués synchrones*. Mémoire de Master 2. Université Joseph Fourier, juin 2007.

[Mar04] Corine MARCHAND, *Mise au point d'algorithmes répartis dans un environnement fortement variable, et expérimentation dans le contexte des pico-réseaux.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, 2004.

[MBS03] O. MARIN, M. BERTIER, and P. SENS. Darx - *a framework for the fault-tolerant support of agent software*.*issre*, 00 :406, 2003.

[MRR01] A. Mostéfaoui, S. Rajsbaum, and M. Raynal. *Conditions on input vectors for consensus solvability in asynchronous distributed systems*.In Proc. of the 33rd ACM Symposium on Theory of Computing (STOC'01), pages 153-162. ACM Press, 2001

[MRR02] A. Mostéfaoui, S. Rajsbaum, and M. Raynal. *A versatile and modular consensus protocol*.In Proc.Of the 2002 International Conference on Dependable Systems and Networks (DSN-2002).IEEE, 2002.

[MRRR01] A. Mostéfaoui, S. Rajsbaum, M. Raynal, and M. Roy. *Efficient condition-based consensus*.In Proc. of the 8th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO'01), pages 275{291.Carleton Univ. Press, 2001.

[MRT00] A. Mostéfaoui, M. Raynal, and F. Tronel. *The best of both worlds: a hybrid approach to solve consensus*. In Proc. of the Int. Conference on Dependable Systems and Networks (FTCS/DCCA), pages 513-522, New York, NY, Jun 2000.

[Mul93] S. E. Mullender. *Distributed Systems*.Addison-Wesley, 2nd Edition, 1993.

[Mul94] S. Mullender, *Distributed Systems* (*second edition*). ACM Press, 1994.

[Ple07] Julien PLEY, *Protocoles d'accord pour la gestion d'une grille de calcul dynamique.* Thèse de Doctorat, Université de Rennes 1, décembre 2007.

[PR3] P.RaipinParevédy and M. Raynal. *Uniform agreement despite process omission failures*. In Proc. IEEE IPDPS workshop on fault-tolerant Parallel and distributed Systems (FTPDS‟03). IEEE, 2003

[PR4] P.RaipinParevédy and M. Raynal. *Optimal early stopping uniform consensus in synchronous system with process omission failures*. In Proc. of Sixteenth ACM symposium on Parallelism in Algorithms and Architectures (SPAA‟04). ACM Press, 2004.

[Riv92] R. Rivest. The *MD5 Message-Digest Algorithm*. Internet RFC-1321, 1992.

[Roy03] Matthieu ROY, *Synchronisation distribuée sans attente : application à la résolution des problèmes d'accord par contrainte des données*. Thèse de Doctorat, Université de Rennes 1, Novembre 2003

[RSA78] R. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*.Communications of the ACM, 21(2), 1978.

[Sch90] F. Schneider .*Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial*. ACM Computing Surveys, 22(4), 1990.

[Tan97] A. Tanenbaum, *Réseaux (3e edition).* Dunod / Prentice Hall, 1997.[Tsu92] G. Tsudik. *Message Authentication with One-Way Hash Functions*.ACM Computer Communications Review, 22(5), 1992. [Riv92] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC-1321, 1992.

Résumé

Les protocoles de réplication Byzantine existants satisfont les deux critères standards d'exactitude, la *sûreté* et la *vivacité*, même en présence de défaillances Byzantines. Les performances de ces protocoles sont le plus généralement évaluées en l'absence de défaillances de processeurs et sont habituellement bonnes dans ce cas. Cependant, les processus défaillants peuvent de manière significative dégrader les performances de quelques protocoles, limitant leur utilité pratique dans les environnements de confrontation. Ce travail démontre l'ampleur de la dégradation de performances possible dans quelques protocoles existants qui satisfont la *vivacité* et qui donnent de bons résultats en l‟absence de défaillances Byzantines. Nous proposons un nouveau protocole de réplication, tolérant aux défaillances Byzantines, qui répond au nouveau critère d'exactitude et évaluons ses performances dans des exécutions sans défaillances et sous attaques.

Mots clés : Attaques, tolérance aux fautes Byzantines, réplication de machines à état, systèmes distribués.

Abstract

Existing Byzantine-resilient replication protocols satisfy two standard correctness criteria, safety and liveness, even in the presence of Byzantine faults. The performance of these protocols is most commonly assessed in the absence of processor faults and is usually good in that case. However, faulty processes can significantly degrade the performance of some protocols, limiting their practical utility in adversarial environments. This work demonstrates the extent of performance degradation possible in some existing protocols that do satisfy liveness and that do perform well absent Byzantine faults. We propose a new Byzantine fault-tolerant replication protocol that meets the new correctness criterion and evaluate its performance in fault-free executions and when under attack.

Key Word: under attack, Byzantine fault tolerance, replicated state machines, distributed systems.