

# ZooKeeper

Un outil de coordination de systèmes distribués

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

DataCloud - Master 2 SAR 2020/2021



*"Because Coordinating Distributed Systems is a Zoo"*, Apache Foundation

sources :

<https://zookeeper.apache.org/>

ZooKeeper : Distributed Process Coordination, Flavio Junqueira, O'Reilly Media, ISBN : 978-1-449-36130-3

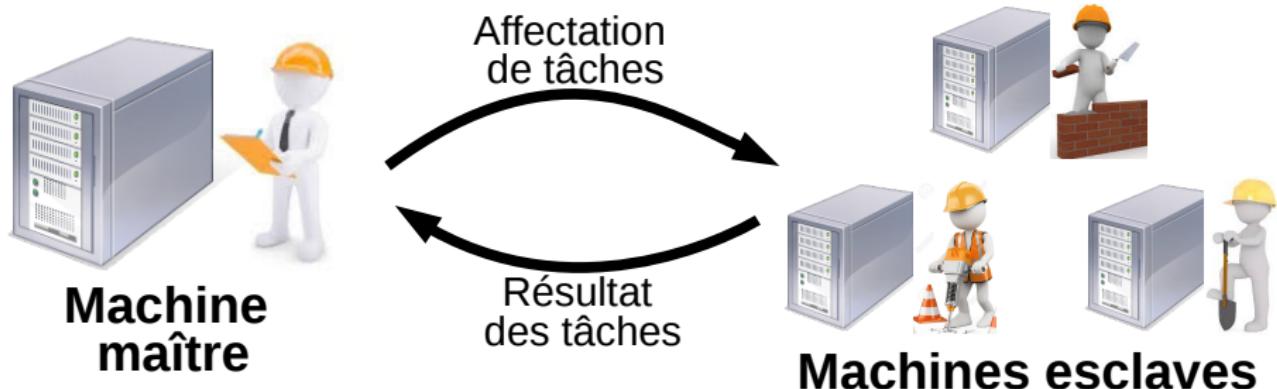
## Définition

Ensemble d'unités de calcul autonomes communiquant via un réseau.

## Une coordination difficile

- Pas de mémoire partagée  
    ⇒ pas de vision globale
- Pas d'horloge globale  
    ⇒ Difficile d'ordonner des événements
- Réseau non fiable  
    ⇒ perte/duplication de message
- Unités de calcul non fiables  
    ⇒ panne possible

# Exemple : un système distribué maître esclave

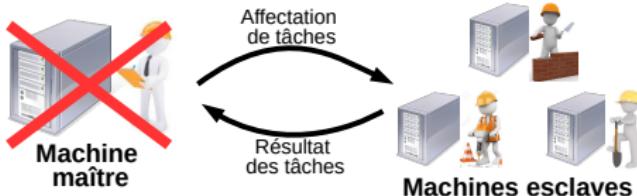


## Problématique

Comment tolérer les pannes

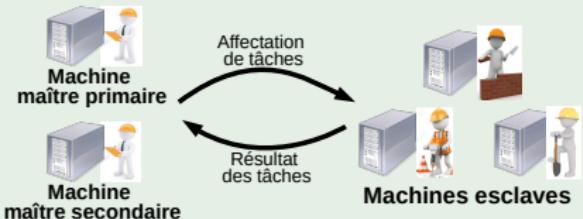
- de la machine maître
- d'une machine esclave
- du réseau

# Exemple : panne du maître



## Une solution

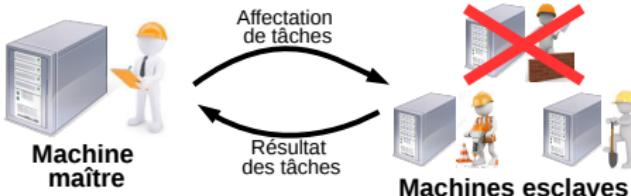
Un maître secondaire qui prend la place du maître primaire en cas de panne



## To do

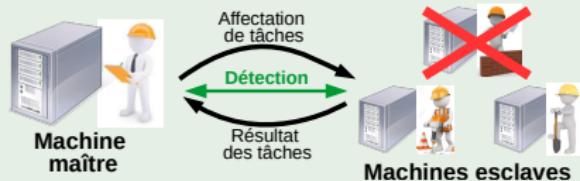
- Déployer un support de sauvegarde stable partagé entre les 2 maîtres
- Implanter un mécanisme **fiable** de détection de panne du maître primaire

# Exemple : panne d'un esclave



## Une solution

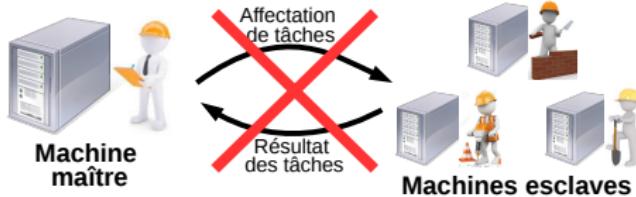
Le maître peut détecter les pannes des esclaves et réaffecter les tâches perdues



## To do

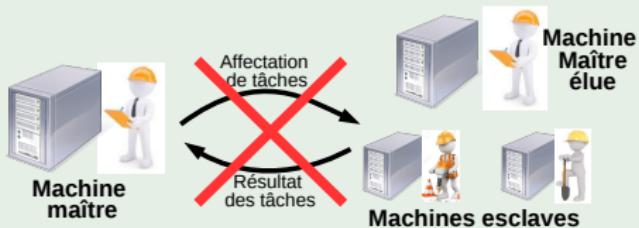
- Modifier le protocole maître-esclaves

# Exemple : panne du réseau



## Une solution

Les workers peuvent élire un nouveau master



## To do

- Implanter un mécanisme qui détecte la déconnexion avec le master
- Implanter un mécanisme d'élection de leader

## Exemple : résumé

### To Do (pour chaque système à concevoir)

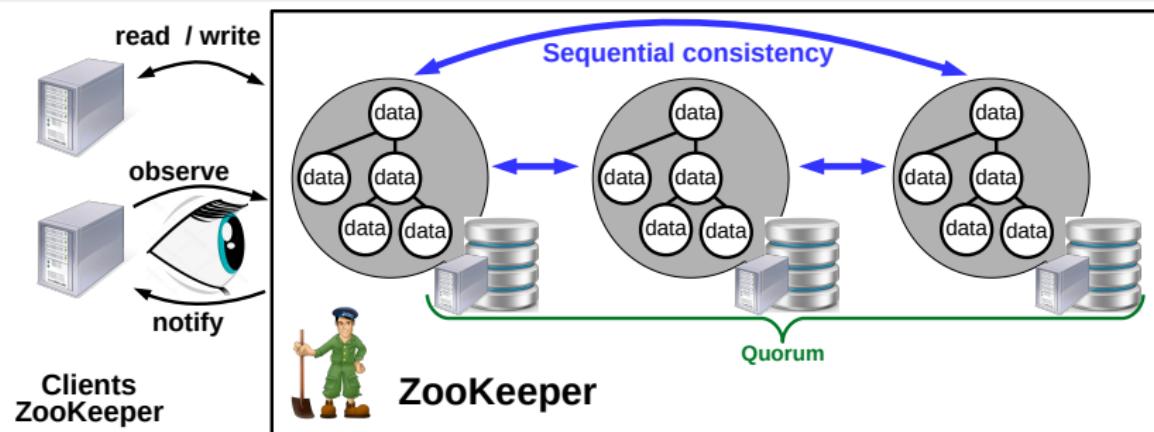
- Mécanisme d'élection de leader
- Déetecter les pannes/les nouvelles machines
- Stockage des méta-données du système
- Mécanisme de synchronisation fiable

**Besoin d'un système fiable offrant ces services**

# Zookeeper : Qu'est ce que c'est ?

Un entrepôt de données développé en Java

- arborescent  $\Rightarrow$  organisation hiérarchique des données
- persistant  $\Rightarrow$  données sauvegardées et restaurables
- répliqué  $\Rightarrow$  données toujours accessible même en cas de défaillance
- séquentiellement cohérent  $\Rightarrow$  ordre total sur les écritures
- observable  $\Rightarrow$  notification des changements aux clients



# Objectifs de Zookeeper

## Ce qu'il permet de faire

- Stocker des méta-données
- Serveur bootstrap
- Offrir un service de nommage (annuaire)
- Implanter des mécanismes de coordination de processus distribués à travers une API simple
  - élection de leader
  - supervision
  - synchronisation (verrou, barrière, ...)
  - diffusion de messages

## Ce qu'il ne fait pas

- Stockage de données massives ou relationnelles
- Exposer une API haut niveau de coordination

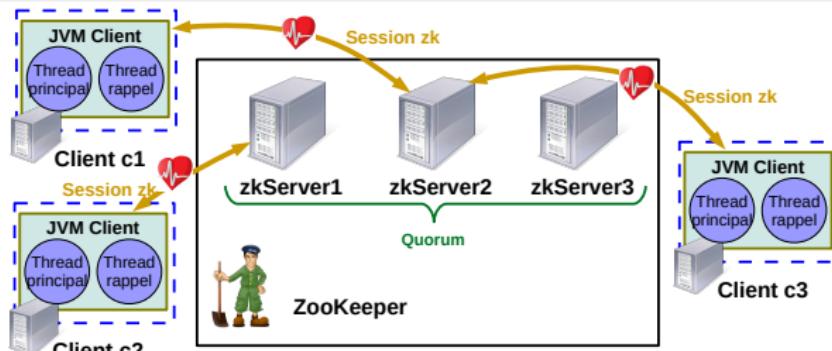
## Où est-il utilisé ?

- Logiciels de la fondation Apache
  - Hbase (SGBD) : supervision, stockage de méta-données
  - Kafka (Pub-Sub) : détection de crash, découvertes, sauvegarde d'états
  - Solr (moteur de recherche) : stockage/mise à jour de méta-données
- Yahoo ! : élection de leader, détection de crash, stockage de méta-données
- Facebook Messages : découverte de services, partitionnement de données

# Interaction système-client

## Caractéristiques

- Le client doit établir une connexion et maintenir une session
- Toute opération du système est associée à une session cliente
- La connexion s'établit sur un seul serveur choisi arbitrairement
- Une session implique deux threads :
  - le thread principal du programme
  - un thread pour les rappel/notification de Zookeeper
- Heartbeats réguliers pour maintenir la session



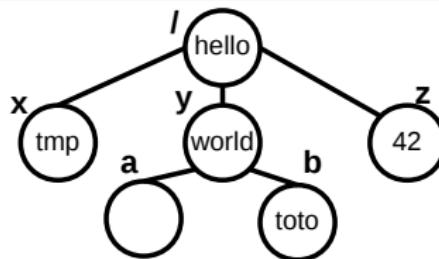
# Un modèle de données clé-valeur

## Les znodes (= clés)

- Associé à une valeur modifiable
- Organisation en arbre → un znode possède :
  - un znode père
  - un ensemble de znodes fils
- Identifié par un chemin absolu à partir d'un znode racine

## Les valeurs

- non typée → tableau d'octets
- limitée à 1 Mo



# Opérations sur les znodes

## Opérations d'écriture

- ***set*** : modifier la valeur d'un znode
- ***create*** : créer un znode à partir d'un znode père existant
- ***delete*** : effacer un znode (precond : pas de fils)

## Opérations de lecture

- ***get*** : lire la valeur d'un znode
- ***ls*** ou ***getChildren*** : lister les noms des fils d'un znode
- ***exists*** : tester l'existence d'un znode

## Remarques

- Il existe une version synchrone et asynchrone de chaque opération.
- Depuis la v3.4, possibilité d'exécuter plusieurs opérations dans un bloc atomique ⇒ Transactions multi-opérations

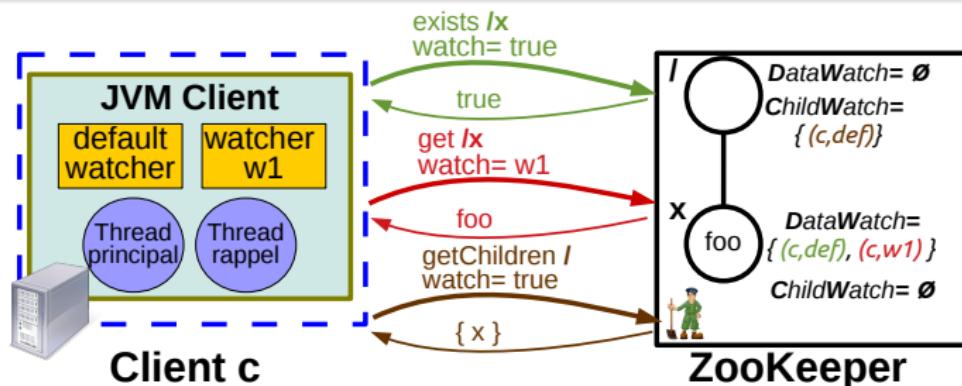
# Surveiller l'état d'un znode

## Watch

- Abonnement auprès du système pour être notifier uniquement de la prochaine modification sur un znode donné.
- Son enregistrement se fait lors d'une opération de lecture

## Watcher

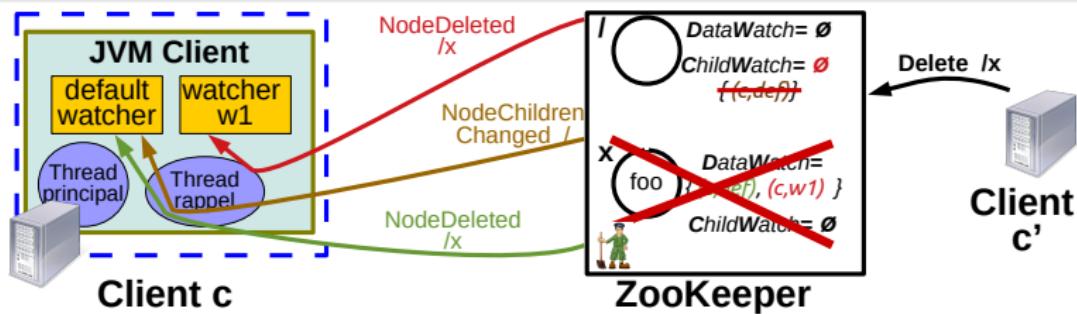
Classe instanciée coté client contenant le code à exécuter lors du changement d'état du znode surveillé (callback).



# Les notifications

## Les types de notification

- **NodeCreated** : création d'un znode
- **NodeDataChanged** : changement des données du znode
- **NodeDeleted** : suppression d'un znode
- **NodeChildrenChanged** : ajout ou suppression d'un fils



## Attention

Le nouvel état du znode n'est pas transmis dans le message de la notification :  $\Rightarrow$  une opération de lecture reste nécessaire

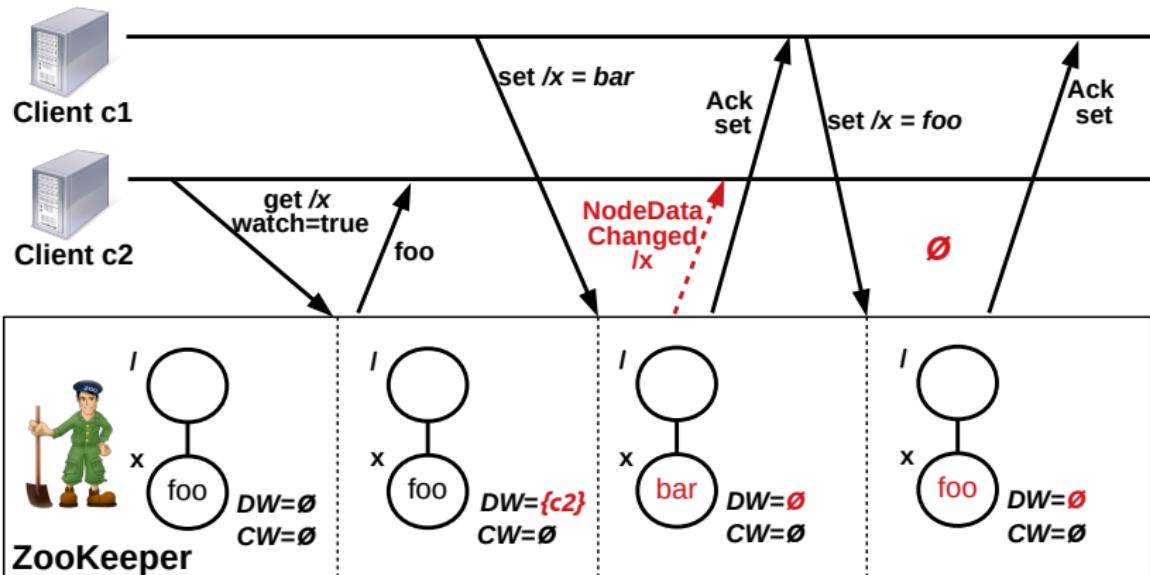
# Type de Watch et notifications associées

Type de watch	Data watch		Child watch
Définition	surveiller l'état d'un znode		surveiller l'état de l'ensemble des fils d'un znode
Opérations d'abonnement	<i>get</i>	<i>exists</i>	<i>ls/getChildren</i>
Notifications possibles	<i>NodeDataChanged</i> <i>NodeDeleted</i>	<i>NodeCreated</i> <i>NodeDataChanged</i> <i>NodeDeleted</i>	<i>NodeChildrenChanged</i>

## Remarque

La création/suppression d'un znode peut entraîner à la fois un *NodeCreated*/*NodeDeleted* et un *NodeChildrenChanged*.

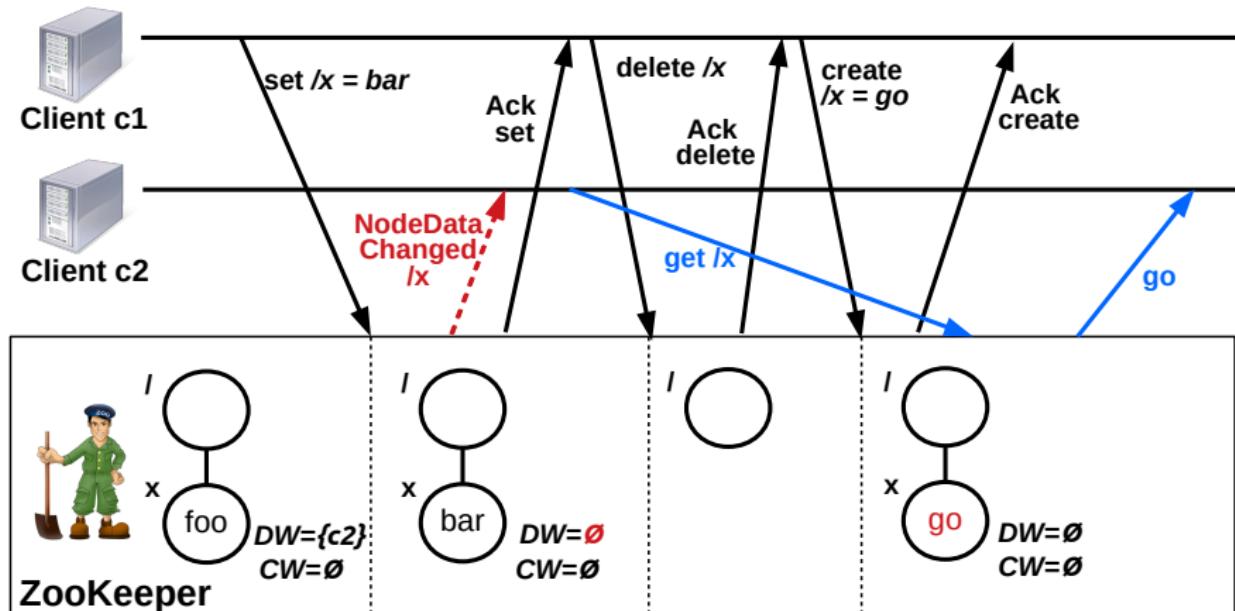
# Pièges des watch



## Une seule notification par watch

- Dès que le client reçoit l'événement, le watch n'est plus valable
- L'enregistrement d'un nouveau watch est nécessaire pour recevoir les notifications suivantes

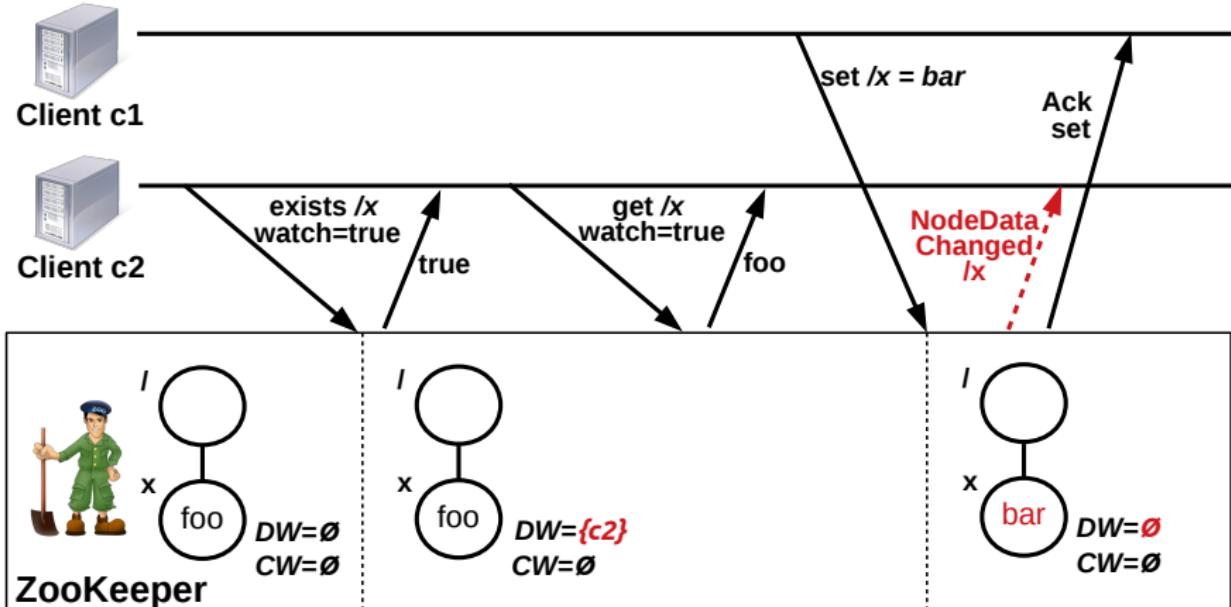
# Pièges des watch



Impossible de voir toutes les modifications

Plusieurs changements peuvent survenir entre la notification de l'événement et le réabonnement.

# Pièges des watch



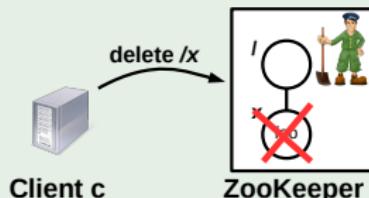
**Impossible d'anticiper les notifications futures**

Au plus un abonnement du même type d'un même client sur un même watcher peut être enregistré sur un znode.

# Les types de znodes

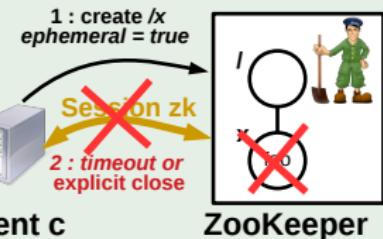
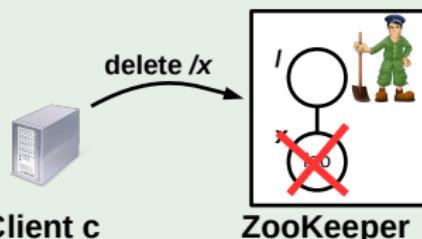
## Znode persistant (type par défaut)

Znode supprimé uniquement par une requête explicite de suppression.



## Znode éphémère

Znode supprimé soit par une requête explicite de suppression soit par la fin de session du client qui l'a créé. **⇒ Ne peuvent pas avoir de fils**

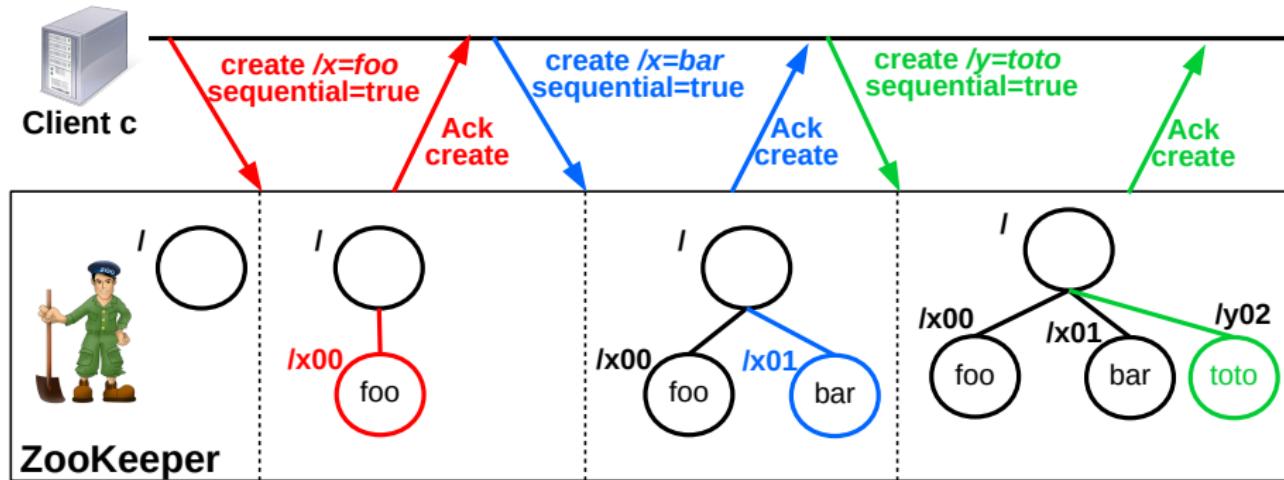


# Nommage séquentiel des znodes

## Caractéristiques

Ajout d'un suffixe entier au nom du znode au moment de sa création :

- Le suffixe se calcule à partir d'un compteur incrémental et atomique
- Le système maintient un compteur par znode parent.



# Les méta-données sur les znodes

## Identifiants de transaction

- *cZid* : transaction qui a créé le znode
- *mZid* : dernière transaction qui a modifié le znode
- *pZid* : transaction qui a créé le dernier znode fils

## Timestamp et compteurs

- *ctime* : date à laquelle le znode a été créé
- *mtime* : date de la dernière transaction sur le znode
- *version* : nombre de fois où la donnée a été modifiée
- *cversion* : nombre de fois où l'ensemble de fils a été modifié
- *aclVersion* : nombre de fois où les droits d'accès ont été changés

## Autres

- *dataLength* : taille des données du znode
- *numChildren* : nombre de fils

## Cohérence séquentielle

Toute opération d'écriture respecte un ordre total :

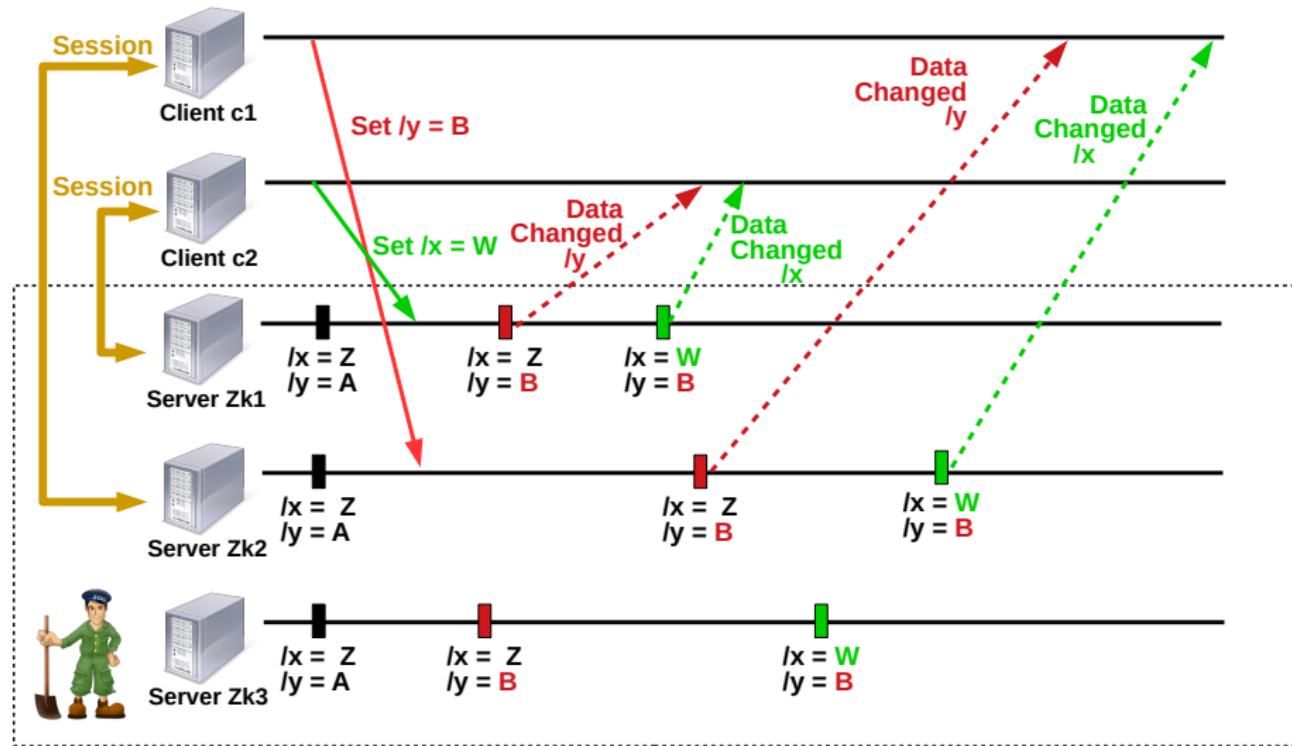
⇒ **Tous les serveurs voient les modifications dans le même ordre**

## Procédure de traitement d'une opération d'écriture

- ① Attribution d'un identifiant unique de transaction (zxid)
- ② Propagation de l'écriture à l'ensemble du quorum + maj méta-données
- ③ Envois des notifications aux éventuels clients abonnés

# Garantie d'ordre : schéma de deux écritures concurrentes

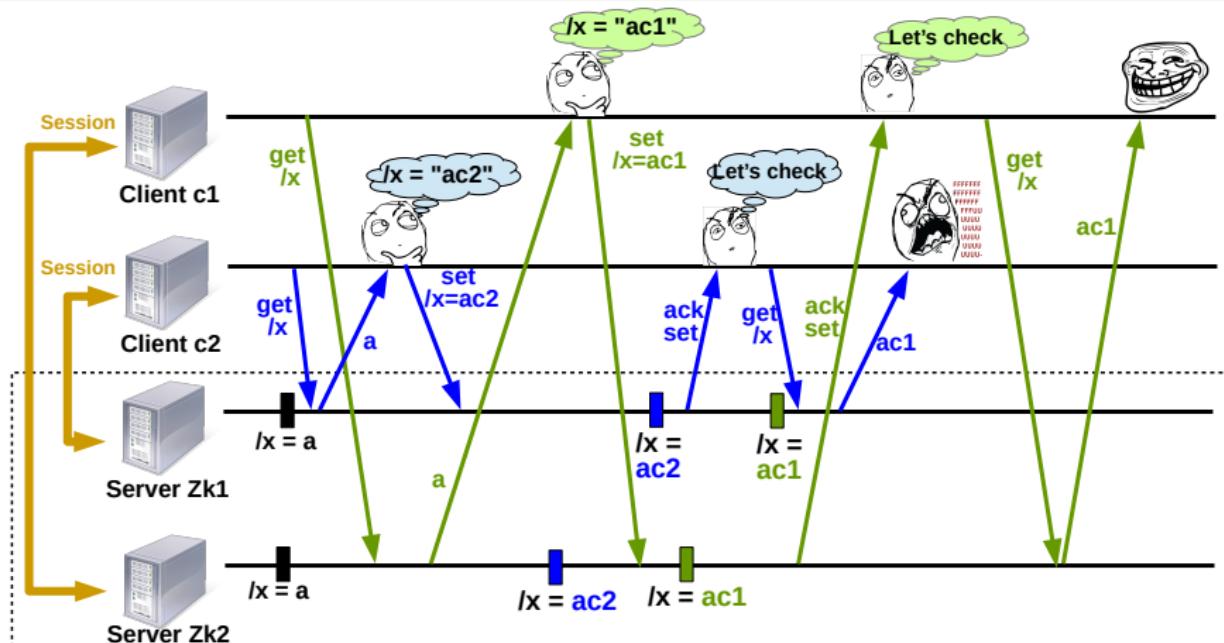
Hypothèse :  $C_1$  et  $C_2$  ont un watch sur  $/x$  et  $/y$



# Cas des écritures concurrentes conflictuelles

## Problématiques

- Comment gérer les modifications concurrentes sur le même znode ?
- Comment assurer qu'un client modifie toujours le dernier état ?

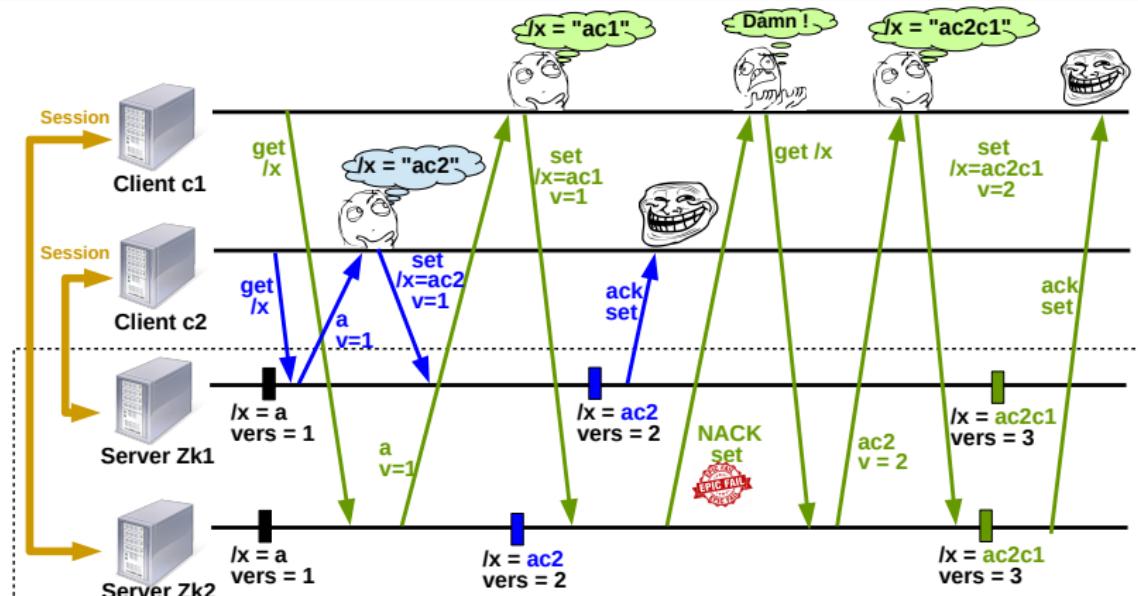


# Cas des écritures concurrentes conflictuelles

## Solution

Les appels *set* et *delete* prennent en argument num de version attendu :

- si argument = version courante du znode , l'opération réussie
- sinon elle échoue



# Exemple de synchronisation

## Codage d'un verrou sans vivacité

```
Lock()
begin
    cs ← false;
    while not cs do
        call create node /lock ;
        if creation succeeded then
            cs ← true;
        else
            call exists /lock and watch=true;
            if /lock exists then
                wait NodeDeleted event for /lock;
```

```
Unlock()
begin
    call delete /lock
```

Que faire pour assurer la vivacité ?

# Exemple de synchronisation

## Une solution non optimisée

**Lock()**

**begin**

$cs \leftarrow \text{false};$

$myZnode \leftarrow \text{call create ephemeral and sequential node } /lock;$   
  **while** not  $cs$  **do**

$children \leftarrow \text{call getChildren of } /, \text{watch}=\text{true};$   
    **if**  $myZnode$  has the smallest id in  $children$

**then**

$cs \leftarrow \text{true};$

**else**

**wait** NodeChildrenChanged event for  $/$ ;

**Unlock()**

**begin**

**call** delete  
   $myZnode$

Que faire pour optimiser les communications ?

# Exemple de synchronisation

## Une meilleure solution

```
Lock()
begin
    cs ← false;
    myZnode ← call create ephemeral and sequential node
    /lock ;
    while not cs do
        children ← call getChildren of /,watch=false;
        if myZnode has the smallest id in children then
            cs ← true;
        else
            call exists my predecessor in children,watch=true;
            if my predecessor exists then
                wait NodeDeleted event for my predecessor;
```

```
Unlock()
begin
    call delete
    myZnode
```

# Mécanismes internes : rôle des différents serveurs

## Types de serveur



**le leader** : serveur central pour séquencer toutes les transactions

⇒ désigné par une élection de l'ensemble des serveurs



**les followers** : votent les transactions proposées par le leader



**les observers** : serveurs qui prennent juste acte des décisions mais qui ne prennent pas part aux votes

⇒ optionnels mais utiles pour passer à l'échelle

## Remarque

En production le nombre de serveurs doit être impair et  $\geq 3$ .

## Principes généraux

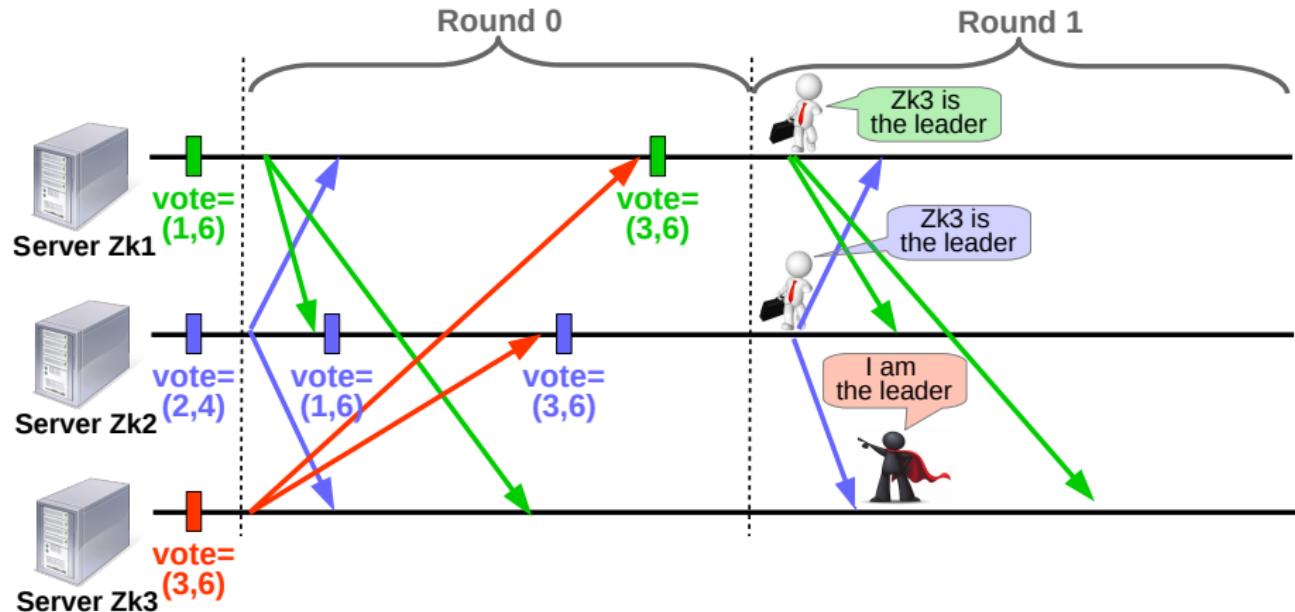
- Une élection se déclenche dès qu'il n'y a pas ou plus de leader
- Chaque serveur possède deux entiers :
  - son **sid** : un identifiant défini statiquement
  - le **zxid** de la transaction la plus récente qu'il a pris en compte
- Le leader doit être le serveur opérationnel ayant le plus grand zxid ou bien le plus grand sid en cas d'égalité de zxid

## Algorithme à base de round

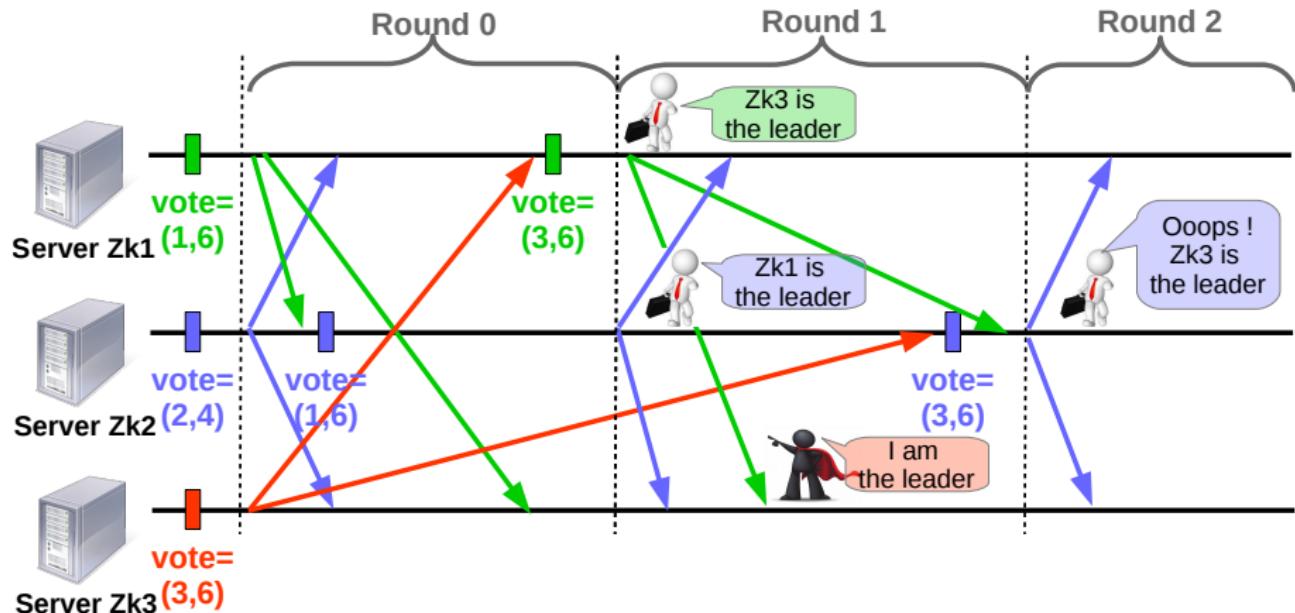
Pour un round  $N$  :

- si  $N = 1$  ou mon vote a changé durant le round  $N - 1$   
Diffuser mon vote à l'ensemble des serveurs
- si j'ai reçu une majorité de vote égaux alors leader = mon vote
- à la réception d'un vote :  
si vote reçu supérieur au mien alors mon vote  $\leftarrow$  vote reçu

# Mécanismes internes : Exemple d'élection sans erreur



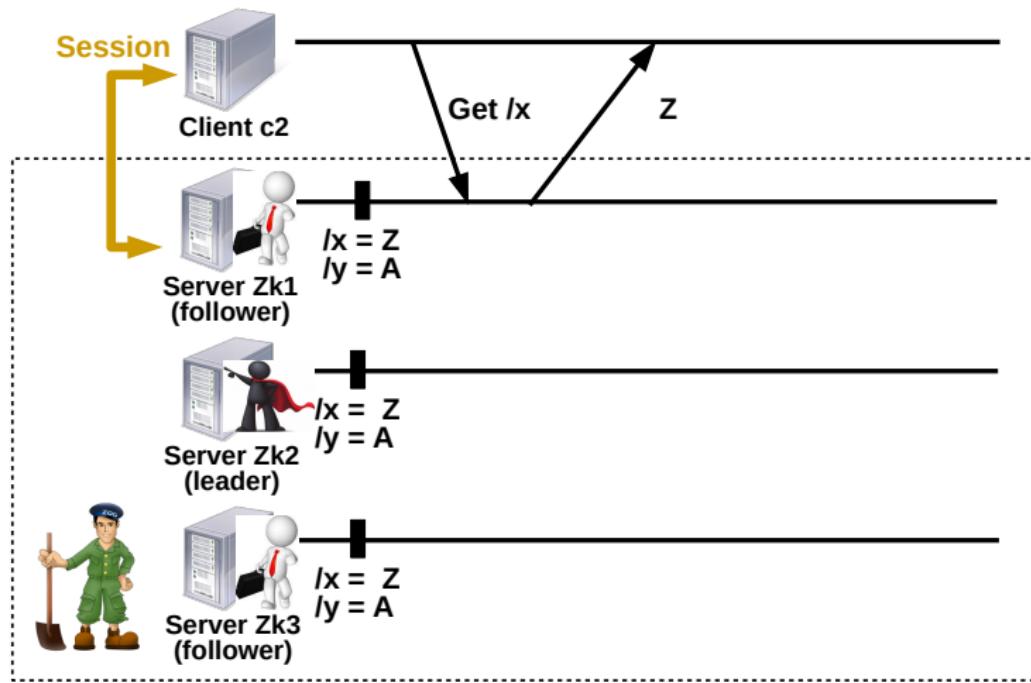
# Mécanismes internes : Exemple d'élection avec erreur



# Mécanismes internes : requête de lecture

## Principe

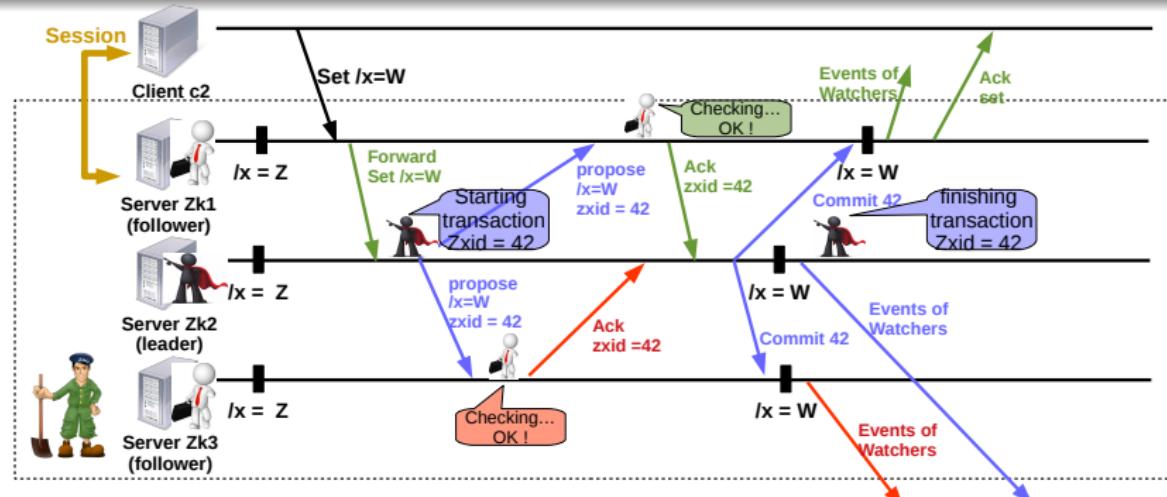
Le serveur renvoie directement la dernière valeur connue localement.



# Mécanismes internes : requête d'écriture (protocole Zab)

## Principe du ZooKeeper Atomic Broadcast

- Toute requête d'écriture est retransmise au leader
- Application de la mise à jour aux autres serveurs via un mécanisme de **commit à deux phases**
- Lors du commit sur un serveur la transaction est atomiquement appliquée et loguée sur le disque local.



# Interagir avec Zookeeper

## Depuis un langage

- Java : langage natif de la plateforme
- C

## Depuis un shell dédié

```
[zk: localhost:2181(CONNECTED) 0] help
ZooKeeper -server host:port cmd args
    stat path [watch]
    set path data [version]
    ls path [watch]
    printwatches on|off
    delete path [version]
    sync path
    rmr path
    get path [watch]
    create [-s] [-e] path data acl
    addauth scheme auth
    quit
    connect host:port
...
...
```