Algorithmique Distribuée
Tolérance aux pannes

Laurent Philippe

Master 2 Informatique
Université de Franche-Comté
UFR des Sciences et Techniques

2018 / 2019

Sommaire

Introduction

Contexte

La tolérance aux fautes s'inscrit dans le domaine plus vaste de la sûreté de fonctionnement.

Sûreté de fonctionnement - Dependable Computing

Propriété d'un système informatique qui permet à ses utilisateurs de placer une confiance justifiée dans le service que délivre le système.
"[..] the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [..]" [a]

a. IFIP WG10.4 on Dependable Computing and Fault Tolerance

Sûreté de fonctionnement

Les différents aspects de la sûreté de fonctionnement

- Fiabilité : le système est en état (continu) de rendre le service (reliability)
- Disponibilité : le service est disponible en permanence (availability)
- Sécurité (au sens sûreté) : un mauvais fonctionnement du système n'a pas d'incidence catastrophique sur son environnement (safety)
- Sécurité (au sens protégé) : sécurité vis-à-vis des actions autorisées (security)
- Intégrité : absence d'altération du système (integrity).
- Maintenabilité : capacité à réaliser/subir des modifications et réparations (Maintainability)

Tolérance aux pannes

Les différents aspects de la tolérance aux pannes

- Fiabilité
- Disponibilité
- Sécurité

Resilience (capacité de reprise)

"Resilience is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation."

Menaces pour la tolérance

La fiabilité du système peut-être affectée par des menaces (threats). Trois termes caractérisent ces menaces :

- Faute (fault ou bug) : **défaut du système**. La présence d'une faute peut ou non conduire à un échec si les conditions de production de la faute ne sont jamais atteints.
- Erreur (error) : **divergence entre le comportement attendu et le comportement actuel**. Les erreurs se déclenchent à l'exécution lorsque que tout ou partie du système entre dans un état non attendu à cause de l'activation d'une faute. Dans la mesure où les erreurs sont générées par des états invalides elle peuvent être difficiles à observer sans outil, comme un debugger.
- Échec (failure) : **instance de temps où le système produit des comportements contraires à sa spécification**. Une erreur ne conduit pas forcément à un échec dans la mesure où des techniques de tolérance aux pannes peuvent être mise en œuvre pour limiter l'impact des erreurs. Par exemple sur une erreur une exception peut-être levée pour réaliser un traitement conforme aux spécifications.

Menaces pour la tolérance

- Les échecs sont observées aux limites du système. Ce sont des erreurs qui sont devenues observables
- Les fautes, erreurs et échecs sont liés par un enchaînement :
  Faute $\rightarrow$ Erreur $\rightarrow$ Échec
- Lorsqu'une faute est activée, une erreur peut agir comme comme une faute car elle peut produire de nouvelles conditions d'erreur.
- Lorsqu'une erreur se propage en dehors des limites du système elle conduit à un échec
- L'échec est le point à partir duquel on peut dire que le service ne satisfait plus ses spécifications
- Puisque les données de sortie d'un service peuvent être utilisées par un deuxième, l'échec d'un service peut se propager à un autre sous la forme d'une faute. La chaîne boucle alors :
  Faute $\rightarrow$ Erreur $\rightarrow$ Échec $\rightarrow$ Faute

Moyens

La compréhension des mécanismes inhérents à la chaîne Faute-Erreur-Echec permet la construction d'outils pour la briser et ainsi augmenter la fiabilité du système.

- La prévention qui vise à limiter l'introduction de fautes dans le système (méthodes de développement, techniques d'implémentation)
- La suppression de fautes qui peut être réalisée soit pendant le développement (vérification, preuve) soit en production (cycle de maintenance)
- Le pronostic tente de prédire les fautes pour les contourner
- La tolérance aux fautes vise à introduire des mécanismes qui permettent au système de continuer à offrir le service demandé en présence de fautes, même si ce service peut être dans un état dégradé.

Modèles temporels

Définition

Le modèle temporel qui fixe les hypothèses temporelles, c'est-à-dire les bornes maximales et minimales des délais de traitement et de communication

Trois modèles temporels

- **Modèle TBC** - Temps Borné et Connu - les bornes existent et elles sont connues à priori
- **Modèle TBI** - Temps Borné mais Inconnu - les bornes existent mais leurs valeurs ne sont pas connues
- **Modèle TNB** - Temps Non Borné - les bornes n'existent pas

On parle aussi de systèmes **synchrones** (modèle TBC) et de systèmes **asynchrones** (modèles TNB).

Modèles de pannes

On peut classer les fautes en fonction des erreurs qu'elles génèrent.

Les défaillances par arrêt

Encore appelées pannes franches ou pannes permanentes ou crash : le processus défaillant s'arrête et ne fournit plus aucune réponse. Toute réponse reçue est supposée correcte.

- En système synchrone, la détection de la panne est réalisée en utilisant des délais.
- En systèmes asynchrone on ne peut pas distinguer un processus très lent d'une panne franche.

Modèles de pannes

Les défaillances par omission

Encore appelées transitoires ou intermittentes : le composant défaillant ne répond pas à un événement en entrée. Le composant ne fournit pas son service habituel pendant une certaine période. Ensuite il peut à nouveau fonctionner de façon correcte.
On parle alors de :

- Panne transitoire : apparaît une seule fois et disparaît
- Panne intermittente : elle est plus ou moins périodique

Exemples :

- perte de messages
- destruction de messages pour éviter la congestion
- destruction d'activités pour éviter l'inter-blocage ou les problèmes de contrôle de concurrence.

Modèles de pannes

Les défaillances par valeur

Le composant défaillant répond dans le temps imparti, mais envoie une valeur erronée par rapport au résultat attendu.
Exemple : une ligne de communication qui délivre des messages altérés par rapport à leur contenu initial.

Les défaillances temporelles

Le composant défaillant répond avec une bonne valeur, mais en dehors du temps imparti (soit trop tôt, soit trop tard).
Exemples :

- surcharge d'un processeur
- horloge trop rapide
- délai de transmission trop long

Modèles de pannes

Les défaillances arbitraires (byzantines)

Ce sont les défaillances qui combinent à la fois les fautes temporelles et les fautes de valeur. Évidemment, c'est le cas le plus difficile à traiter.

Hiérarchie des modèles de pannes

- Un crash est un cas particulier de défaillance arbitraire
- Un processus qui ne démarre pas (*Initially dead process*) est un cas particulier d'un crash

Modèles de pannes

Le modèle de défaillance choisi est aussi important que le modèle temporel, puisqu'il détermine la solution de conception.
On verra par la suite que :

- chaque technique de tolérance aux fautes permet de masquer un certain type de pannes
- plus le type de défaillance est sévère, plus la technique de tolérance aux fautes appropriée est coûteuse en termes de ressources et de complexité.

Modèles d'algorithmes

Algorithmes robustes

- Conçus pour garantir le fonctionnement continu et correct des processus qui s'exécutent correctement malgré les pannes qui arrivent dans d'autres processus.
- Une illustration des algorithmes robustes peut-être donnée à travers les algorithmes de vote dans lesquels un processus n'acceptera certaines informations qu'à condition que suffisamment d'autres processus aient déclaré avoir reçu la même information. Par contre, il n'attendra jamais de recevoir des informations de l'ensemble des autres processus car cela conduit à un deadlock en cas de crash.
- Le nombre de fautes est généralement limité car la réponse à une faute est souvent un état dégradé.

Modèles d'algorithmes

Algorithmes auto-stabilisants

- Supposent que les processus reviennent dans un état cohérent au bout d'un temps fini (transient failures)
- Capacité à revenir dans un état cohérent
- Soit intrinsèque, soit après le passage par un état intermédiaire de réparation
- Algorithmes qui supportent un plus grand nombre de fautes, potentiellement infini mais limité dans le temps

Sommaire

Algorithmes tolérants aux pannes

Problèmes de décision

- La littérature sur la tolérance aux panne se concentre sur les algorithmes de décision : chaque processus correct prend une décision en fonction de ses entrées
- N processus au plus t fautes, t-résistant
- Principales propriétés : terminaison, cohérance et non trivialité

Prise de décision

- Déterministe : la prise de décision est prouvée
- Probabiliste : la probabilité de prise de décision doit tendre vers 1. Exlut toutes les solutions où un processus attend plus de $N - t$ réponses

Algorithmes tolérants aux pannes

Terminaison

Tous les processus finissent pas décider, pas d'attente infinie.

- Déterministe : à prouver sur tous les processus
- Probabiliste : la probabilité doit tendre vers 1. Exlut toutes les solutions où un processus attend plus de $N - t$ réponses

Cohérance

Définition d'une relation entre les décisions prises par les différents algorithmes.

- Cas simple : toutes les décisions sont identiques, consensus
- Décision sur la base d'un vecteur de sortie avec une sortie par processus

Algorithmes tolérants aux pannes

Non trivialité

- Exclut les algorithmes avec une sortie fixe du problème où les processus décident sans communication.
- Par exemple, un consensus peut être résolut par des processus qui décident tous 0 dès le début
- Deux décisions différentes sont possibles dans des exécutions différentes de l'algorithme.
- Par exemple un consensus a des exécutions qui conduisent à décider 1 et des exécutions qui conduisent à décider 0

Algorithme déterministe de consensus 1-crash résistant

Définition

- Terminaison : tous les processus décident dans le cas d'une exécution 1-crash
- Accord : si, dans une configuration atteignable, pour deux processus corrects p et q et une valeur initiale b, $y_p \neq b$ et $y_q \neq b$ alors $y_p = y_q$
- Non-trivialité : pour $v = 0$ et pour $v = 1$ (décision) il existe des configurations atteignables dans lequelles pour un processus p, $y_p = v$

Propriété

Fischern Lynch et Paterson ont montré qu'il n'est possible d'obtenir un consensus 1-crash résistant en modèle asynchrone

Initially Dead Processes

Identification des processus actifs

- Modèle simpple : aucun processus ne tombe après avoir exécuté un évènement
- Si on reçoit un message d'un processus, alors il est vivant et on peut attendre de nouveaux messages de sa part
- On appelle une exécution "*t-initially dead fair execution*"' (exécution correcte) une exécution où au moins $L = N - t$ processus sont actifs
- Hypothèse : le réseau est complet et il n'y a pas de perte de message
- Exercice : déterminer l'ensemble des processus actifs

Initially Dead Processes

Var : $Proc_p, Succ_p, Alive_p, Rcvd_p$: ens. de processus **init** $\emptyset$
début

pour $q \in Proc_p$ **faire**
`send`$\langle$**name**, $p\rangle$ à q

tant que $\#Succ_p < L$ **faire**
`receive`$\langle$**name**, $q\rangle$
$Succ_p = Succ_p \cup \{q\}$

pour $q \in Proc_p$ **faire**
`send`$\langle$**pre**, $p, Succ_p\rangle$ à q

$Alive_p = Succ_p$
tant que $Alive_p \not\subseteq Rcvd_p$ **faire**
`receive`$\langle$**pre**, $q, Succ_q\rangle$
$Alive_p = Alive_p \cup Succ_q \cup \{q\}$
$Rcvd_p = Rcvd_p \cup \{q\}$

Initially Dead Processes

Calcul d'une seule composante connexe

- Echange des messages permet de relier entre eux les processus vivants et d'établir en commun l'ensemble des processus actifs
- Si le réseau n'est pas connexe quel est l'effet de l'algorithme ?
- Exercice : à quelle condition est-on sûr d'avoir une seule composante connexe ?
- Une fois l'algorithme de composante connexe achevé et sous les mêmes hypothèses, il est trivial de mettre en place un algorithme d'élection ou de consensus

Algorithmes de consensus probabilistes

Contexte

- Pas d'algorithme de consensus qui peut prendre une décision en un temps fini mais la probabilité peut tendre vers 1
- Repose sur un accord partiel pour chaque exécution
- Probabilité prise sur un ensemble d'exécutions, repose sur une probabilité de réception des messages
- Les algorithmes opèrent en rondes. Dans une ronde un processus diffuse un message et attend les retour de $N - t$ messages. On définit par $R(q, p, k)$ la réception par p du message de ronde k de q
- Hypothèse de *fair scheduling* :
  - $\exists \epsilon, \forall p, q, k : Pr[R(q, p, k)] \geq \epsilon$
  - $\forall k$ et les processus p, q, r les évènements $R(q, p, k)$ et $R(q, r, k)$ sont indépendants

Algorithmes de consensus probabilistes

Algorithme crash-résistant

- Algorithme de Bracha et Toueg
- Opère par rondes pour prendre une décision : 0 ou 1
- A la ronde k, un processus envoie un message de vote à tous, y compris lui-même, puis attend $N - t$ réponses (pas de risque de deadlock)
- A chaque ronde le processus vote soit 0 soit 1, avec un poids associé : le nombre de votes reçus pour cette valeur dans la ronde précédente
- Un vote avec un poids supérieur à $N/2$ est appelé un témoin
- Dans une ronde il ne peut pas y avoir de témoins avec des valeurs différentes
- Si un processus reçoit un témoin dans une ronde k, il vote pour cette valeur à la ronde $k + 1$, sinon il vote pour la majorité
- Un processus qui a pris une décision arrête sa boucle et diffuse un témoin pour les deux rondes suivantes pour aider les autres à décider

Algorithmes de consensus probabilistes

Var : $value_p$: (0,1) **init** x_p /* vote de p */

$round_p$: entier **init** 0 /* numero de ronde */

$weight_p$: entier **init** 1 /* poids du vote */

$msgs_p[0..1]$: entier **init** 0 /* votes reçus */

$witness_p[0..1]$: entier **init** 0 /* witness reçus */

Algorithmes de consensus probabilistes

début
  tant que $y_p = b$ **faire**
    $witness_p[0] = 0; witness_p[1] = 0; msgs_p[0] = 0; msgs_p[1] = 0$
    $\forall q$ send$\langle vote, round_p, value_p, weight_p \rangle$ à q
    tant que $msgs_p[0] + msgs_p[1] < N - t$ **faire**
      receive$\langle vote, r, v, w \rangle$
      si $r > round_p$ **alors**
        send$\langle vote, r, v, w \rangle$ à p
      sinon
        si $r = round_p$ **alors**
          $msgs_p[v] = msgs_p[v] + 1$
          si $w > N/2$ **alors** $witness_p[v] = witness_p[v] + 1$
        sinon
          skip

Algorithmes de consensus probabilistes

début

  tant que $y_p = b$ **faire**

    ...

    si $witness_p[0] > 0$ **alors** $value_p = 0$

    sinon si $witness_p[1] > 0$ **alors** $value_p = 1$

    sinon si $msgs_p[0] > msgs_p[1]$ **alors** $value_p = 0$

    sinon $value_p = 1$

    $weight_p = \#msgs_p[value_p]$

    si $witness_p[value_p] > t$ **alors** $y_p = value_p$

    $round_p = round_p + 1$

  $\forall q$ $\texttt{send}\langle vote, round_p, value_p, N - t\rangle$ à q

  $\forall q$ $\texttt{send}\langle vote, round_p + 1, value_p, N - t\rangle$ à q

Algorithmes de consensus probabilistes

Exercice

- Dérouler l'algorithme pour 4 processus dont un crash
- Combien de crashs supporte l'algorithme ?

Algorithme de consensus Paxos

Objectif

- Définir une valeur commune à un ensemble de processus
- Supporter f fautes à condition d'avoir assez de processus
- Algorithme prouvé :
  - Safe + Live : systèmes synchrones
  - Safe : systèmes asynchrones
- Fautes franches (non-byzantines)

Présentation

- Proposé par L. Lamport en 1990
- Utilisé dans les systèmes de bases de données répliquées (Systèmes WEB) par exemple pour traiter les requêtes dans le même ordre

Algorithme de consensus Paxos

Propriétés sur la valeur de consensus

- Seulement une valeur qui a été proposée peut être choisie (non-trivilaité)
- Une seule valeur est choisie (consistence)
- Un processus (learner) n'apprend jamais qu'une valeur a été choisie si elle ne l'a pas été (conservatisme)

Hypothèses sur le système

- Les processus travaillent à une vitesse quelconque, peuvent s'arrêter et redémarrer
- Les processus peuvent mémoriser de manière persistante les données liées à l'algorithme
- Les messages peuvent être délivrés au bout d'un temps inifimment long, peuvent être dupliqués ou peuvent être

Algorithme de consensus Paxos

Les rôles

Paxos définit trois rôles :

- **Proposers** : proposent les valeurs sur lesquelles le consensus doit être trouvé
- **Acceptors** : décident des valeurs acceptées
- **Learners** : collectent les valeurs obtenues par consensus

Mise en œuvre des rôles

- Il faut des processus qui assurent chacun des rôles
- Le nombre de processus qui assurent les rôles de *proposers* de *learners* est quelconque, mais il faut plus d'un learner pour que cela présente un intérêt
- Le nombre d'*acceptors* doit être au moins de $2f + 1$ pour tolérer f fautes

Algorithme de consensus Paxos

Données

- Les *proposers* génèrent des suites de valeurs S_n différentes entre tous les *proposers* (suites d'entiers différents)
- Les *acceptors* possèdent deux variables persitantes pour mémoriser les propositions de valeurs qui lui ont été faites : le dernièr numéro de séquence proposé S_i (initialisée à 0) et une valeur associée AV_i (initialisée à `nil`)

Messages

- `PREPARE` : proposition d'une valeur
- `OK/OK` : acceptation/refus d'une valeur
- `NOTIFY` : information des valeurs acceptées
- `CHOSEN` : information des valeurs choisies

Algorithme de consensus Paxos

Principe : phase 1

- Les *proposers* envoient à tous les *acceptors* une requête `PREPARE` avec un numéro de séquence unique n
- À la réception d'un message `PREPARE` il y a deux cas :
  - L'*acceptor* n'a jamais reçu de valeur ($AV_i = nill$) et il doit toujours accepter la première valeur qu'il reçoit
  - Un *acceptor* ne peux accepter qu'une proposition avec un numéro de séquence plus grand que S_i
  - Si le numéro de séquence n reç u est plus petit que sa valeur S_i, l'*acceptor* peut soit ingnorer la proposition (il n'y répond pas), soit la réfuser (il envoie un messaage `KO` avec n)
- Il mémorise les valeurs acceptées sur stockage persistant
- Chaque acceptor retourne au *proposer* la dernière valeur AV_i acceptée

Algorithme de consensus Paxos

Principe : phase 2

- Une valeur est choisie si elle est acceptées par une majorité d'*acceptors* : le *proposer* a reçu une majorité $(F + 1)$ de réponse `OK`.
- Si une majorité d'*acceptors* retournent des valeurs AV_i à `nil` alors le *proposer* peut choisir la valeur de son choix, en générale une nouvelle valeur,
- Si une majorité d'*acceptors* retrounent des valeurs AV_i différentes de `nil`, alors le *proposer* prend la valeur avec le plus grand numéro de séquence n'
- Le *proposer* envoie un message `NOTIFY` aux *acceptors* avec les valeurs choisies (valeur de séquence n' et valeur associée AV).

Algorithme de consensus Paxos

Principe : phase 2

- À la récepetion du message `NOTIFY`, les *acceptors* enregistrent le couple (n', AV)
- Ils informent les *learners* que la valeur AV a été choisie avec un message `CHOSEN` qui contient cette valeur.
- Si un *learner* reçoit une majorité de messages `CHOSEN` pour la valeur AV alors il est sûr que cette valeur a été choisie.

Principe : fin

- Valeur acceptée pour cette instance
- Remettre les valeurs de AV à *nil*
- Commencer une nouvelle instance pour choisir une nouvelle valeur

Algorithme de consensus Paxos

Nombre de *proposers*

- Le nombre de *proposers* est quelconque dans l'algorithme d'origine
- Si il y a plus d'un *proposer* l'algorithme n'est pas garanti pour les réseaux asynchrones : deux *proposers* peuvent faire des propositions chacun leur tour en augmentant le nuémro de séquence pour gagner $\rightarrow$ sans fin
- Élire un *proposer* maître parmi les *proposers*, sans garantie dans les réseaux asynchrones
- Les *proposers* esclaves envoient leur propositions au *proposer* maître

Algorithme de consensus Paxos

Nombre d' *acceptors*

- Suffisant pour tolérer les pannes $(2F + 1)$
- Augmente le nombre de messages

Nombre de *learners*

- Le nombre de *learners* est quelconque dans l'algorithme d'origine
- Élire un *learner* maître parmi les *learners* pour limiter le nombre de messages envoyé par les *acceptors* $(a * l)$
- Le *learner* maître diffuse les valeurs acceptées au *learners* esclave

Algorithme de consensus Paxos

Applications à un ensemble de serveurs

- Ensemble de serveurs répliqués (ex : BD)
- Problème : traiter toutes les requêtes dans le même ordre
- Tous les serveurs sont *proposer*, *acceptor* et *leraner*
- Élire un *proposer* maître parmi les serveurs et un *learner* maître, pas forcément les mêmes
- Quand une requête arrive, la donner au *proposer* maître
- *Proposer* maître soumet aux *acceptors*
- Si grand nombre d'acceptors prendre un sous-ensemble
- Faire des instances de Paxos pour chaque valeur proposée
- Quand une valeur est choisie le serveur réalise la requête

Sommaire

Utilisation de la mémoire stable

Les différentes approches

- **Les processus enregistrent des points de reprise de façon asynchrone et indépendamment les uns des autres**. En cas de défaillance, on doit chercher parmi les snapshots locaux un ensemble qui représente un état global cohérent (coupure constistante)
- **L'enregistrement des points de reprise est pré-programmé** afin de générer un ensemble de points de reprise correspondant à un état global cohérent
- **On assure une coordination dynamique entre les actions d'enregistrement de points de reprise**, de telle façon que l'ensemble de points de reprise représente un état global cohérent (algorithmes de snapshot)

Utilisation de la mémoire stable

Cas asynchrone

On part des snapshots locaux et on essaie de trouver une coupure constitante :

- Trouver des états où les liens entre les processus sont vides : $\forall p, q : sent_{pq} \backslash rcvd_{pq} = \emptyset$
- Trouver les états maximaux e_p^i
- On part des derniers évènements enregistrés et chaque processus “recule” jusqu’à trouver un évènement pour lequel tous les messages ont été reçus
- Algorithme pas tolérant aux pannes
- Peut conduire à retourner à l’état initial

Duplication de processus

Principe

- Duplication d'un processus, une ou plus réplique
- Utilisation de la copie en cas de perte du maître
- Peut-être utilisée en cas de crash ou faute byzantine
- Pas d'interférence avec l'algorithme de base

3 techniques différentes

- La duplication passive
- La duplication semi-active
- La duplication active

La duplication passive

Fonctionnement

- Une copie primaire, N copies secondaires (ou de secours)
- Seule la copie primaire
  - traite les messages d'entrée
  - fournit les messages en sortie
- En l'absence de fautes, les autres copies ne traitent pas les messages en entrée
- L'état interne des copies secondaires maintenu/géré par la copie primaire

La duplication passive

Mise à jour des copies secondaires

- Soit, la copie primaire leur envoie périodiquement ses propres points de reprise
- Soit à chaque requête client, la copie primaire envoie les modifications aux copies secondaires

Utilisation de la communication de groupe

- Les copies secondaires font partie d'un même groupe
- La copie primaire diffuse de façon fiable ses points de reprise sur le groupe : toutes les copies reçoivent les messages ou aucune

La duplication passive

Détection de la défaillance de la copie primaire

La duplication passive doit offrir un mécanisme permettant :

- de s'assurer que la copie primaire est toujours opérationnelle
- de déclencher, en cas de défaillance, le protocole d'élection d'une nouvelle copie primaire
- aux clients de connaître la nouvelle copie primaire

La duplication passive

Défaillance de la copie primaire

Pour savoir si la copie primaire fonctionne :

- Envois de requêtes de type « es-tu vivant ? » (heart beat) à la copie primaire
- Si non-réponse : copie primaire considérée comme défaillante

Requêtes envoyées par :

- par les copies secondaires,
- par un service indépendant
- par les clients

Fréquence des requêtes détermine la rapidité de recouvrement du groupe de serveurs en cas de défaillance

La duplication passive

Défaillance de la copie primaire

Quand défaillance de la copie primaire est détectée :
Solution 1 :

- Un protocole d'élection permet de choisir une nouvelle copie primaire parmi les copies secondaires
- La copie élue effectue un recouvrement à partir du point de reprise le plus récent

Solution 2 :

- L'ordre dans lequel les copies secondaires pourront devenir primaires est déterminé à l'avance
- Évite une élection
- Cas où plusieurs crashs

La duplication passive

Défaillance d'une copie secondaire

Les copies secondaires qui ne reçoivent pas les points de reprise ne doivent pas pouvoir devenir des copies primaires
Comment cela est-il réalisé et par qui ?

- Soit par le protocole d'élection : il doit savoir identifier les copies dont l'état interne est obsolète
- Soit par la copie primaire : elle exclut du groupe toute copie secondaire qui ne répond pas à un envoi de point de reprise

La duplication passive

Commentaires

Technique relativement facile à mettre en oeuvre.

- Pas de mécanisme particulier de filtrage et « collage » des réponses pour les requêtes
- Elle tolère un certain degré de non-déterminisme dans le fonctionnement des copies
- Si la copie primaire devient un goulet d'étranglement pour l'ensemble du système, il est possible de :
  - Avoir plusieurs copies qui peuvent agir en tant que copies primaires vis-à-vis de leurs clients
  - Déléguer les requêtes en « lecture » qui peuvent être traitées par les copies secondaires pourvu que leur état interne soit à jour

La duplication passive

Commentaires

La fréquence d'envoi et de traitement des points de reprise :

- Coûteuse en termes de ressources
- Pénalise le traitement normal des requêtes envoyées par les clients
- Si la probabilité de défaillance de la copie primaire est faible, il est possible d'espacer les points de reprise.
- Cela peut avoir une incidence sur les clients :
  - Doivent être capables d'effectuer un recouvrement de leur propre état par rapport au serveur défaillant
  - Doivent avoir la capacité de re-soumettre un certain nombre de requêtes et de re-évaluer les réponses correspondantes

La duplication passive

Commentaires

Trouver le bon compromis entre :

- Les besoins de l'application en termes de fiabilité et de performances
- La complexité des processus clients et serveurs
- Les caractéristiques du système distribué

Propriétés

- Permet de gérer les fautes par arrêt, coupures réseaux mais pas les fautes byzantines et fautes par valeur
- Similitude entre la duplication passive et les techniques à base de mémoire stable

La duplication active

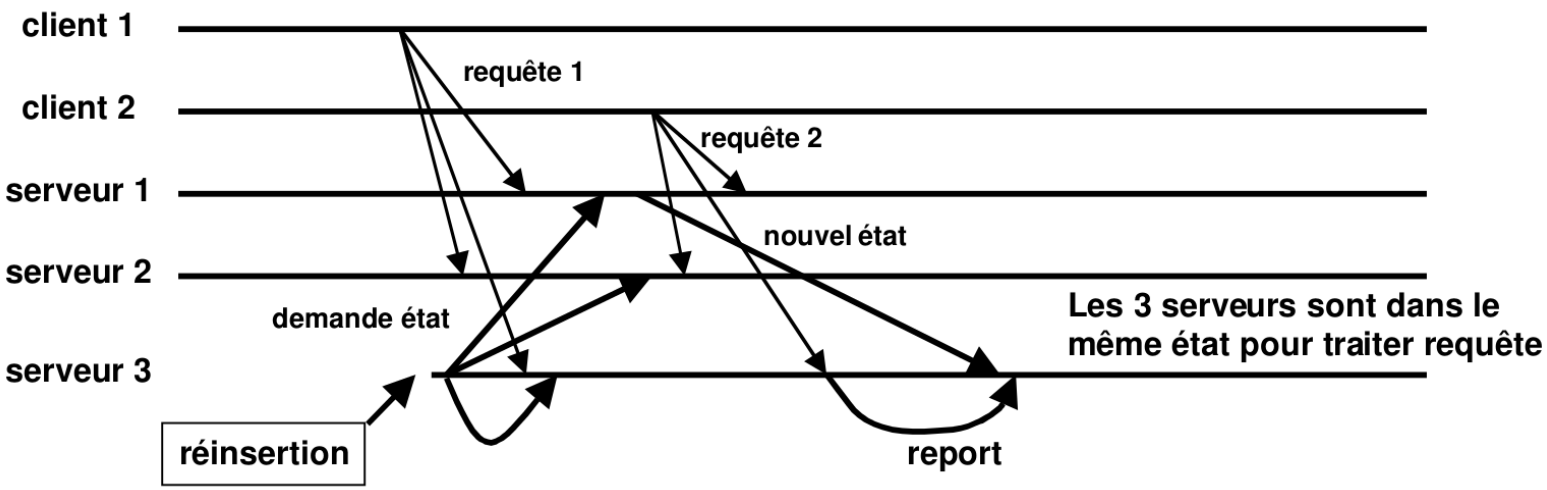
Fonctionnement

- Serveur déterministe :
  - Réponses déterminées par séquence de requêtes
  - Comportement du serveur est indépendant du temps physique
- Tous les processus en état de marche traitent toutes les requêtes
- Toutes les requêtes sont traitées dans le même ordre (abcast)
- Réponse au client :
  - Une seule réponse est envoyée au client (consensus)
  - Le client ne prend en compte qu'une seule réponse

La duplication active

Ré-insertition après panne

Diffusion d'une demande de reconstruction d'état



La duplication semi-active

Fonctionnement

- Technique hybride : mi-chemin entre la duplication active et la duplication passive
- Messages d'entrée diffusés à l'ensemble des copies

La duplication semi-active

Fonctionnement

- Une seule copie, la copie leader :
  - traite tous les messages
  - produit les résultats en sortie
- Les autres copies, les copies suiveuses (follower)
  - ne produisent pas de résultats en sortie
  - utilisent les messages en entrée pour mettre à jour leur état interne ou « notifications » en provenance de la copie leader

La duplication semi-active

Défaillance de la copie leader

- Un protocole d'élection permet de choisir une nouvelle copie leader parmi les copies suiveuses
- La copie élue n'a pas besoin d'effectuer un recouvrement
- Différence par rapport à la duplication passive ?

Références

- "Tolérance aux fautes : principes et mécanismes" de JB Setfani (INRIA) et S Krakowiak (Université J Fourier)
- "Tolérance aux fautes 1 et 2" de S.Krakowiak (Université J Fourier)
- "Systèmes et applications répartis : sûreté de fonctionnement" Cnam-Cedric
- "Contribution à l'intégration de la tolérance aux fautes dans les environnements répartis" de P.Sens (HDR Université de Paris 6)