Architecture Distribuées
Le casse-tête disponibilité-cohérence des données

Les moteurs de gestion de données distribuées sont souvent confrontés à deux aspects principaux, la gestion de la cohérence des données et la gestion de la disponibilité des données. Par exemple, les systèmes NoSQL ne peuvent supporter, à la fois, plus de deux caractéristiques du théorème de CAP. Si la plupart des systèmes assure la tolérance aux pannes, la cohérence et la disponibilité des données ne sont jamais assurées ensemble.

Dans le cadre de ce travail, nous nous intéressons à la mise en place d'une architecture capable de supporter les 3 caractéristiques :

- Cohérence des données
- Disponibilité des données
- Tolérance aux pannes

Le protocole de mise en place couvre les phases suivantes :

- Mise en place de l'architecture (reposant sur plusieurs machines)
- La gestion de la disponibilité
- La gestion de la cohérence et particulièrement l'accès concurrentiel aux données

Organisation du travail :

Les étudiants doivent

- Travailler en groupe
- Monter un réseau local composé de plusieurs machines

-

NB : Le choix du langage, pour le développement des programmes nécessaires aux différents tests, vous est laissé libre.

1. Gestion de la disponibilité

Dans notre solution technique, nous faisons le choix d'utiliser Spark. Ce dernier repose sur une architecture maître-esclave (et donc sur une architecture avec un point de défaillance, le maître). Pour pallier cette limite, nous utilisons Zookeeper (ZK).

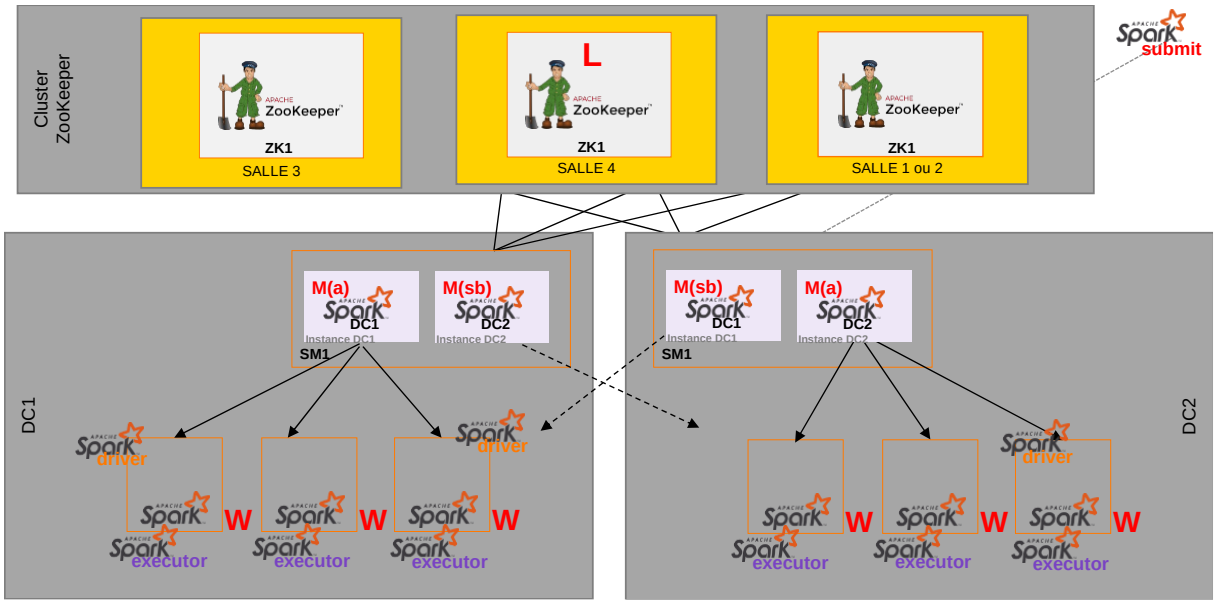


Figure 1 Exemple d'architecture SPARK/ZK en multi-sites

a. Travail à réaliser

- Notre objectif est de mettre en place :
  - Un cluster Spark composé d'un Master et de plusieurs workers
  - Un ensemble ZK pour supporter la résilience
  - Observer le fonctionnement d'élection et de perte d'un ou plusieurs leaders

1. Mettre en place l'architecture
   - Installer et configurer Spark
   - Installer et configurer ZK
   - Faire communiquer Spark et ZK


Pour rappel,

- Si l'un des serveurs ZooKeeper est en panne, ZK en utilisera un autre de la liste.
- Tant que la majorité des serveurs ZooKeeper sont opérationnels, le service sera disponible.
- Parce que Zookeeper nécessite une majorité absolue, il est préférable d'utiliser un nombre impair de machines. Généralement 3 ou 5. Par exemple, avec quatre machines, ZooKeeper ne peut gérer que la panne d'une seule machine ; si deux machines échouent, les deux machines restantes ne constituent pas la majorité. Cependant, avec cinq machines, ZooKeeper peut gérer la défaillance de deux machines


Point configuration supposons que nous avons un ensemble ZK composé de 5 znodes.

- **Au niveau du fichier : /opt/zookeeper/conf/zoo.cfg, ajouter les IP des machines**

server.1=192.168.1.10:2888:3888
server.2=192.168.1.20:2888:3888
server.3=192.168.1.30:2888:3888
server.4=192.168.1.40:2888:3888
server.5=192.168.1.50:2888:3888

- **Au niveau du fichier : /opt/spark/conf/spark-env.sh, ajouter les instructions suivantes mais adaptées à vos IP et vos Path.**
-
    - Export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER
    - -Dspark.deploy.zookeeper.url=192.168.1.10:2181,192.168.1.20:2181, 192.168.1.30:2181, 192.168.1.40:2181,192.168.1.50:2181
    - -Dspark.deploy.zookeeper.dir=/opt/spark/spark_recovery"

- **Créer un fichier *myid* dans le répertoire data : Ce fichier contient uniquement une ligne : le numéro de l'instance Zookeeper**

- Nous pouvons également configurer les paramètres suivants :

    - **tickTime=2000**
        - La propriété **initLimit** définit le nombre de graduations d'attente pendant les tentatives de connexion au nœud principal ZooKeeper. Par exemple, si la propriété **initLimit** est définie sur 5 et que la propriété tickTime est de 2000 millisecondes, un noeud ZooKeeper attend 5*2000 millisecondes avant d'abandonner la connexion et de décider de choisir un nouveau leader.

- **initLimit=5**
  - Il est recommandé de conserver la valeur par défaut de la propriété **tickTime** et de régler la propriété **initLimit** en fonction de votre environnement et de votre déploiement afin de réduire l'impact des échecs de nœud.

- **syncLimit=2**
  - limite à laquelle un serveur peut être déclaré obsolète par rapport à un leader

b. Test de résilience à réaliser

Le but ici est d'observer le basculement d'un master à l'autre et le dysfonctionnement pouvant être causé au niveau de l'applicatif.

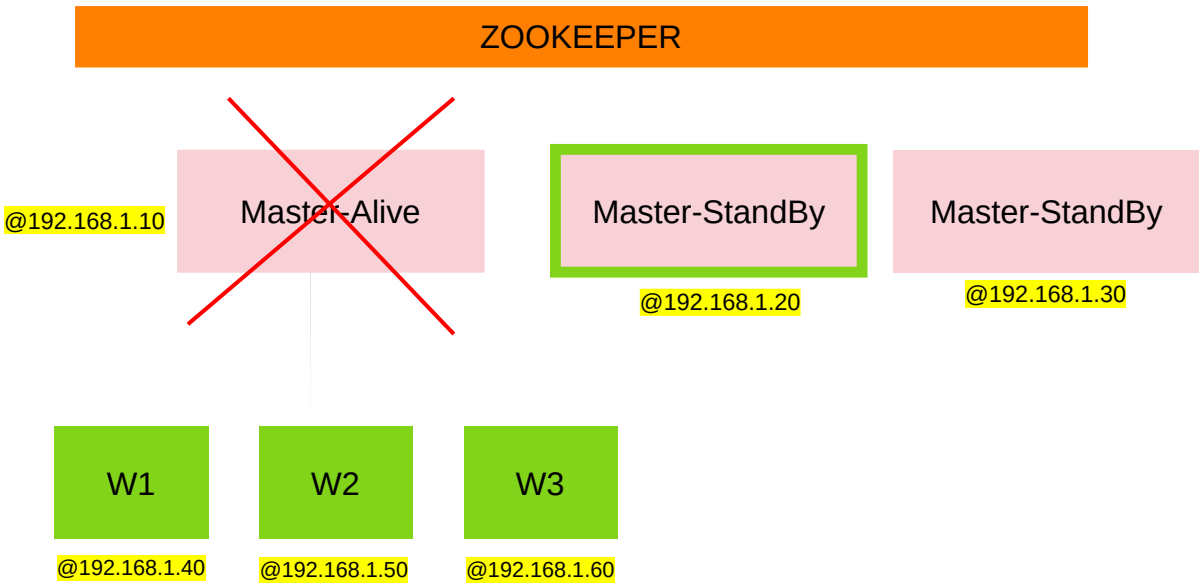


Figure 2 Fonctionnement de la haute disponibilité de SPARK en utilisant ZK

- Test 1
  - Démarrer Spark
  - Arrêter le Master
  - Que constatez-vous ? plusieurs informations peuvent-êtres observées.

- Test 2
  - Lancer une requête consommatrice (select * from table_reservations ou un job de votre choix)
  - Arrêter le master. Que constatez-vous ?
    - La requête a-t-elle été aboutie ?
      - Si oui avec quel master
- Test 3
  - Lancer une requête consommatrice (select * from table_reservations)
  - Arrêter le master. Que constatez-vous ?
    - La requête a-t-elle été aboutie ?
      - Si oui avec quel master
  - Lancer une autre Requête pendant l'arrêt du master
    - Que constatez-vous ?

-

- Test 4
  - Écrire un job que vous exécuterez (on peut également utiliser des exemples spark présent dans le répertoire examples de l'install spark) :
    - en mode cluster
      - Éteindre la machine client. Que constatez-vous ?
    - En mode client
      - Éteindre la machine client. Que constatez-vous ?

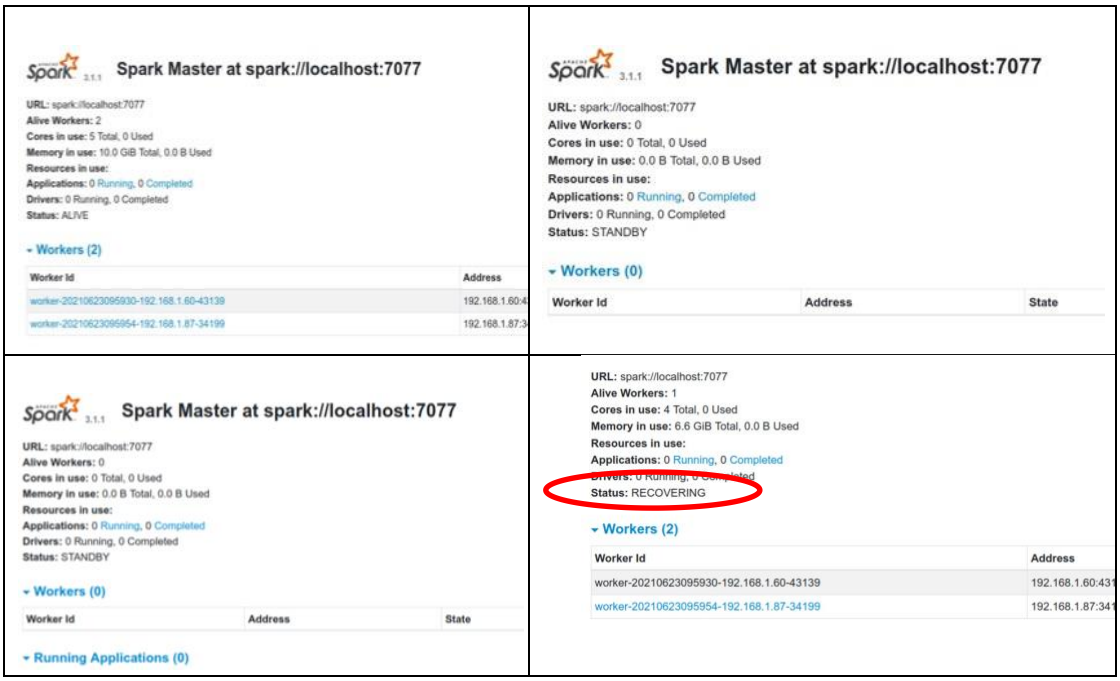


Figure 3 Les différents état des masters Spark gérés par ZK

2. Gestion de la cohérence des données

Nous souhaitons désormais enrichir l'architecture par l'ajout du SGBD Cassandra comme moteur de stockage. Plusieurs scénarios sont possibles.

- S1 : Actif / Actif des traitements et données
- S2 : Actif / Actif des traitements et Actif / Passif des données
- S3 : Actif / Passif des traitements et données

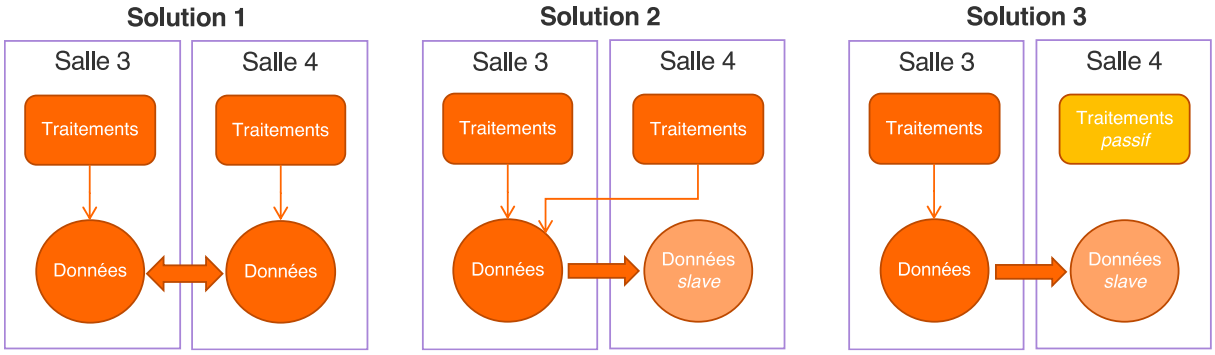


Figure 4 Quelques scénarios d'utilisation d'un cluster multi-sites

2. Quel scénario avez-vous choisi pour l'ajout de Cassandra dans l'architecture initiale ? comment êtes-vous arrivés à ce choix ?

3. Revoir votre chiffrage pour considérer l'ajout de Cassandra selon le scénario décidé.

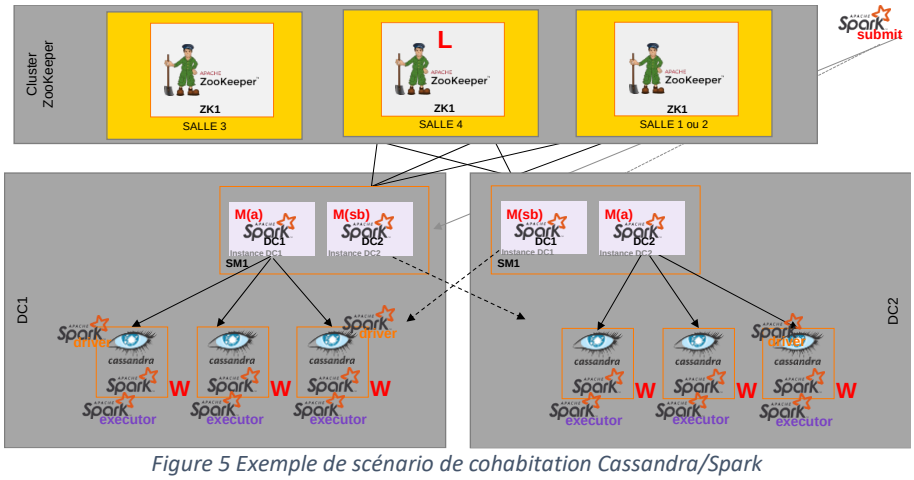


Figure 5 Exemple de scénario de cohabitation Cassandra/Spark

4. Installer et configurer Cassandra

5. Configurer Cassandra et Spark pour permettre leur communication.

a. Gestion de la cohérence des données (optionnelle)

Afin de découvrir la cohérence des données avec le moteur Cassandra, nous allons effectuer un ensemble de tests.

Pour effectuer ces tests, vous pouvez utiliser le programme java sous Moodle.

- Dans le cas de l'utilisation de deux DC
  - Fixer l'écriture en Each-quorum
  - Fixer la lecture en Local-quorum
  -
-
- Dans le cas d'un seul DC
  - Fixer l'écriture en local-quorum
  - Fixer la lecture en one

- Selon les cas, analyser le comportement de votre serveur selon les incidents suivants :
  - Éteindre un nœud
  - Éteindre deux nœuds du même DC
  - Éteindre un nœud de chaque DC
  - Éteindre deux nœuds du même DC
  - Éteindre deux nœuds de chaque DC

Le driver de Cassandra met à disposition une politique de retry-policy qui permet de rejouer les requêtes en cas d'échec.

- Pour ceux qui utilisent le driver 3 de Cassandra (https://docs.datastax.com/en/developer/java-driver/3.6/manual/retries/)

- Pour ceux qui utilisent le driver 4 (à préconiser) : https://docs.datastax.com/en/developer/java-driver/4.4/manual/core/retries/

= > Mettre en place une stratégie de décision, qui permet en cas d'indisponibilité de la consistance (consistency) initiale :

6. Écriture : En cas d'échec, passer du each-quorum à local-quorum

```
static Cluster connect(){
        String[] tableau = { "172.17.0.2", "172.17.0.3", "172.17.0.4", "172.17.0.5","172.17.0.6","172.17.0.7"};
        QueryOptions qu= new QueryOptions().setConsistencyLevel(ConsistencyLevel.LOCAL_QUORUM);
```

```
        SocketOptions socketOptions = new SocketOptions().setReadTimeoutMillis(100000);
    return Cluster.builder()                                    // (1)
        .addContactPoints(tableau)
        .withQueryOptions(new QueryOptions().
                setConsistencyLevel(ConsistencyLevel.LOCAL_QUORUM))
        .withLoadBalancingPolicy(new
com.datastax.driver.core.policies.DCAwareRoundRobinPolicy.Builder()
                .withLocalDc("DC1")
                .withUsedHostsPerRemoteDc(0).allowRemoteDCsForLocalConsistencyLevel().build())
        .withReconnectionPolicy(new ConstantReconnectionPolicy(1000))
        .withRetryPolicy(DefaultRetryPolicy.INSTANCE)
        .withQueryOptions(new QueryOptions().setFetchSize(2000))
        .withSocketOptions(socketOptions)
        .build();
}
```

a. Test de résilience

TEST 1 : DC1 (1UP) & DC2 (2 UP)

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
UN  172.17.0.2  4.25 MiB   256          ?       3178eebf-de5d-4902-a856-0d67b811b2fd  rack1
Datacenter: DC2
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
UN  172.17.0.3  2.76 MiB   256          ?       6e536c1a-1c1d-4859-81a8-5cf57ae85809  rack1
DN  172.17.0.5  2.55 MiB   256          ?       146608cc-fe26-4aff-b445-7758404bb13f  rack1
UN  172.17.0.4  2.96 MiB   256          ?       16afc9d2-5dc5-49df-9c3d-280f76e8c260  rack1
```

TEST 2 : DC1(down) DC2(2up)

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
DN  172.17.0.2  4.25 MiB   256          ?       3178eebf-de5d-4902-a856-0d67b811b2fd  rack1
Datacenter: DC2
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
UN  172.17.0.3  2.7 MiB    256          ?       6e536c1a-1c1d-4859-81a8-5cf57ae85809  rack1
DN  172.17.0.5  2.55 MiB   256          ?       146608cc-fe26-4aff-b445-7758404bb13f  rack1
UN  172.17.0.4  2.71 MiB   256          ?       16afc9d2-5dc5-49df-9c3d-280f76e8c260  rack1
```

TEST 3 : DC1(1 up) DC2(1up)

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
UN  172.17.0.2  4.26 MiB   256          ?       3178eebf-de5d-4902-a856-0d67b811b2fd  rack1
Datacenter: DC2
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
UN  172.17.0.3  2.7 MiB    256          ?       6e536c1a-1c1d-4859-81a8-5cf57ae85809  rack1
DN  172.17.0.5  2.55 MiB   256          ?       146608cc-fe26-4aff-b445-7758404bb13f  rack1
DN  172.17.0.4  2.71 MiB   256          ?       16afc9d2-5dc5-49df-9c3d-280f76e8c260  rack1
```

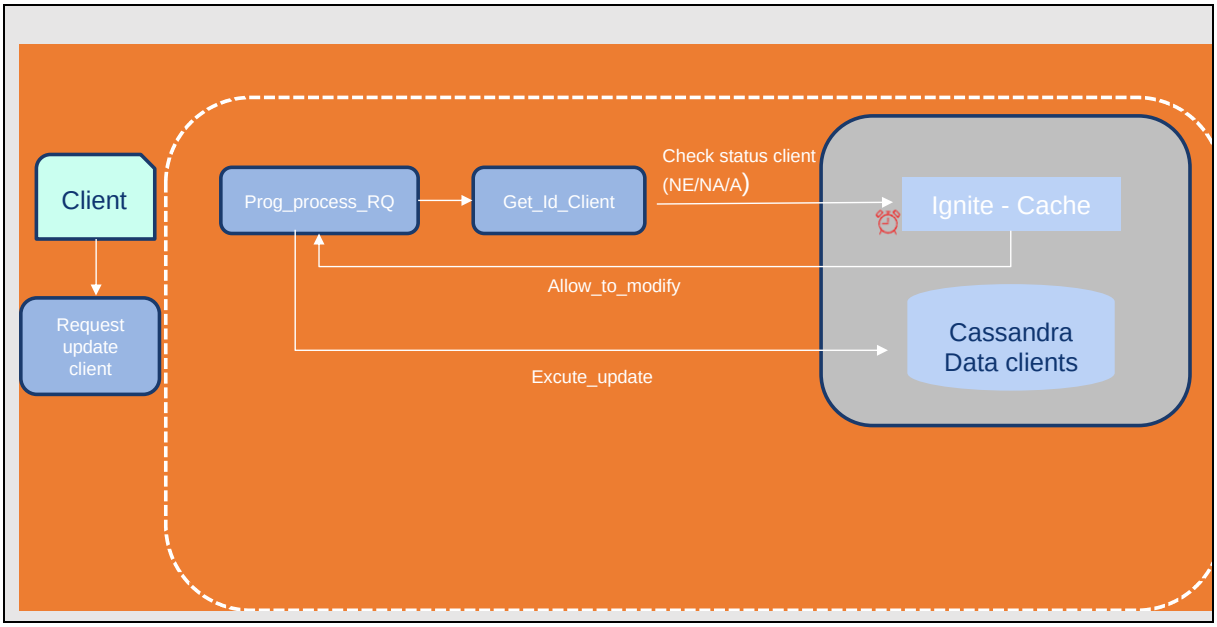
3. Gestion de l'accès concurrentiel (optionnelle)

Cassandra ne gère pas l'accès concurrentiel, nous enrichissons notre architecture par l'ajout de la solution Ignite.

- Écrire un programme Ignite pour gérer les locks.
    - Pour cette fonctionnalité, nous nous appuyons sur le projet du festival d'Avignon ou sur la gestion des logements.
        - Ajouter une colonne prix
        - Ajouter une colonne Etat_disponibilité (loué, dispo, rénovation)

Le programme doit donc reposer sur Ignite pour gérer toute modification relative aux logement (particulièrement prix et disponibilité).

- Augmenter tous les prix (ou le prix d'un logement donné)
- Simultanément, un autre client essaie de lire les données des logements (ou d'un logement donné) pour une éventuelle location.


Client
Request update client
Prog_process_RQ
Get_Id_Client
Check status client (NE/NA/A)
Ignite - Cache
Allow_to_modify
Excute_update
Cassandra Data clients