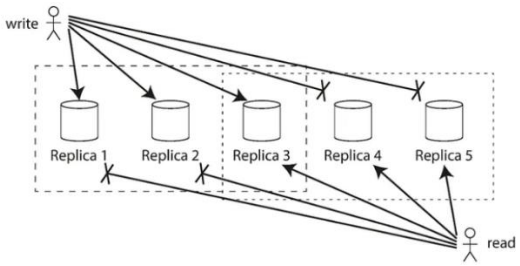
Fondements des Systèmes répartis

TP2 : Réplication dans Systèmes distribués, disponibilité et cohérence

Enseignant : Sofiane Ouni

Background

1. Réplication



La réplication consiste à copier les données sur plusieurs nœuds. Elle offre plusieurs avantages :

- **Disponibilité** : Si un **nœud échoue**, **les autres peuvent continuer** à servir les requêtes.
- **Résilience** : En **cas de panne** matérielle ou de défaillance, les **données restent accessibles**.
- **Performances** : La réplication permet de répartir la charge entre les nœuds.

Il existe différentes stratégies de réplication :

- **Réplication synchrone** : Les données sont **copiées immédiatement** sur tous les nœuds.
- **Réplication asynchrone** : Les données sont **copiées avec un certain délai**, ce qui peut entraîner **une incohérence temporaire**.

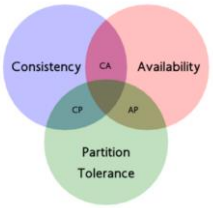
2. La haute disponibilité

La haute disponibilité dans les systèmes distribués est la capacité d'un système informatique à être **accessible et fiable presque 100 % du temps**, réduisant ainsi, voire éliminant, **les temps d'arrêt**. Voici les points clés à retenir :

- Objectif de la haute disponibilité :
  - Assurer que le système fonctionne à son plus **haut niveau de performances**.
  - Garantir que le service ou serveur donné est **accessible sans interruption**.
  - **Maintenir des performances satisfaisantes** sur une période donnée.
- **Détection** et **élimination des points de défaillance** :
  - Une infrastructure à haute disponibilité doit identifier et éliminer les points de défaillance uniques.
  - Un **point de défaillance unique** peut déconnecter tout le système en cas de panne.
  - Les **défaillances** peuvent être **matérielles, logicielles**, liées au **réseau**, etc.
- **Réplication** pour la Haute Disponibilité :
  - La **réplication** est un mécanisme clé pour **atteindre la haute disponibilité.**
  - En répliquant les **données sur plusieurs nœuds,** on garantit que même **si un nœud échoue**, les **autres peuvent continuer** à servir les requêtes.

- La **réplication offre une redondance des données** et permet de maintenir la **disponibilité du service**.

3. la théorie CAP :



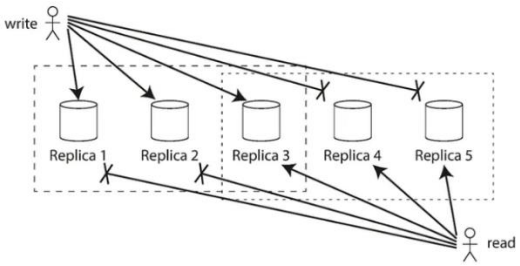
CAP Theorem Explained

Le **théorème CAP** (ou théorème de Brewer) est essentiel pour comprendre les propriétés des **systèmes distribués**. Le théorème **CAP**(**Consistency, Availability, Partition tolerance**) stipule qu'un système distribué ne peut garantir simultanément les trois propriétés.
Voici une explication détaillée :

- **Cohérence (Consistency)** :
  - La cohérence signifie que **tous les clients voient les mêmes données en même temps**, quel que soit le nœud auquel ils se connectent.
  - Pour atteindre la cohérence, chaque fois que des données sont écrites sur un nœud, elles doivent être instantanément transmises ou répliquées vers tous les autres nœuds du système avant que l'écriture soit considérée comme « réussie ».
- **Disponibilité (Availability)** :
  - La disponibilité signifie qu'un client qui effectue une requête de données obtient une réponse, même si un ou plusieurs nœuds sont en panne.
  - En d'autres termes, tous les nœuds actifs du système distribué renvoient une réponse valide à toutes les requêtes, sans exception.
- **Tolérance au partitionnement (Partition Tolérance)** :
  - Un partitionnement est une rupture

Travail demandé

Dans ce TP, nous voudrons réaliser un prototype d'application réparties pour la réplication des données (Base de données, …, ou pour simplifier un Fichier Texte) sur plusieurs machines ou disques (pour simplifier nous allons travailler sur une seule machine et définir à la place un répertoire par réplication ou disque). Nous allons choisir deux stratégies pour la gestion des réplications. La première qui va favoriser la cohérence (consistancy) des données sur les réplications. La deuxième qui va faire un certain compromis entre la disponibilité et la cohérence.

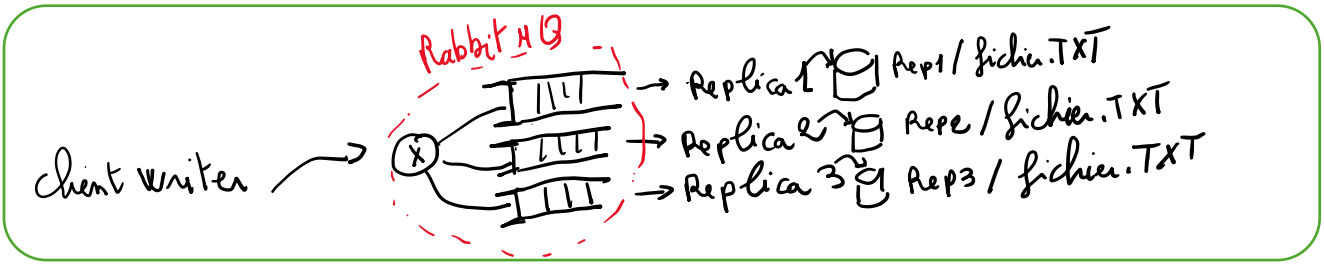


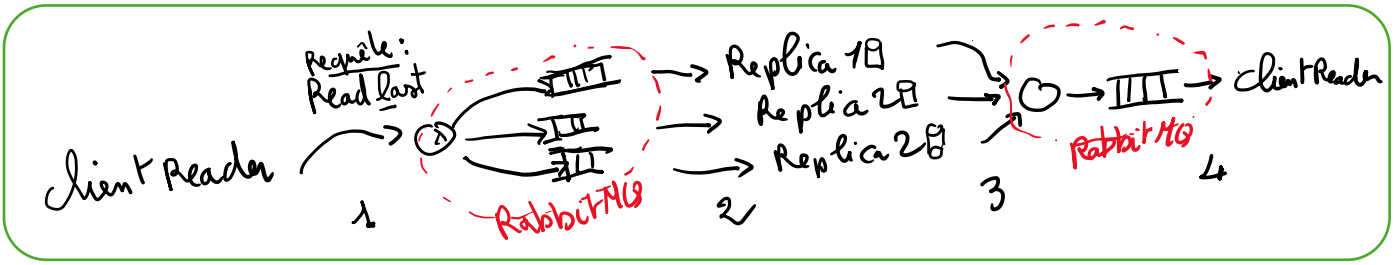
Pour ce TP, nous allons utiliser **RabbitMQ** comme **middleware MoM** pour l'échanges entre les différentes entités (ou processus) de l'application répartie. On suppose également 3 programmes ou processus : Un processus **ClientWriter** (un processus client qui lance des transactions de lignes à ajouter dans les fichiers Textes), un autre processus **ClientReader** (un processus client

qui lit les lignes des fichiers Textes) et un processus **Replica** (un processus qui permet de répondre aux requêtes des clients que se soit pour la lecture ou l'écriture). Dans l'exemple suivant, on définit les deux traitements à savoir l'écriture du client puis la lecture du client.

On suppose que le **fichier texte** à traiter est de la forme suivante (numéro pour la ligne du texte et puis le texte de la ligne) :

```
1    Texte 1…
2    Texte 2….
4    Texte 4….
7    Texte 7…
```





1. Réaliser le processus de **ClientWrite** (programme java) qui permet d'émettre un message d'ajout d'une ligne de texte vers le broker RabbitMQ.
2. Réaliser le processus **Replica** (programme) qui permet de lire de sa file correspondante de RabbitMQ un message d'une ligne à ajouter pour son fichier dans son répertoire correspondant (local). Il y a **3 processus Replica** qui auront le même code (à ne pas copier le code) mais avec un argument qui indique le numéro du processus. On aura alors le lancement de ces programmes en ligne de commande comme suit :
   Java Replica 1
   Java Replica 2
   Java Replica 3
3. Vérifier que le fonctionnement des copies sur les différents réplicas est réalisé correctement.
4. Réaliser le code du processus ClientReader. Ce processus envoi un message de requête **'Read Last'** (cette requête demande l'affichage de la dernière ligne du fichier texte). Puis le client à travers le même processus clientReader attend à la fin les messages des différents Replica. Puis il assure l'affichage.
5. Pour la question 4, il n'est pas nécessaire de lire les réponses de tous les Replica. Pour faire rapidement, on peut se contenter de la première réponse. On peut ici simuler la panne d'un de replica en arrêtant son exécution et puis on réalise la lecture du client pour voir comment la réplication permet d'assurer la disponibilité des données.

6. On va réaliser une simulation d'exécution dans laquelle, le client clientWriter écrit ces deux lignes de données : « 1 Texte message1 » puis « 2 Texte message2 » . Ensuite, nous arrêtons Replica 2 (comme s'il est en panne). Puis le client clientWriter écrit ces deux lignes de données : « 3 Texte message 3 » puis « 4 Texte message4 » . On remet en suite le processus Replica 2 comme s'il a repris son fonctionnement. Vérifier les 3 fichiers des 3 replica . Le problème ici est que les 3 replica n'ont pas le même contenu de fichier Texte.
7. Créer une nouvelle version de ClientReaderV2 (il faut garder l'ancienne version pour l'éventuel validation) qui va faire une requête d'affichage totale du fichier Texte avec un message : '**Read All**' . Puis ce même processus va lire les lignes des 3 fichiers et puis va afficher les lignes qui apparaissent dans la majorité des propositions des 3 replica. Il ne faut pas oublier aussi de mettre une nouvelle version du processus Replica pour prendre en considération cette nouvelle requête du client. Ce processus envoi la totalité du fichier dont chaque message envoyé contient une seule ligne du fichier. Donc, la transmission se fait ligne par ligne.