NFE204 – Bases de données documentaires et distribuées
Projet : étude de Redis

Daniel PONT
juillet 2019

Remerciements

Je tiens ici à exprimer ma vive gratitude à l'équipe pédagogique : MM. Philippe Rigaux, Nicolas Travers et Raphaël Fournier S'niehotta pour la qualité de l'enseignement et de l'organisation de l'UE. Au-delà du cours de lui-même, j'ai beaucoup apprécié la disponibilité de M. Rigaux sur le site moodle et pendant les séances de regroupement.

Un grand merci également à toutes les personnes du CNAM qui ont permis de rendre NFE204 accessible en enseignement à distance.

Je souhaite également témoigner ma reconnaissance à pôle emploi (particulièrement à Mmes Chantal CHALVIDAN et Marie-Noëlle BADOL) pour le financement de ma formation.

"There will be no silver bullet. Rather it will be a combination of technologies"

– Andrew Goulding

Sommaire

1. Introduction

Ce document est le rapport du projet NFE204 « Entreposage et fouille de données » organisé par le CNAM Paris et porte sur l'étude du système NoSQL Redis. Nous expliquerons les principales fonctionnalités de Redis au travers d'une application de suivi temps réel d'un marché boursier. Après une présentation générale, nous décrirons différentes procédures d'installation puis :

- les principales commandes de lecture /écriture ;
- l'insertion de données en masse ;
- la gestion de la persistance ;
- la gestion de la réplication ;
- la gestion du partitionnement ;
- le mécanisme de publication /abonnement de messages.

2. Présentation de Redis

2.1 Introduction

Redis (remote dictionary server) est un SGBD (système de gestion de base de données) :

- open source (site officiel : https://redis.io/);
- de type clé-valeur ;
- avec stockage en mémoire (le mécanisme assurant la durabilité des données est optionnel) ;
- distribuable ;

Redis a été créé en 2009 par Salvatore Sanfilippo qui travaillait à cette époque sur un analyseur temps-réel de logs web. Rencontrant des problèmes importants de scalabilité avec les bases de données traditionnelles, Sanfilippo décida de mettre au point son propre système, conçu pour répondre aux défis posés par les trois V caractéristiques des traitements « Big Data » :

- le volume (logs web) ;
- la vitesse (analyse de flux de données en temps réel) ;
- la variété (données non structurées).

Après quelques semaines de validation réussie en interne, Sanfilippo décida de publier Redis sous licence open source. Le projet connut alors rapidement un succès important, plus particulièrement au sein de la communauté Ruby, GitHub (en 2009) et Instagram (en 2011) étant parmi les premières entreprises notables à l'avoir adopté.

A l'heure où ce document est rédigé (juillet 2019), Sanfilippo dirige toujours le développement open source de Redis. Il est un des acteurs majeurs de la société Redis Lab qui est le principal sponsor du projet. D'après le site db-engines.com Redis est actuellement le huitième SGBD le plus utilisé au monde et le premier pour les bases de données clé - valeur.

Il s'agit d'une solution qui présente les avantages typiques d'un systèmes NoSQL (possibilité de lire/écrire très rapidement une forte quantité de données non structurées, passage à l'échelle,…) mais aussi des limitations importantes, tout aussi typiques de ce genre de système (propriétés ACID[1] non satisfaites, pas de notion de formes normales[2], langage de requête non standard, etc.). En résumé, Redis est avant tout un dictionnaire[3] distribuable sophistiqué, extrêmement performant, qui permet de compléter les fonctionnalités offertes par les bases relationnelles mais en aucun cas de les remplacer totalement.

2.2 Principes fondamentaux

Redis vise principalement à fournir les meilleures performances possibles en terme de rapidité de traitement et de passage à l'échelle.

Afin d'atteindre ces objectifs, **les données sont intégralement stockées en mémoire**. Cela réduit drastiquement les temps de lecture/écriture en évitant les accès disques, particulièrement coûteux sur ce plan. Redis fournit également des mécanismes de persistance (soit en réalisant des snapshots à intervalle de temps régulier, soit par journalisation). Cependant, ces mécanismes ont plus pour rôle d'assurer la reprise sur panne (en restaurant dans la RAM les enregistrement sauvegardés sur disque) que de conserver les données sur le long terme.

Une deuxième caractéristique importante de Redis est sa « légèreté » : il s'agit d'une **application mono-thread avec une faible empreinte mémoire**, simple à installer et configurer, comme nous le verrons au chapitre 3. Dans une optique de passage à l'échelle, on peut donc multiplier le nombre d'instances dans une grappe de serveurs sans grande difficulté. En général on installe une instance par machine (réelle ou virtuelle).

La réplication des données est alors prise en charge par une architecture de type maître-esclave (cf. chapitre 8). Quant au partitionnement, la solution privilégiée est « Redis Cluster » (cf. chapitre 9).

Un mot sur l'implémentation de Redis : la partie serveur, écrite en C, a été conçue pour fonctionner sur des systèmes d'exploitation *Nix (Unix / Linux / MacOS X), elle utilise donc des fonctionnalités spécifiques à ces systèmes (fork, horodatage POSIX ,etc…). Il existe un portage sous Windows, mais ce portage est partiel, expérimental et n'est pas supporté officiellement par Redis Lab.

Si la partie serveur Redis doit donc être exécuté dans un environnement *Nix, les clients en revanche peuvent s'exécuter sur quasiment n'importe quelle plateforme et il existe des interfaces pour les principaux langages de programmation (C, java, python, ruby, perl, PHP, R, Objective-C, …)

1) Atomicité, cohérence, isolation et durabilité : ce sont les propriétés garantissant qu'une transaction est exécutée de façon fiable.

2) Avec Redis, les enregistrements sont stockées dans une seule structure de données et non dans un ensemble de tables liées par des relations. Sachant qu'il n'y a pas non plus d'index secondaire et que les fonctions d'agrégation sont limités, le format des données insérées est donc fortement contraint par le type de requêtes dont elles seront l'objet.

3) Même si les valeurs peuvent être des structures plus complexes que de simples chaînes de caractères (cf sous-chapitre 2.3.2) et que Redis peut être utilisé comme un système de messagerie

2.3 Types de données supportées

Le type de valeur Redis le plus élémentaire est le type *String* qui est « binary safe ». Cela signifie qu'une *String* Redis peut contenir n'importe quel type de données (image JPEG, objet Ruby sérialisé, etc.…). En terme de taille mémoire, une *String* est limitée à 512Mo.

2.3.1 Clé

Les clés Redis sont des *Strings*. Elles peuvent donc en théorie avoir une taille allant jusqu'à 512Mo. En pratique toutefois, l'usage de clés trop longues (ex : 1024 octets) est déconseillé nos seulement en raison en raison de la place qu'elles occupent en mémoire mais aussi parce que les comparaisons, effectuées notamment dans le cadre d'une recherche, sont coûteuses. A l'inverse, le fait d'avoir des clés courtes mais peu explicites n'est pas forcément une bonne idée (exemple : "u1000flw" est vraisemblablement une clé moins pertinente que "user:1000:followers").

2.3.2 Valeur

Pour les valeurs, Redis supporte non seulement les *Strings* mais aussi les collections de *Strings* suivantes:

- *Lists* ;
- *Sets* (ensemble d'éléments sans doublon et non trié) ;
- *Sorted sets* (ensemble sans doublon et trié selon un nombre flottant appelé « score ») ;
- *Hashes* (hashtable où la clé et la valeur sont des *Strings*).

D'autres types de données, plus sophistiquées permettent de répondre à des besoins spécifiques :

- *HyperLogLogs* : structure disponible depuis la version Redis 2.8.9 permettant d'obtenir la cardinalité approximative de très grand ensembles de valeurs. Pour donner un ordre d'idée, un hyperloglog occupant 1.5Kb en mémoire est capable d'estimer la cardinalité d'un ensemble comportant 10^9 éléments avec une erreur type de 2% ;
- *Streams* : introduits dans la version 5.0 de Redis, les streams offrent à la fois les fonctionnalités d'un journal d'évènements (dans lequel on peut insérer des éléments, mais pas les éditer) et d'une file d'attente multi-consommateurs ;
- *Geohashs* : chaînes de caractères permettant de représenter des données géospatiales et implémentés dans Redis à partir de la version 3.2.

Nous examinerons ces types de données plus en détail dans le chapitre 4 (commandes de lecture / écriture).

2.4 Cas d'utilisation

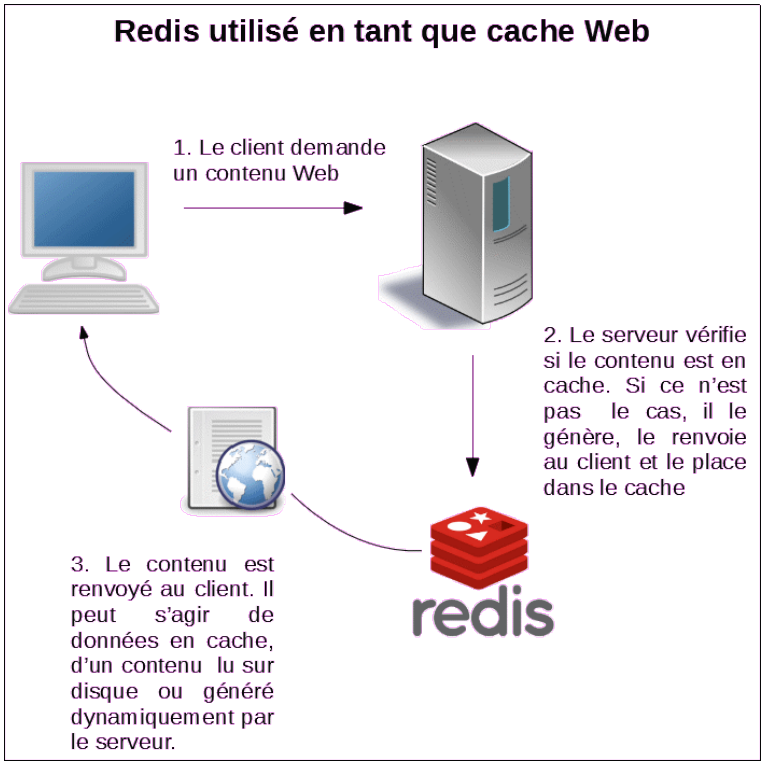
Redis peut être employé en tant que :

- système NoSQL à part entière ;
- complément d'un autre système NoSQL (tels que MongoDB ou ElasticSearch) ;
- « brique » logicielle élémentaire fournissant une fonctionnalité (très) spécifique (ex : cache serveur, compteur permettant de bloquer un client émettant trop de requêtes sur un site web, etc...) .

Nous présenterons dans ce chapitre les principaux cas d'utilisations standards, pour plus de détail, un bon point de départ est l'article « How to take advantage of Redis just adding it to your stack » par l'auteur de Redis, Salvatore Sanfilippo. Une autre source d'informations intéressante est le livre « Mastering Redis » de Jeremy Nelson, publié en 2016 aux éditions Packt Publishing. Les schémas inclus dans les sous-chapitres suivants sont fortement inspirés de cet ouvrage.

2.4.1 Cache

Redis est fréquemment utilisé comme cache serveur pour des applications Web. De nombreux frameworks populaires tel que Ruby-on-Rails et Node.js proposent ainsi des interfaces afin de l'intégrer en tant que module externe. Les données mises en cache peuvent aller de simples chaînes de caractères (ex. : le nom de l'utilisateur connecté à l'application) à des pages web complètes, en passant en passant par des widgets (ex. : un panel affichant les derniers articles consultés par le client d'un site de e-commerce). Le principe de fonctionnement est décrit dans le schéma ci-dessous :



Outre sa rapidité et sa légèreté Redis offre ici une autre atout : la possibilité de définir des dates d'expiration associées aux clés. C'est une fonctionnalité importante puisqu'elle permet d'implémenter des **stratégies de cache LRU (Less Recently Used)** : quand le cache est plein, ce sont les données les plus anciennes qui sont supprimées en priorité.

Soulignons enfin que dans ce premier cas d'utilisation, Redis n'est pas du tout employé en tant que base de données. Les données sont issues d'autres sources : bases relationnelles (Oracle, MySQL, PostgreSQL, …), fichiers, contenu généré dynamiquement par le serveur d'applications, etc.

2.4.2 Suivi d'activité en temps réel

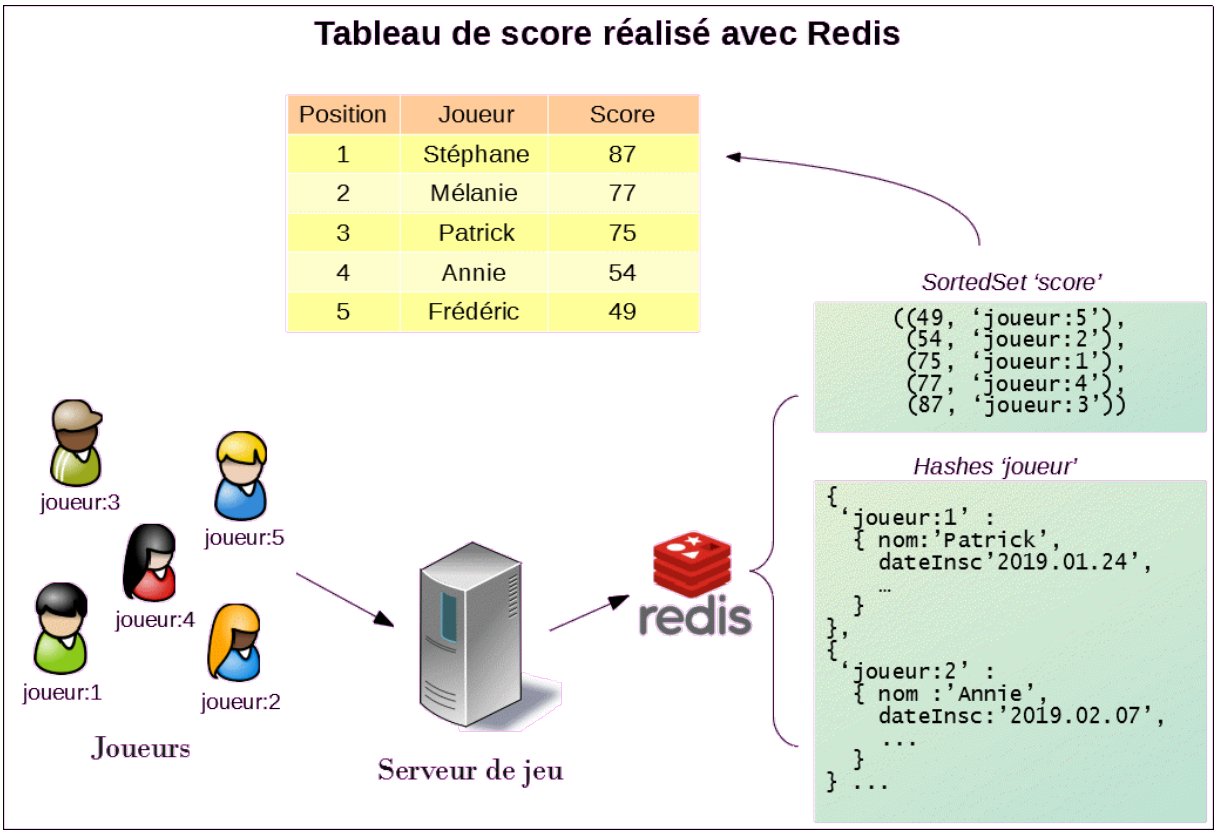
Ce deuxième cas d'utilisation, le suivi d'activité en temps réel, recouvre un vaste champ d'applications. Il peut s'agir d'une fonctionnalité relativement basique comme compter le nombre de messages postés par un utilisateur sur un forum dans un intervalle de temps donné, afin de s'assurer que ce ne sont pas des spams émis par un robot. Le suivi temps réel sert aussi à des traitements analytiques nettement plus poussés : par exemple mesurer précisément l'activité d'un internaute (sur quels liens a t'il cliqué, quelles ont été les pages visitées, dans quel ordre, combien de temps a été consacré à la lecture de chaque page, etc.).

2.4.3 Tableau de score / classement

Redis est fréquemment employé dans l'implémentation de tableaux de score, notamment pour des applications de jeux en ligne. La figure ci-dessous montre un cas simple avec deux types de données Redis :

- Une structure *sorted set* qui contient les scores (ici triés par ordre croissant) de chaque joueur (représenté par son identifiant) ;
- Une structure *hashes* qui contient le profil de chaque joueur (nom, date d'inscription, etc.)

Le tableau de score est constitué à partir des informations issues de ces deux structures :



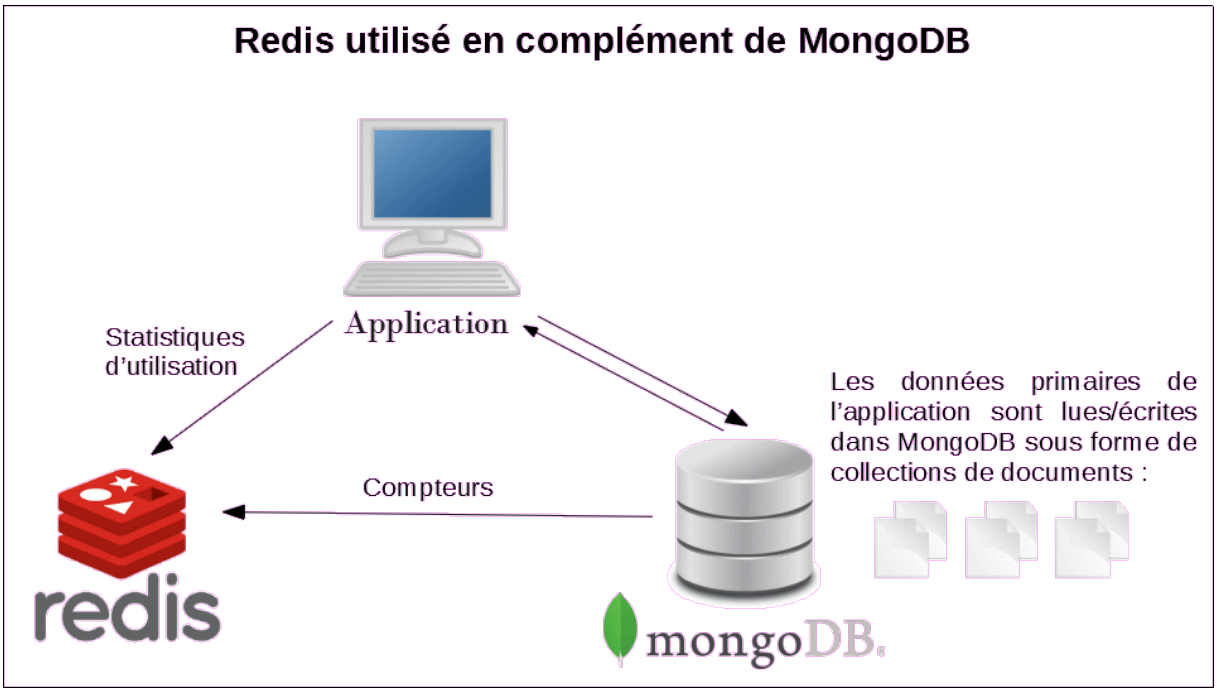
L'intérêt de disposer d'un tableau de score face à développer, léger, scalable, mis à à jour en temps réel est évident dans le contexte des jeux en ligne, notamment les MMO (Massive Multiplayer Online Games -ex : « World of Warcraft ») qui peuvent comporter plusieurs milliers de participants connectés simultanément. Cependant ce type de tableaux n'est pas limité à une utilisation ludique. Sur GitHub par exemple, ils prennent la forme de calendriers dans lesquels sont affichés le nombre de contributions pour chaque compte utilisateur.

2.4.4 Redis en tant que complément de MongoDB

Deux articles publiés en 2014 par DJ Walker-Morgan de la société compose (une filiale d'IBM) montrent en quoi l'utilisation conjointe de Redis et MongoDB s'avère pertinente. Dans le premier article (Redis, MongoDB & the Power of Incremency) Walker-Morgan décrit une application qui comportait un goulet d'étranglement dû à l'incrémentation de plusieurs compteurs (un par client) dans MongoDB. Plutôt que de multiplier le nombre d'instances de MongoDB, il s'est avéré plus simple d'ajouter une base Redis et de lui déléguer la fonctionnalité « compteur incrémental ».

Dans le second article (« Why (and how to) Redis with your MongoDB »), Walker-Morgan explique entre autres comment l'algorithme HyperLogLog (de Philippe Flajolet et al.), implémenté dans Redis, permet d'évaluer avec précision et moindre coût le nombre d'éléments d'un grand ensemble de données afin d'obtenir les statistiques d'utilisation d'une application.

Ces utilisations de Redis en complément de MongoDB sont résumées dans le schéma ci-dessous :



2.4.5 Redis, Logstash et ElasticSearch

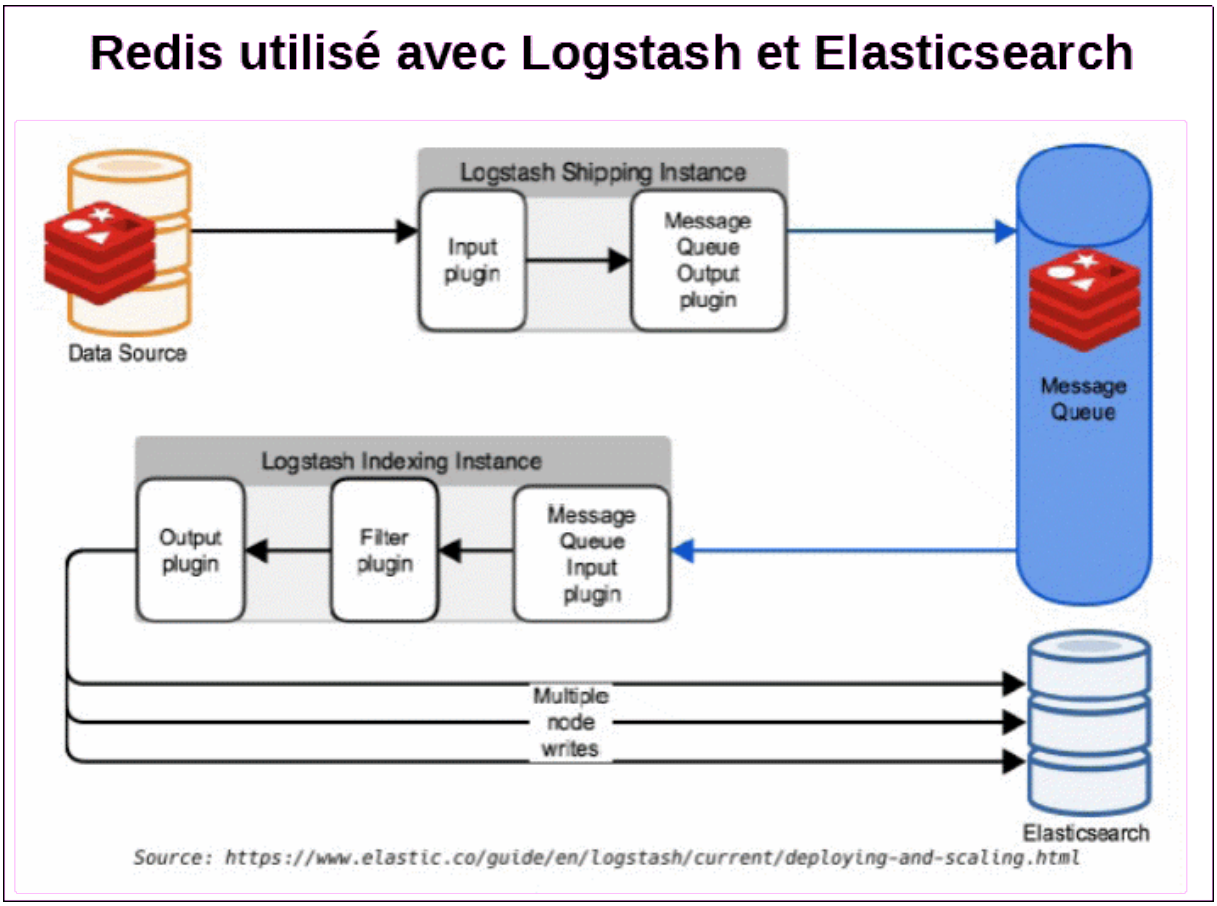
Logstash est un outil open source qui :

- collecte les traces (journaux système, journaux Web, journaux d'erreurs...) issues de différentes sources ;
- les stocke dans ElasticSearch ;
- les présente (sous forme de graphiques et/ou de rapports) avec le projet Kibana.

Le trio : ElasticSearch, Logstash et Kibana, souvent appelé « ELK Stack », est une solution logicielle populaire pour analyser les logs. Redis est régulièrement employé :

- en amont de ce trio, comme plugin d'entrée de Logstash afin de fédérer les traces issues de différentes sources ;
- comme file d'attente de message entre une instance Logstash chargée de collecter les traces et une autre instance Logstash chargée elle de stocker les données dans ElasticSearch.

Ces deux modes de fonctionnement sont représentés dans le schéma ci-dessous :



2.4.6 Mécanisme Pub/Sub

Nous présenterons une exemple d'utilisation de ce mécanisme dans le chapitre 10.

3. Installation de Redis

Nous présentons ici les procédures d'installation et la configuration de Redis en ligne de commandes sur une distribution linux ubuntu (version 18.04).

3.1 Installation et configuration d'une instance Redis simple

Pour installer une instance Redis simple sur ubuntu, l'option la plus simple est d'utiliser le gestionnaire de paquets APT (Advance Packaging Tool) :

```
# Mise à jour du cache APT
sudo apt update
# Installation du serveur Redis
sudo apt install redis-server
```

Une autre option est de télécharger et de compiler le code source Redis de manière à être sûr d'avoir la version stable la plus à jour. C'est la solution que nous décrivons dans les sous-paragraphes suivants.

3.1.1 Installation de Redis à partir du code source

Les lignes de commandes afin de télécharger et compiler la version stable de Redis la plus récente sont listées ci-dessous :

```
# Mise à jour des paquets APT et installation des dépendances
sudo apt-get update
sudo apt-get install build-essential tcl curl

# Téléchargement et extraction du code source
# de la version stable la plus récente de Redis
cd /tmp
curl -O http://download.redis.io/redis-stable.tar.gz
tar xzvf redis-stable.tar.gz

# Compilation et installation
cd redis-stable
make
make test
sudo make install
```

A ce stade on peut déjà démarrer une instance Redis simple, avec les paramètres par défaut, en saisissant l'instruction suivante :

```
/usr/local/bin/redis-server
```

3.1.2 Configuration

Dans cette partie, nous montrons comment :

1. Mettre en place un fichier de configuration pour paramétrer le serveur Redis ;
2. Exécuter le serveur Redis sous forme de service.

En supposant qu'on a installé Redis à partir des sources comme indiqué dans le chapitre précédent, on dispose d'un fichier de configuration standard qu'on peut recopier :

```
# Création du fichier de conf. (copié à partir des sources)
sudo mkdir /etc/redis
sudo cp /tmp/redis-stable/redis.conf /etc/redis
```

On peut ensuite éditer ce fichier, par exemple avec nano :

```
sudo nano /etc/redis/redis.conf
```

Ubuntu utilise *systemd* comme système d'initialisation, il faut donc chercher dans le fichier redis.conf, le paramètre *supervised* et remplacer la valeur *no* par *systemd* :

etc/redis/redis.conf

```
. . .

# If you run Redis from upstart or systemd, Redis can interact with your
# supervision tree. Options:
#   supervised no      - no supervision interaction
#   supervised upstart - signal upstart by putting Redis into SIGSTOP mode
#   supervised systemd - signal systemd by writing READY=1 to $NOTIFY_SOCKET
#   supervised auto    - detect upstart or systemd method based on
```

```
#                        UPSTART_JOB or NOTIFY_SOCKET environment variables
# Note: these supervision methods only signal "process is ready."
#       They do not enable continuous liveness pings back to your supervisor.
supervised systemd
. . .
```

Il faut également spécifier le paramètre *dir* correspondant au répertoire dans lequel Redis sauvegarde les données persistantes :

etc/redis/redis.conf

```
. . .

# The working directory.
#
# The DB will be written inside this directory, with the filename specified
# above using the 'dbfilename' configuration directive.
#
# The Append Only File will also be created inside this directory.
#
# Note that you must specify a directory here, not a file name.
dir /var/lib/redis

. . .
```

L'étape suivante consiste à réaliser le paramétrage servant à exécuter Redis en tant que service. On commence par créer le fichier *redis.service* :

```
sudo nano /etc/systemd/system/redis.service
```

Puis on saisit dans ce fichier les paramètres suivants :

/etc/systemd/system/redis.service

```
[Unit]
Description=Redis In-Memory Data Store
After=network.target

[Service]
User=redis
Group=redis
ExecStart=/usr/local/bin/redis-server /etc/redis/redis.conf
ExecStop=/usr/local/bin/redis-cli shutdown
Restart=always

[Install]
WantedBy=multi-user.target
```

La section *Unit* précise que le réseau doit être disponible avant de démarrer le service. La section *Service* spécifie quant à elle les commandes de démarrage et d'arrêt du service. Pour des raisons de sécurité, il est souhaitable d'exécuter Redis avec un utilisateur dédié (et non le compte root). La création et la configuration de cet utilisateur (*redis*) s'effectue avec les commandes ci-dessous :

```
# Création de l'utilisateur et du groupe "redis"
sudo adduser --system --group --no-create-home redis

# Création du réperoire hébergeant les données du serveur redis
sudo mkdir /var/lib/redis
```

```
# Ce répertoire appartient à l'utilisateur redis
sudo chown redis:redis /var/lib/redis

# Les autres utilisateurs n'y ont pas  accès
sudo chmod 770 /var/lib/redis
```

Il ne reste plus alors qu'à démarrer le service Redis et à tester son status :

```
sudo systemctl start redis
sudo systemctl status redis
```

Si tout fonctionne normalement, un message indiquant que le service est actif apparaît :

```
● redis.service - Redis In-Memory Data Store
   Loaded: loaded (/etc/systemd/system/redis.service; disabled; vendor preset: enabled)
   Active: active (running) since Fri 2019-07-16 12:10:40 CEST; 20s ago
 Main PID: 13790 (redis-server)
    Tasks: 4 (limit: 4915)
   CGroup: /system.slice/redis.service
           └─13790 /usr/local/bin/redis-server 127.0.0.1:6379
```

Finalement, pour valider la configuration, on peut se connecter à l'instance Redis (via le client en ligne de commande installé par défaut avec le serveur) et taper quelques instructions:

```
redis-cli
127.0.0.1:6379> ping
```

```
PONG
```

```
127.0.0.1:6379> set test "Ca marche"
```

```
OK
```

3.2 Installation d'un système maître-esclave (réplication)

Dans ce chapitre, nous emploierons le logiciel docker afin d'installer un système Redis multi-nœud sur une seule machine physique. Avec cette solution, chaque nœud est exécuté dans un conteneur distinct. Nous paramétrerons l'architecture de l'ensemble via un fichier de configuration docker-compose. .

3.2.1 Installation de docker et docker-compose

Les commandes d'installation de docker et docker-compose sur une distribution ubuntu sont les suivantes :

```
# Mise à jour des paquets APT
sudo apt update

# Installation des dépendances
sudo apt install apt-transport-https ca-certificates software-properties-common

# Récupération et installation de la clé GPG (Gnu Privacy Guard)
# du dépôt officiel docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

```
# Ajout du dépôt officiel docker à APT
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic
stable"

# Installation de docker
sudo apt install docker-ce

# Paramètrage pour exécuter docker avec l'utilisateur courant (sans sudo)
sudo usermod -aG docker ${USER}
su - ${USER}

# Téléchargement de docker compose
sudo curl -L "https://github.com/docker/compose/releases/download/1.23.1/docker-compose-$
(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# Paramètrage des droits pour rendre docker-compose exécutable
sudo chmod +x /usr/local/bin/docker-compose
```

3.2.2 Création d'un nœud maître, d'un nœud esclave et d'un nœud sentinel.

Il existe (notamment sur GitHub) des configurations docker-compose prêtes à l'emploi afin d'installer une architecture Redis distribuée intégrant un mécanisme de réplication. Nous avons retenu celle de Li Yi et Ty Alexander (https://github.com/AliyunContainerService/redis-cluster) en raison de sa souplesse d'utilisation.

Par défaut, cette solution offre une architecture avec trois nœuds Redis :

- un nœud maître
- un nœud esclave
- un nœud sentinel (dont nous étudierons plus loin le rôle)

La topologie du système est décrite avec un fichier YAML(Yet Another Markup Language) :

```
master:
  image: redis:5
slave:
  image: redis:5
  command: redis-server --slaveof redis-master 6379
  links:
    - master:redis-master
sentinel:
  build: sentinel
  environment:
    - SENTINEL_DOWN_AFTER=5000
    - SENTINEL_FAILOVER=5000
  links:
    - master:redis-master
    - slave
```

La mise en place est réalisée avec quelques lignes de commande :

```
# Téléchargement des fichiers
git clone  https://github.com/AliyunContainerService/redis-cluster

# Construction de l'image docker correspondant au nœud sentinel
cd redis-cluster
docker-compose build
```

```
# Démarrage du système
docker-compose up -d
```

L'instruction suivante permet de vérifier l'état du système :

```
docker-compose ps
```

```
Name                              Command                  State       Ports
-------------------------------------------------------------------------------------------
redis-cluster_master_1       docker-entrypoint.sh redis ...   Up      6379/tcp
redis-cluster_sentinel_1     sentinel-entrypoint.sh           Up      26379/tcp, 6379/tcp
redis-cluster_slave_1        docker-entrypoint.sh redis ...   Up      6379/tcp
```

On peut très facilement augmenter le nombre de sentinels et d'esclaves:

```
docker-compose scale sentinel=3
docker-compose scale slave=2
docker-compose ps
```

```
Name                              Command                State          Ports
-------------------------------------------------------------------------------------------
redis-cluster_master_1     docker-entrypoint.sh redis ...   Up      6379/tcp
redis-cluster_sentinel_1   sentinel-entrypoint.sh           Up      26379/tcp, 6379/tcp
redis-cluster_sentinel_2   sentinel-entrypoint.sh           Up      26379/tcp, 6379/tcp
redis-cluster_sentinel_3   sentinel-entrypoint.sh           Up      26379/tcp, 6379/tcp
redis-cluster_slave_1      docker-entrypoint.sh redis ...   Up      6379/tcp
redis-cluster_slave_2      docker-entrypoint.sh redis ...   Up      6379/tcp
```

Pour tester la reprise sur panne il suffit de mettre en pause le nœud *redis-cluster_master_1* puis d'interroger un des nœuds sentinel afin de vérifier que le nouveau nœud maître est *redis-cluster_slave1* ou *redis-cluster_slave_2*

```
docker pause redis-cluster_master_1
docker exec redis-cluster_sentinel_1  redis-cli -p 26379 SENTINEL get-master-addr-by-name mymaster
```

Nous reviendrons sur les mécanismes de réplication redis dans le chapitre 8.

3.3 Installation d'un cluster Redis (partitionnement)

Afin d'installer, sur une seule machine physique, un cluster Redis minimal permettant d'étudier le mécanisme de partitionnement , nous suivrons les instructions fournies dans la documentation officielle (https://redis.io/topics/cluster-tutorial):

```
# création du répertoire racine du cluster
mkdir cluster-test
cd cluster-test
# récupération de l'exécutable redis-server
# produit par la compilation des sources redis (cf. §4.1.1)
cp /usr/local/bin/redis-server .
# création d'un sous-répertoire par nœud composant le cluster
# le nom du sous-répertoire correspond au port TCP du noeud
mkdir 7000 7001 7002 7003 7004 7005
```

On crée ensuite un fichier redis.conf par nœud :

```
# configuration pour le noeud 7000
# les paramètres sont les mêmes pour les autres nœuds à l'exception du numéro de port
cd 7000
nano redis.conf
```

cluster-test/7000/redis.conf

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

On peut ensuite démarrer les nœuds, par exemple en ouvrant 6 onglets dans une fenêtre de terminal et en saisissant les instructions suivantes :

```
# instructions à saisir dans le 1er onglet :
# (si le répertoire courant est cluster-test)
cd 7000
../redis-server ./redis.conf
# répéter les instructions ci-dessus pour les noeuds 7001 à 7005
# dans les onglets 2 à 6
. . .
```

Remarque : vu que nous n'avons pas fourni de fichier *nodes.conf*, chaque nœud s'attribue son propre identifiant comme indiqué dans les traces :

```
12642:M 29 Jul 2019 17:19:17.609 * No cluster configuration found, I'm
de2a2eab58808ceed2df79c41bdb6dcc42f6ffc6
```

Il ne reste plus, afin de compléter l'installation du cluster redis, qu'à exécuter l'instruction suivante (disponible depuis la version 5 de redis, avant il fallait utiliser un script ruby) :

```
redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 \
--cluster-replicas 1
```

Redis propose alors une configuration type pour les 6 nœuds avec :

- 3 paires maître-esclave (car le nombre de réplicas, spécifié par *cluster-replicas*, vaut 1) ;
- une partition équitable des hashs correspondant aux futurs fragments (*hashs slots*) sur les 3 paires de nœuds.

On obtient ainsi la sortie reproduite ci-dessous :

```
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:7004 to 127.0.0.1:7000
Adding replica 127.0.0.1:7005 to 127.0.0.1:7001
Adding replica 127.0.0.1:7003 to 127.0.0.1:7002
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: de2a2eab58808ceed2df79c41bdb6dcc42f6ffc6 127.0.0.1:7000
```

```
   slots:[0-5460] (5461 slots) master
M: f4067d357deb32f702bf40484a9eca3340ab5105 127.0.0.1:7001
   slots:[5461-10922] (5462 slots) master
M: 7ac2b7e1ac2f5847bec6145d9a939fb9be559292 127.0.0.1:7002
   slots:[10923-16383] (5461 slots) master
S: 1d4f0a26c03a748b4037aa2a3724029130ff3964 127.0.0.1:7003
   replicates 7ac2b7e1ac2f5847bec6145d9a939fb9be559292
S: 57074491d12ff52c8eff5c0b10b3e48b959d85e5 127.0.0.1:7004
   replicates de2a2eab58808ceed2df79c41bdb6dcc42f6ffc6
S: c5bbfa310ff60ed178989bd0cd04bbf368486193 127.0.0.1:7005
   replicates f4067d357deb32f702bf40484a9eca3340ab5105
Can I set the above configuration? (type 'yes' to accept): yes
```

Si on choisit de valider cette configuration en tapant « yes », Redis crée un fichier nodes.conf par instance (respectivement dans les sous-répertoires 7000,7001, …, 7005) et le cluster devient opérationnel. Nous reviendrons sur le fonctionnement du cluster que nous venons d'installer dans le chapitre 9.

4. Commandes de lecture/écriture

4.1 Commandes communes aux différents types de données

Afin de présenter les commandes communes aux différents types de données, définissions une paire clé-valeur simple :

```
127.0.0.1:6379> set ORCL "Oracle Corporation"
OK
```

4.1.1 Exists : vérification de l'existence d'une clé

La commande *exists* permet de vérifier l'existence d'une clé dans la base. Elle retourne respectivement 1 ou 0 suivant que la clé existe ou non :

```
127.0.0.1:6379> exists ORCL
(integer) 1
```

4.1.2 Del : suppression d'une clé

La commande *del* permet de supprimer une clé et la valeur associée. *Del* retourne 1 si la suppression a bien été effectuée, 0 sinon :

```
127.0.0.1:6379> del ORCL
(integer) 1
```

4.1.3 Expire : clé avec durée de vie limitée

Redis offre la possibilité de définir un délai d'expiration sur chaque clé. Une fois ce délai écoulé, la clé et la valeur associée sont automatiquement supprimées :

```
127.0.0.1:6379> set sessionUtilisateur191087 "login: jbelfort"
OK
# mise en  place d'un délai d'expiration de 1 heure soit 3600 secondes :
127.0.0.1:6379> expire sessionUtilisateur191087 3600
(integer) 1
```

On peut vérifier à tout moment le temps restant avant expiration du délai :

```
127.0.0.1:6379> ttl sessionUtilisateur191087
(integer) 2410
```

On peut également définir le délai d'expiration lors de la création de la clé :

```
127.0.0.1:6379> set sessionUtilisateur061008 "login: ggekko" ex 3600
OK
```

Ajoutons qu'il est possible de :

- travailler avec des délais en millisecondes au lieu de secondes. Dans ce cas il faut remplacer la commande *expire* par *pexpire* et la commande *ttl* par *pttl ;*
- définir des délais non seulement avec une durée mais également avec une date d'expiration (spécifiée sous forme de timestamp unix via les fonctions *expireat* / *pexpireat*)

4.2 Redis Strings

4.2.1 Fonctions de base

Le type *string* est le type de données Redis le plus simple. Pour créer/modifier une *string* on utilise la commande s*e*t, pour la lire la commande *get* :

```
# création ou modification de la clé AAPL
127.0.0.1:6379> set AAPL "Apple Inc."
OK
# lecture de la clé AAPL
127.0.0.1:6379> get AAPL
"Apple Inc."
# set nx : création stricte  (ne  fonctionne que si la clé n'existe pas encore)
127.0.0.1:6379> set GOOG "Google Inc." nx
OK
# set xx : modification stricte (ne fonctionne que si la clé existe déjà)
127.0.0.1:6379> set GOOG "Alphabet Inc." xx
OK
```

Il est également possible de lire / écrire simultanément plusieurs *strings*:

```
# écriture simultanée de plusieurs Strings
127.0.0.1:6379> mset AMZN "Amazon.com Inc." FB "Facebook, Inc."
```

```
OK
```

```
# lecture simultanée de plusieurs Strings
127.0.0.1:6379> mget AMZN FB
```

```
1) "Amazon.com Inc."
2) "Facebook, Inc."
```

D'autre part, le type *string* n'est pas limité aux chaînes de caractères, on peut aussi l'utiliser comme compteur avec les commandes *incr(by)* / *decr(by)*:

```
# initialisation du compteur
127.0.0.1:6379> set nb_transactions_nasdaq 0
```

```
OK
```

```
# incrément d'une unité
127.0.0.1:6379> incr nb_transactions_nasdaq
```

```
(integer) 1
```

```
# décrément d'une unité
127.0.0.1:6379> decr nb_transactions_nasdaq
```

```
(integer) 0
```

```
# incrément de n unités
127.0.0.1:6379> incrby nb_transactions_nasdaq 20
```

```
(integer) 20
```

```
# décrément de n unités
127.0.0.1:6379> decrby nb_transactions_nasdaq 5
```

```
(integer) 15
```

4.2.2 Bitmaps

Les *bitmaps* Redis ne sont pas à proprement parler un type de données à part entière mais plutôt des opérations portant sur les bits d'une s*tring*. Ces opérations permettent entre autres de créer et d' effacer des bits individuels, de compter tous les bits à 1, trouver le premier bit défini ou non défini. L'un des principaux avantages des *bitmaps* Redis réside dans leur efficacité en terme de stockage : il est possible de stocker les valeurs d'une variable relatives à 4 milliards d'individus en n'utilisant que 512 Mo de mémoire. Considérons par exemple la clé *status_trader* que nous allons créer afin de décrire la présence/absence de traders dans une salle de marché. Pour simplifier nous nous limiterons à un ensemble de cinq traders, chacun représenté par un bit. Ainsi *status_trader*= "10001" signifie que le premier et le cinquième trader sont présents, les trois autres absents. Les fonctions de lecture/écriture du *bitmap* sont les suivantes :

```
# initialisation / édition d'un bit dans la bitmap
# on commence par écrire dans le bit de position 0 la valeur 1
# la valeur retournée correspond à la valeur stockée dans le bit avant écriture
# (ici 0 car on n'avait encore rien écrit dans le bitmap)
127.0.0.1:6379> setbit status_trader 0 1
```

```
(integer) 0
```

```
# écriture dans le bit de position 1 la valeur 0
```

```
127.0.0.1:6379> setbit status_trader 1 0
(integer) 0
# écriture dans le bit de position 2 la valeur 0
127.0.0.1:6379> setbit status_trader 2 0
(integer) 0
# écriture dans le bit de position 3 la valeur 0
127.0.0.1:6379> setbit status_trader 3 0
(integer) 0
# écriture dans le bit de position 4 la valeur 1
127.0.0.1:6379> setbit status_trader 4 1
(integer) 0
# lecture du bit à la position 4
127.0.0.1:6379> getbit status_trader 4
(integer) 1
# comptage du nombre de bits ayant la valeur 1
127.0.0.1:6379> bitcount status_trader
(integer) 2
# détermination dela position du premier bit ayant une valeur donnée entre 2 positions
# (ex : 1er bit ayant la valeur 1 entre les positions 2 et 4)
127.0.0.1:6379> bitpos status_trader 1 0 4
(integer) 0
```

4.3 Redis Lists

Les *lists* sont des séquences d'éléments ordonnées. Il est important de noter que les *lists* Redis sont des **listes chaînées** (et non des tableaux comme en python). Cela implique que l'ajout d'un nouvel élément en tête ou en queue d'une liste est effectuée en **temps constant** (la vitesse de l'ajout ne dépend pas de la taille de la liste). L'avantage réside dans la capacité d'ajouter des éléments à une très longue liste très rapidement. L'inconvénient (par rapport à un liste implémentée avec un tableau) est que le temps d'accès d'un élément à partir de son index est proportionnel à la valeur de l'index. En conséquence, si on souhaite disposer d'une collection ordonnée permettant d'accéder rapidement à des éléments dont la position est aléatoires, une *list* n'est pas une solution adaptée. Dans ce cas, il vaut mieux utiliser un *sorted set*.

Les principales commandes d'écriture/lecture dans les *lists* sont les suivantes :

```
# création et/ou ajout d'un ou plusieurs élément(s) en fin de liste
127.0.0.1:6379> rpush titres_nasdaq_100 ADBE ADI ADP ADSK
(integer) 4
# création et/ou ajout d'un ou plusieurs élément(s) en début de liste
127.0.0.1:6379> lpush titres_nasdaq_100 AAL AAPL
(integer) 6
# récupération d'éléments situés entre 2 indexes.
# l'index de début et/ou de fin peuvent être négatifs : cela indique le nbre d'elts à omettre
# ex. 0 -4 indique tous les éléments à partir du premier, sauf les 4 derniers
127.0.0.1:6379> lrange titres_nasdaq_100 0 -4
1) "AAPL"
2) "AAL"
```

```
3) "ADBE"
```

```
# récupération et suppression du dernier élément de la list
127.0.0.1:6379>rpop titres_nasdaq_100
```

```
"ADSK"
```

```
# récupération et suppression du premier élément de la list
127.0.0.1:6379>lpop titres_nasdaq_100
```

```
"AAPL"
```

```
# réduction de la liste aux éléments dont les indexes sont situés entre 2 valeurs
127.0.0.1:6379>ltrim titres_nasdaq_100 1 3
```

```
OK
```

Remarque : il existe également des versions bloquantes des commandes de lecture *rpop* et *lpop*. Ce sont respectivement *brpop* et *blpop*. Ces commandes ne rendent la main à l'appelant que lorsqu'un nouvel élément est ajouté à la liste ou qu'un délai d'expiration spécifié par l'utilisateur est atteint.

4.4 Redis Hashes

Un *hash* Redis permet d'associer à un identifiant un ensemble de paires clé-valeur :

```
# création d'un hash
127.0.0.1:6379> hmset AAPL_profil secteur technologie ceo "Tim Cook" nb_employes 100000
```

```
OK
```

```
# ajout d'une paire clé/valeur dans un hash
127.0.0.1:6379> hmset AAPL_profil siege "Cupertino, CA, US"
```

```
OK
```

```
# récupération de la valeur associée à une clé
127.0.0.1:6379> hget AAPL_profil ceo
```

```
"Tim Cook"
```

```
# récupération de toutes les paires clé/valeur
127.0.0.1:6379> hgetall AAPL_profil
```

```
1) "secteur"
2) "technologie"
3) "ceo"
4) "Tim Cook"
5) "nb_employes"
6) "100000"
7) "siege"
8) "Cupertino, CA, US"
```

4.5 Redis Sets

Les *sets* Redis sont des ensembles de *strings* non ordonnés et sans doublons, sur les quels on peut appliquer des opérations ensemblistes :

```
# création / ajout d'élément(s) dans un set (GAFA)
127.0.0.1:6379> sadd GAFA GOOG AAPL FB AMZN
```

```
(integer) 4
```

```
# récupération de tous les éléments du set (l'ensemble n'est pas ordonné)
127.0.0.1:6379> smembers GAFA
```

```
1) "AAPL"
2) "AMZN"
3) "FB"
4) "GOOG"
```

```
# vérification qu'un élément appartient au set
127.0.0.1:6379> sismember GAFA AMZN
```

```
(integer) 1
```

```
# vérification qu'un élément n'appartient pas au set
127.0.0.1:6379> sismember GAFA MSFT
```

```
(integer) 0
```

```
# création d'un nouveau set (BATX)
127.0.0.1:6379> sadd BATX  BABA BIDU TCEH XIACF
```

```
(integer) 4
```

```
# union de deux sets (GAFA et BATX) pour produire un nouveau set (GAFA_BATX)
127.0.0.1:6379> sunionstore GAFA_BATX GAFA BATX
```

```
(integer) 8
```

Il existe d'autres commandes permettant par exemple de retirer un élément d'un *set* au hasard (*spop*) ou de créer un set correspondant à l'intersection de deux autres sets (*sinterstore*).

4.6 Redis Sorted sets

Les *sorted set* Redis sont un type de données hybride pouvant être vu comme la combinaison d'un set (ensemble de *strings* sans doublons) et d'un *hash* (chaque valeur est associée à une clé, le *score,* qui est un nombre réel et sert à trier l'ensemble). Les *sorted sets* sont naturellement utilisé pour stocker des classements :

```
# création / ajout d'élément(s) dans un sorted set  avec  la fonction ZADD:
# ZADD est similaire au SADD des sets simples mais prend un argument supplémentaire :
# le score utilisé pour le classement
127.0.0.1:6379> zadd performances_nasdaq 36.99 AVDR 35.94 CETX  27.93 NCSM 32.78 CLVS 34.00
ZVO 34.67 ICUI
```

```
(integer) 6
```

```
# ZRANGE /  ZREVRANGE : récupération des éléments du sorted set situés entre 2 indexes
# (ordonnés respectivement  par score croissant ZRANGE /décroissant ZREVRANGE)
# " 0 -1" signifie du premier au dernier élément (cf. fonction LRANGE des sets simples)
127.0.0.1:6379> zrange performances_nasdaq 0 -1
```

```
1) "NCSM"
2) "CLVS"
3) "ZVO"
4) "ICUI"
5) "CETX"
6) "AVDR"
```

```
# il est possible d'inclure les scores dans le résultat :
127.0.0.1:6379> zrange performances_nasdaq 0 -1 withscores
```

```
 1) "NCSM"
 2) "27.93"
 3) "CLVS"
 4) "32.780000000000001"
 5) "ZVO"
 6) "34"
 7) "ICUI"
 8) "34.670000000000002"
 9) "CETX"
10) "35.939999999999998"
11) "AVDR"
12) "36.990000000000002"
```

```
# on peut également définir un intervalle à partir de deux scores :
127.0.0.1:6379> zrangebyscore performances_nasdaq 34 35
```

```
1) "ZVO"
2) "ICUI"
```

```
# suppression d'éléments dont le score est situés dans un intervalle
127.0.0.1:6379> zremrangebyscore performances_nasdaq 34 35
```

```
(integer) 2
```

```
# lecture du rang d'un élément dans un sorted set
# (le 1er élement par ordre de score décroissant a le rang 0)
127.0.0.1:6379> zrank performances_nasdaq NCSM
```

```
(integer) 0
```

Remarque : depuis la version 2.8, Redis propose également des fonctions traitant les intervalles lexicographiques. Ce sont les fonctions *zrangebylex*, *zrevrangebylex*, *zremrangebylex* et *zlexcount*. Avec ces fonctions, le tri est réalisé non pas en fonction du score mais de l'ordre intrinsèque des éléments (typiquement l'ordre alphabétique pour des éléments textuels).

4.7 Redis HyperLogLogs

Un *HyperLogLog* (ou *HLL*) est une structure de données probabiliste permettant d'estimer la cardinalité d'un (très grand) ensemble d'éléments. L'intérêt est d'obtenir une valeur approximée :

- avec une erreur standard qui, dans le cas de l'implémentation Redis, est inférieure à 1% ;
- en ne consommant qu'une quantité de mémoire constante et faible (12 Ko dans le pire des cas) quelle que soit la taille de l'ensemble.

Il s'agit donc d'une alternative extrêmement efficace aux méthodes de comptage exact qui nécessitent généralement d'utiliser une quantité de mémoire proportionnelle au nombre d'éléments à compter. Un HLL Redis est simple d'utilisation et fonctionne suivant les mêmes principes que les autres types de données :

```
# création / ajout unitaire d'élément dans un HLL (HyperLogLogs)
127.0.0.1:6379> pfadd hll_transactions_aapl 09a6023f
```

```
(integer) 1
```

```
# création/ ajout de plusieurs éléments dans un HLL
127.0.0.1:6379> pfadd hll_transactions_aapl 015900e4 02940142 064c022a 044701b6
```

```
(integer) 1
```

```
# estimation du nombre d'éléments ayant été ajoutés dans un HLL
127.0.0.1:6379> pfcount hll_transactions_aapl
```

```
(integer) 5
```

```
# création d'un nouvel HLL
127.0.0.1:6379> pfadd hll_transactions_goog 05c101dd 012b00c8 043801b1
```

```
(integer) 1
```

```
# union de deux HLL pour en produire un nouvel HLL_ : hll_trans_goog_aapl
127.0.0.1:6379> pfmerge hll_trans_goog_aapl hll_transactions_aapl hll_transactions_goog
```

```
OK
```

```
# estimation du nombre d'éléments dans le nouvel HLL
127.0.0.1:6379> pfcount  hll_trans_goog_aapl
```

```
(integer) 8
```

4.8 Redis Streams

Les Streams sont un nouveau type de données Redis introduit dans la version 5.0. Il s'agit d'une structure :

- analogue à celle d'un journal d'évènements (on peut y ajouter des données mais pas modifier les données qui y sont stockées) ;
- munie d'opérations de lecture bloquantes permettant aux consommateurs d'attendre de nouvelles données ajoutées à un flux par les producteurs ;
- implémentant le principe de *consumer groups* (groupes de consommateurs). L'idée étant de permettre à un groupe de clients de coopérer en consommant une partie différente du même flux de messages.

La création / l'ajout de données dans une stream fonctionne suivant le même principe que pour les autres types de données Redis :

```
# création d'une stream Redis qui représente un journal de transactions sur le NASDAQ
# NB : le "*" correspond à l'identifiant de la donnée insérée dans le journal
# cet identifiant est généré automatiquement par la fonction xadd et retourné à l'appelant
127.0.0.1:6379> xadd  transactions_nasdaq * titre AAPL type VENTE prix 207.62
                      nbre 10000 operateur IVE.J
```

```
"1567194173848-0"
```

```
# ajout d'une entrée dans la stream "transactions_nasdaq"
127.0.0.1:6379> xadd  transactions_nasdaq * titre GOOG type ACHAT prix 1188.89
nbre 2000 operateur IVE.J
```

```
"1567194235515-0"
```

```
# lecture du nombre d'éléments dans la stream
127.0.0.1:6379> xlen transactions_nasdaq
```

```
(integer) 2
```

```
# lecture des entrées dont les identifiants sont situées dans un intervalle donné
# (ici  "- +" signifie : l'ensemble des identifiants du plus petit au plus grand)
127.0.0.1:6379> xrange transactions_nasdaq  - +
```

```
1) 1) "1567194173848-0"
   2)  1) "titre"
       2) "AAPL"
       3) "type"
       4) "VENTE"
       5) "prix"
       6) "207.62"
       7) "nbre"
       8) "10000"
       9) "operateur"
      10) "IVE.J"
2) 1) "1567194235515-0"
   2) 1) "titre"
      2) "GOOG"
      3) "type"
      4) "ACHAT"
      5) "prix"
      6) "1188.89
      7) "nbre"
      8) "2000"
      9) "operateur"
     10) "IVE.J"
```

```
# lecture des n (ici avec n=2) dernières entrées d'une ou plusieurs stream
# NB : le dernier paramètre indique le timeout en millisecond (0= jamais)
127.0.0.1:6379> xread count 2 streams transactions_nasdaq 0
```

```
# même résultat que commande précédente
```

```
# lecture bloquante d'une stream (0 ici aussi correspond au timeout - jamais - en ms)
# "$" représente la dernière entrée stockée sans la stream (celle avec l'identifiant max.)
127.0.0.1:6379> xread block 0 streams transactions_nasdaq $
```

L'implémentation de *consumer groups* est réalisée de la manière suivante :

```
# création du consumer group "traders"
# "$" - alias ID de dernier elt - indique que seuls les nouveaux elts de la stream
# seront distribués au groupe.
# "0" (au lieu de "$") diffuserait au groupe tous les elts
127.0.0.1:6379> xgroup create transactions_nasdaq traders $
```

```
OK
```

```
# lecture du dernier elt de la stream par le consumer "trader1" du group "traders"
127.0.0.1:6379> xreadgroup traders trader1 count 1 streams transactions_nasdaq >
```

```
1) 1) "1567194174962-0"
   2)  1) "titre"
       2) "AAPL"
       3) "type"
       4) "ACHAT"
       5) "prix"
       6) "207.41"
       7) "nbre"
       8) "15000"
       9) "operateur"
      10) "COOK.T"
```

4.9 Redis Geohash

Il possible, depuis la version 3.2 de stocker dans Redis des données géospatiales :

```
# Ajoute d'éléments géospatiaux (longitude, latitude, nom) à une clé ("apple_geo")
127.0.0.1:6379> geoadd AAPL_geo  -122.0322895 37.3228934 "Cupertino"
                                 -122.029846 37.331867   "One infinite loop"
                                 -122.009244 37.335547   "Apple Park"
```

```
(integer) 3
```

```
# Mesure de distance entre 2 lieux avec une unité spécifique (ici en km)
127.0.0.1:6379> geodist AAPL_geo "One infinite loop" "Apple Park" km
```

```
"1.8677"
```

```
# Récupération des lieux associés à une clé et situés dans un rayon donné autour d'un point
127.0.0.1:6379> georadius AAPL_geo -122 37 37 km
```

```
1) "Cupertino"
```

Remarques :

- les coordonnées pouvant être indexées sont limitées: les zones très proches des pôles ne sont pas indexables. Les limites exactes spécifiées par EPSG: 900913 / EPSG: 3785 / OSGEO: 41001 sont les suivantes:
  - les longitudes valides vont de -180 à 180 degrés,
  - les latitudes valides vont de -85,05112878 à 85,05112878 degrés ;
- il n'y a pas de commande *geodel* pour supprimer les éléments géospatiaux mais on peut utiliser la commande *zrem* (une structure géospatiale Redis est à la base un *sorted set*).

5. Récupération des données du projet

Pour récupérer de « vraies » cotations boursières qui serviront de base pour la suite du projet, nous avons utilisé l'API (Application Programming Interface) Yahoo Finance en python. Cette API est gratuite, simple à exploiter mais présente quelques limitations en terme de fréquences d'interrogation et du nombre de requêtes pouvant être exécutées simultanément. En pratique, nous nous sommes contentés d'interroger cycliquement (toutes les 10 secondes), pendant un peu plus d'une heure, le service yahoo afin d'obtenir les variations du cours temps réel de l'action Apple Inc. (AAPL). Le code que nous avons développé est le suivant :

```
import time
import os
from yahoofinancials import YahooFinancials
from apscheduler.schedulers.background import BackgroundScheduler

stock="AAPL"
# fichier CSV résultat,ici "AAPL_histo_cours.csv"
outputFile = open(stock+"_histo_cours.csv", "w")

def scrap_stock_current_price():
    # récupération du prix courant de l'action (ici "AAPL")
    # avec le timestamp
    timestamp = str(int(time.time()))
    yahoo_financials = YahooFinancials(stock)
    current_price = str(yahoo_financials.get_current_price())
    # afin de pouvoir importer les données dans un sorted set Redis,
    # chaque ligne du fichier CSV a le format suivant:
    # [timestamp],[prix_action]:[timesamp]
    line= timestamp+","+current_price+":"+timestamp
    print(line)
    outputFile.write(line)
    outputFile.write("\n")
    outputFile.flush()

if __name__ == '__main__':
    # utilisation d'un scheduler qui interroge toutes les 10s
    # le service Yahoo Finance
    scheduler = BackgroundScheduler()
    scheduler.add_job(scrap_stock_current_price, 'interval', seconds=10)
    scheduler.start()
    print('Appuyer sur Ctrl+{0} pour quitter'.format('Break' if os.name == 'nt' else 'C'))

    try:
        # simule une activité afin de maintenir en vie le thread principal
        while True:
            time.sleep(2)
    except (KeyboardInterrupt, SystemExit):
        # pas strictement nécessaire en mode "daemon" mais plus propre
        scheduler.shutdown()
        outputFile.close()
```

Le fichier "AAPL_histo_cours.csv" obtenu à l'allure suivante :

```
1564764872,203.84:1564764872
1564764882,203.92:1564764882
1564764892,203.88:1564764892
1564764902,203.89:1564764902
...
```

6. Insertion en masse

Dans les paragraphes qui viennent, nous insérerons le contenu du fichier "AAPL_histo_cours.csv" généré au chapitre précédent dans un *sorted set* Redis. L'objectif est de stocker une série temporelle décrivant l'évolution du cours de l'action Apple dans une structure de données dont la clé est un timestamp et la valeur le prix de l'action :

```
1564764872 203.84
1564764882 203.92
1564764892 203.88
. . .
```

En pratique, l'utilisation d'un *sorted set* Redis pose un problème : les valeurs doivent être uniques ce qui n'est pas forcément le cas dans notre exemple. En effet l'action peut atteindre le même prix à différents moments d'une session de cotation. Pour surmonter cette difficulté, une solution classique et de prendre comme valeur non pas uniquement la donnée mesurée mais une concaténation du timestamp et de la mesure correspondante. Une simple commande Perl nous permet ainsi de convertir le fichier "AAPL_histo_cours.csv" en une structure de données compatible avec un *sorted set* Redis :

```
awk -F, 'BEGIN {OFS=","} { print " $1 " "  $2 " :" $1}' AAPL_histo_cours.csv
```

```
1564764872 203.84:1564764872
1564764882 203.92:1564764882
1564764892 203.88:1564764892
. . .
```

Ensuite se pose la question de l'écriture en base de données.

Une première solution (naïve) pour effectuer une insertion en masse consiste à utiliser un client Redis standard et à ajouter chaque enregistrement l'un après l'autre. Malheureusement c'est une solution très lente car au temps d'insertion d'un enregistrement il faut ajouter le temps d'aller-retour pour chaque commande unitaire.

Afin d'améliorer l'efficacité du processus, on peut envisager le recours à un pipeline : cela implique que le client exécute les commandes d'écriture tout en lisant les réponses en même temps pour vérifier que l'insertion a été exécutée correctement. Cette option présente un inconvénient majeur : seul un faible pourcentage de clients prend en charge les E / S non bloquantes et tous les clients ne sont pas en mesure d'analyser les réponses de manière efficace afin de maximiser le débit.

Pour toutes ces raisons, la méthode longtemps préférée pour importer des données en masse dans Redis (avant la version 2.6) consistait à générer un fichier texte contenant le protocole Redis au format brut (appelons-le data.txt). Une fois ce fichier construit, l'import était réalisé avec la commande unix *netcat* comme suit :

```
(cat data.txt; sleep 10) | nc localhost 6379 > /dev/null
```

Cependant, ce n'est pas un moyen très fiable d'effectuer une importation en masse, car *netcat* ne sait pas vraiment quand toutes les données ont été transférées et ne peut pas rechercher d'erreurs. De plus la génération d'un fichier conforme au protocole Redis n'est pas sans difficulté (il ne s'agit pas du langage d'interrogation standard tel qu'on le saisit en ligne de commandes).

Heureusement *redis-cli*, à partir de la version 2.6 prend en charge un nouveau mode appelé mode *pipe* conçu pour effectuer efficacement une insertion en masse à partir de commandes standards listées dans un fichier. La commande est très simple :

```
cat liste_commendes_insertion.txt | redis-cli --pipe
```

Afin de mettre en œuvre le mode *pipe*, nous avons écrit un petit script qui lit le fichier "AAPL_histo_cours.csv", génère les commandes Redis d'insertion correspondantes et les transmet à *redis-cli*. L'horodate de début d'insertion et celle de fin sont tracées pour mesurer la durée du traitement :

```
# affiche l'heure de début d'insertion des données (avec une précision à la ms)
date +"%T.%3N"

# insère les données du fichier AAPL.CSV en masse avec resids-cli-pipe
# chaque ligne du fichier AAPL_histo_cours.csv est transformé en une instruction Redis - ex :
# CSV :                   "1564764872,1564764872"
# Sortie du script Perl : "ZADD AAPL_histo_cours 1564764872 203.84:1564764872"
awk -F, 'BEGIN {OFS=","}
    { print "ZADD AAPL_histo " $1 " " $2 ":" $1}' AAPL_histo_cours.csv | redis-cli --pipe

# affiche l'heure de fin d'insertion des données
date +"%T.%3N"
```

```
16:01:09.684
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 623
16:01:09.699
```

Comme indiqué dans les traces, plus de 600 enregistrements on été insérés en 15ms, le mode pipe est donc très performant.

On peut bien sûr vérifier que l'insertion s'est effectivement bien déroulée en exécutant une requête sur la structure "AAPL_histo_cours″ :

```
redis-cli
# lit toutes les données(score entre -infini et +infini) du sorted set AAPL_histo_cours
127.0.0.1:6379> ZRANGEBYSCORE AAPL_histo_cours -inf +inf WITHSCORES
```

```
   1) "203.84:1564764872"
   2) "1564764872"
   3) "203.92:1564764882"
   4) "1564764882"
   5) "203.88:1564764892"
   6) "1564764892"
   . . .
```

7. Gestion de la persistance

Redis offre différentes options pour gérer la persistance des données :

- l'absence de persistance. Avec cette option, les données sont stockées en mémoire mais ne sont jamais écrites sur disque. Elles dont donc perdues à l'arrêt de l'instance Redis ;
- la persistance **RDB** (Redis Data Base) effectue des instantanés à un instant précis de l'ensemble de données à des intervalles spécifiés ;
- la persistance **AOF** (Append Only File) enregistre chaque opération d'écriture reçue par le serveur. Lors d'une reprise sur panne l'ensemble de données est reconstruit en rejouant les commandes une par une. Les commandes sont enregistrées dans le même format que le protocole Redis lui-même. De plus Redis est capable de réécrire le journal en arrière-plan lorsqu'il devient trop volumineux ;
- une combinaison de persistance AOF et RDB dans la même instance. Dans ce cas, lorsque Redis redémarre après une panne, le fichier AOF est utilisé pour reconstruire le jeu de données d'origine. Le fichier RDB permet lui d'avoir des sauvegardes globales, régulières servant en vue par exemple de l'installation d'une nouvelle instance Redis.

7.1 La persistance RDB (Redis Data Base)

Pour activer la persistance RDB, il suffit de saisir la ligne suivante dans le fichier de configuration Redis :

```
save 60 1000
```

Cette commande sauvegarde l'intégralité des données de l'instance sur disque toutes les 60 secondes ou dès que 1000 clés ont été écrites (i.e. ajoutées / modifiées / supprimées).

Le processus de sauvegarde s'effectue en trois étapes :

1. Redis "forke" l'instance courante : on obtient donc un processus parent et un processus enfant
2. Le processus enfant commence à écrire l'ensemble des données dans un fichier RDB temporaire
3. Quand l'écriture est terminée, le processus enfant remplace le fichier de sauvegarde RDB précédent par le nouveau fichier.

7.1.1 Avantages de la persistance RDB

Les avantages de la persistance RDB sont les suivants :

- le format RDB permet de sauvegarder de façon très compacte l'ensemble des données d'une instance Redis sans un fichier unique ;
- les fichiers RDB sont parfaits pour les sauvegardes à court et moyen termes. Par exemple, on peut générer un fichiers RDB toutes les heures pendant les dernières 24 heures et enregistrer un instantané de la base de données chaque jour pendant 30 jours.

Cela permet de restaurer facilement différentes versions de l'ensemble de données en cas de panne ;

- le fait de disposer d'une sauvegarde de l'ensemble des données dans un fichier unique et compact est très commode pour la reprise après sinistre. En effet ce fichier peut facilement être transféré vers des centres de données distants, ou dans le cloud (éventuellement chiffré) ;
- la persistance RDB maximise les performances de Redis puisque le seul travail que le processus parent Redis doit réaliser consiste à "forker" un enfant qui fera tout le reste. L'instance parente n'effectuera jamais les entrées / sorties sur disque ou une opération similaire ;
- la persistance RDB permet des redémarrages plus rapides avec de grands ensembles de données par rapport à AOF.

7.1.2 Inconvénients de la persistance RDB

Les inconvénients de la persistance RDB sont les suivants :

- la persistance RDB n'est pas adaptée aux cas où on souhaite éliminer tout risque de perte des données. En effet lorsqu'une panne survient, toutes les données écrites depuis la dernière sauvegarde sont perdues ;
- la création d'un processus enfant via un fork pour réalise la sauvegarde peut prendre beaucoup de temps si le volume des données est élevé et peut amener Redis à suspendre temporairement l'exécution des requêtes clients.

7.2 La persistance AOF (Append Only File)

Pour activer la persistance AOF (disponible de puis la version 1.1), il suffit de saisir la ligne suivante dans le fichier de configuration Redis :

```
appendonly yes
```

Avec cette option, chaque fois que Redis reçoit une commande modifiant le jeu de données (par exemple, SET), il l'historise dans un fichier journal AOF. Lorsque l'instance redémarre (notamment suite à une panne), Redis relit le fichier AOF pour reconstruire l'état.

Bien sûr ce fichier devient de plus en plus volumineux au fur et à mesure que les opérations d'écriture sont effectuées. Pour pallier à ce problème, Redis est capable d'optimiser le fichier AOF en arrière-plan (en supprimant les commandes obsolètes) sans interrompre le service aux clients. Ce processus d'optimisation, disponible de puis la version 2.2 est, à partir de la version 2.4 déclenchée automatiquement.

Une autre caractéristique importante (et configurable !) de la persistance AOF concerne le mode d'écriture dans le fichier journal via la fonction unix *fsync*. Ce mode est défini à l'aide du paramètre **appendfsync** :

- *appendfsync always* : synchronise le fichier journal chaque fois qu'une nouvelle commande est ajoutée à l'AOF. Très très lent, très sécuritaire.

- *appendfsync everysec* : synchronise le fichier toutes les secondes. Assez rapide (dans 2.4, probablement aussi rapide que la capture instantanée) mais comporte le risque de perdre 1 seconde de données en cas de panne.
- *appendfsync no*: aucune dynvhronisation du fichier journal. L'écriture du fichier est gérée par lu système d'exploitation. Il s'agit de La méthode plus rapide et moins sûre. Dans le cas d'un serveur Linux, le système d'exploitation va effacer les données toutes les 30 secondes mais cela dépend du réglage exact du noyau.

La stratégie suggérée (et celle par défaut) est *appendfsync everysec*. C'est la configuration qui offre le meilleur compromis rapidité / sûreté.

7.2.1 Avantages de la persistance AOF

Les avantages de la persistance AOF sont les suivants :

- la persistance AOF est bien plus sûre (en terme de reprise sur panne) que la persistance RDB: avec l'option AOF par défaut, on perf au pire une seconde de données. De plus il est possible de définir différentes stratégies *fsync*: pas de fsync, fsync à la seconde, fsync à chaque requête afin d'obtenir l'équilibre rapidité / sûreté le plus adapté au cas d'utilisation *;*
- le journal AOF est uniquement un journal, il n'y a donc pas de problème de recherche ni de corruption en cas de panne de courant. Même si le journal se termine par une commande à moitié écrite pour une raison quelconque (disque plein ou autre), l'outil *redis-check-aof* est capable de le réparer facilement ;
- Redis est capable d'optimiser automatiquement et de façon sécurisée le fichier AOF en arrière-plan lorsqu'il devient trop volumineux ;
- AOF contient un journal de toutes les opérations les unes après les autres dans un format facile à comprendre et à analyser.

7.2.2 Inconvénients de la persistance AOF

Les inconvénients de la persistance AOF sont les suivants :

- les fichiers AOF sont généralement plus volumineux que les fichiers RDB équivalents pour le même jeu de données ;
- AOF peut être plus lent que RDB en fonction de la stratégie fsync exacte. En général, avec fsync défini à chaque seconde les performances sont toujours très élevées et, avec fsync désactivé, elles devraient être exactement aussi rapides que RDB, même sous forte charge . Ceci dit, RDB est toujours en mesure de fournir davantage de garanties quant à la latence maximale, même dans le cas d'une charge en écriture importante.

8. Gestion de la réplication

8.1 Système maître / esclave

La réplication Redis est basé sur sur un système maître / esclave qui comporte trois mécanismes principaux :

1. Lorsque le système fonctionne normalement, sans défaillance, un nœud maître met à jour le(s) noeud(s) esclave(s) qui lui est (sont) attaché(s). Toutes les opérations provoquant un changement de donnée(s) sur le nœud maître (écriture ,expiration de clé, etc.) sont reproduites sur le(s) noeud(s) esclave(s) ;
2. Suite à une perte de connexion entre un nœud esclave et un nœud maître (causée par une panne réseau ou par un dépassement de timeout lors des échanges inter nœuds), le nœud esclave essaie de se reconnecter et d'effectuer une resynchronisation partielle. Autrement dit le nœud esclave essaie d'obtenir le flux de commandes qui ont été exécutées sur le nœud maître pendant la déconnexion ;
3. Si une resynchronisation partielle s'avère impossible, le nœud esclave demande une resynchronisation totale. Ceci implique un processus plus complexe qui nécessite que le nœud maître exporte l'ensemble des données dont il dispose, l'envoie au nœud esclave tout en continuant à traiter les requêtes des clients.

Un aspect important de la réplication Redis est que celle-ci est par défaut asynchrone. Cette caractéristique a pour but de privilégier la performance (ce qui correspond aux cas d'utilisations de Redis les plus courants) mais présente l'inconvénient de produire des discordances (temporaires) entre les données des nœuds maîtres et celles des nœuds esclaves. Ceci dit, un nœud maître a accès à chaque instant à l'état de synchronisation de ses nœuds esclaves (i.e. le nœud maître sait quelles commandes ont été exécutées sur les nœuds esclaves). Cette connaissance permet, au besoin, de mettre en place une réplication synchrone.

Parmi les autres points clés de la réplication Redi, on peut noter que :

- les nœuds esclaves peuvent être inter-connectés et que cette inter-connexion peut s'effectuer en cascade (un nœud esclave est connecté à un autre nœud esclave qui est lui-même connecté au nœud maître). Depuis la version 4, Redis garantit que tous les nœuds esclaves (directs ou indirects) reçoivent le même flux de commande ;
- la réplication Redis est non bloquante sur le nœud maître : la réplication n'interrompe pas le traitement des requêtes provenant des clients ;
- la réplication Redis est également largement non bloquante sur les nœuds esclaves un client peut toujours interroger ceux-ci (à condition que le paramétrage réalisé dans le fichier de configuration le permette). Simplement si le client interroge le nœud esclave pendant que celui-ci est en train de se synchroniser, le client obtiendra la donnée précédant la synchro.) ;
- la réplication est mise en œuvre soit dans une optique de passage à l'échelle : dans ce cas les nœuds esclaves sont potentiellement accessibles en mode lecture-seule pour décharger le nœud maître. L'autre rôle de la réplication est d'augmenter la sécurisation et la disponibilité des données ;

- il est possible d'utiliser la réplication pour faire en sorte que les sauvegardes des données ne soient plus réalisées par un nœud maître mais soient déléguées à un nœud esclave.

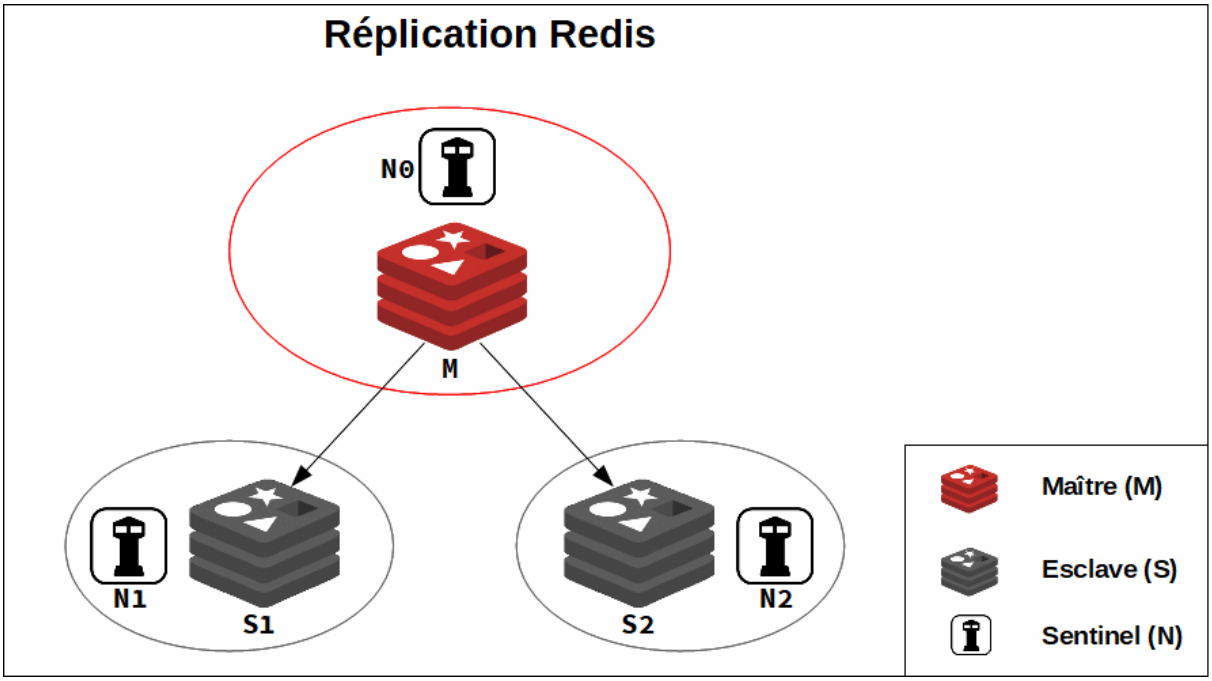
Pour terminer cette partie sur la réplication maître / esclave Redis , signalons que les nœuds maîtres utilisent un identifiant pour chaque flux d'opérations envoyé aux nœuds esclaves. Cet identifiant est incrémenté à chaque envoi de commandes de réplication et permet de définir l'état de synchronisation de chaque nœud du système.

8.2 Redis Sentinel

Sentinel est le mécanisme qui assure la haute disponibilité d'un système Redis. Plus précisément Sentinel a pour rôle de faire en sorte que le système reste opérationnel en cas de panne sans intervention d'un opérateur. Sentinel comporte les fonctionnalités suivantes :

- **surveillance** de l'ensemble des nœuds (maîtres et esclaves) afin de s'assurer qu'ils fonctionnent tous normalement ;
- **notification** à l'administrateur en cas de panne ;
- **basculement automatique** en cas de défaillance d'un nœuds maître : dans ce cas un nœud esclave peut être promu nœud maître (suivant la règle du quorum) et les autres noeuds esclaves sont reconfigurés pour que la réplication se fasse à partir du nouveau nœud maître ;
- **fournisseur de configuration** : les clients Redis s'adressent à une instance sentinel afin d'obtenir l'adresse du nœud maître offrant un service spécifique.

Pour être déployé, le mécanisme Sentinel nécessite au moins trois instances (ceci est imposé par la règle du quorum), chacune étant associée à un nœud du système comme représenté ci-dessous :



L’exécution de processus sentinels requiert un fichier de configuration, dont un exemple basique est présenté ci-dessous :

```
# sentinel monitor <master-group-name> <ip> <port> <quorum>
# où quorum représente le nombre de Sentinels qui doivent s'entendre sur le fait que le
# maître n'est pas joignable avant de démarrer une procédure de basculement.
sentinel monitor master1 127.0.0.1 6379 2
sentinel down-after-milliseconds master1 60000
sentinel failover-timeout master1 180000
sentinel parallel-syncs master1 1

sentinel monitor master2 192.168.1.3 6380 4
sentinel down-after-milliseconds master2 10000
sentinel failover-timeout master2 180000
sentinel parallel-syncs master2 5
```

Notons que fichier (appelons-le "sentinel.conf") ne liste que les nœuds maîtres, les nœuds esclaves quant à eux sont auto-découverts.

Pour démarrer le mécanisme sentine, deux options sont disponibles :

```
# 1) démarrer les processus sentinels avec la commande qui leur est dédiée :
redis-sentinel /chemin/vers/sentinel.conf

# ou

# 2) démarrer les processus avec un mode particulier de l’exécutable redis-server :
redis-server /chemin/vers/sentinel.conf --sentinel
```

Finalement notons les points suivants :

- le port par défaut des processus sentinel est 26379 ;
- l’utilisation de sentinel nécessite un support au niveau des bibliothèques client. Les bibliothèques les plus répandues sont compatibles mais ce n’est pas nécessairement le cas de toutes.

9. Gestion du partitionnement

Historiquement, le partitionnement n'a pas été considéré comme une fonctionnalité à intégrer en priorité dans la feuille de route du développement Redis. Dans les premières versions, les deux principales solutions recommandées par la documentation officielle consistaient à :

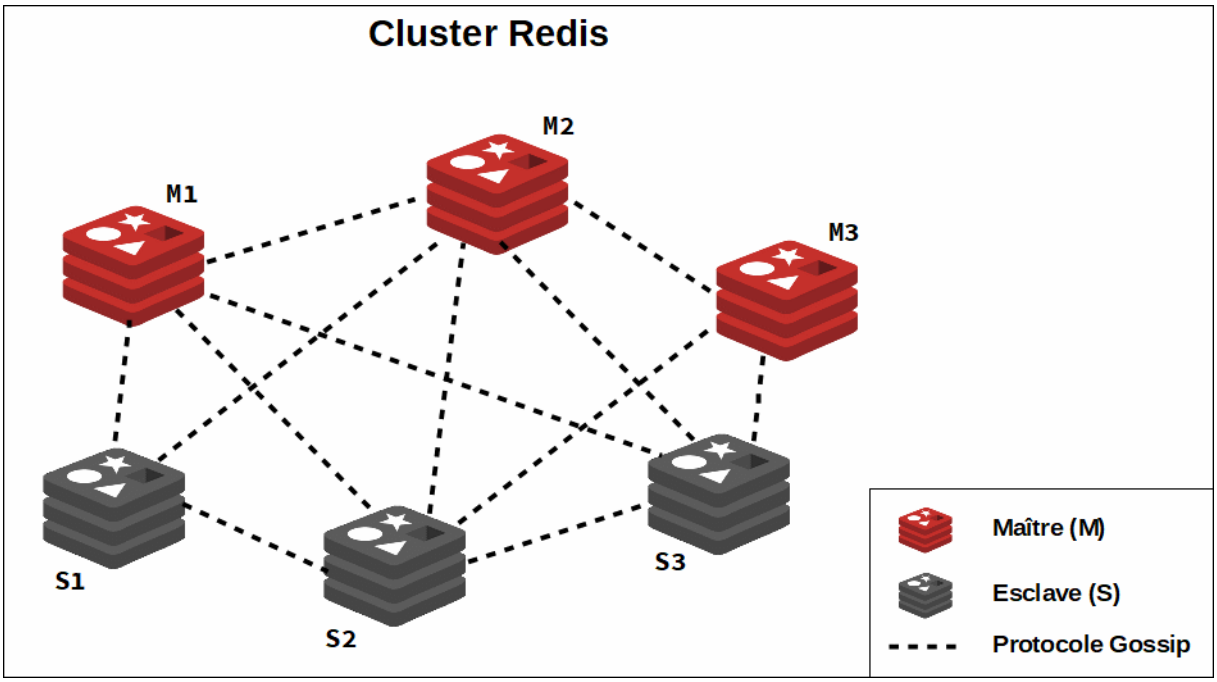
- **gérer le partitionnement côté client** : dans ce cas les clients sélectionnent eux-même directement le bon nœud pour écrire ou lire une clé donnée. De nombreux clients Redis implémentent cette stratégie ;
- **recourir à un proxy** : avec cette option, les clients Redis envoient leurs requêtes à un proxy, supportant le protocole Redis, qui se charge de transmettre la requête à l'instance Redis appropriée conformément au schéma de partitionnement configuré, puis de renvoyer les réponses au client. La solution la plus connue implémentant ce type de stratégie est Twemproxy développée par Twitter ;

Finalement il s'est avéré que proposer une solution de partitionnement intégrée à Redis côté serveur était incontournable pour certains cas d'utilisations. En effet cela répond à deux objectifs importants :

1. créer des bases de données beaucoup plus volumineuses, en utilisant la somme de la mémoire de nombreux ordinateurs. Sans partitionnement, Redis est limité à la quantité de mémoire de la machine sur laquelle l'instance est exécutée ;
2. dimensionner la puissance de calcul ainsi que la bande passante réseau sur une grappe de serveurs.

Dans cette optique, Redis a introduit avec la version 3, l'architecture **Redis Cluster**. Dans cette architecture un client peut envoyer une requête à n'importe quelle instance Redis du système. Cette instance veille ensuite à transmettre la requête au bon nœud. Redis Cluster implémente une forme hybride de routage de requête, avec l'aide du client (la requête n'est pas directement transmise d'une instance Redis à une autre, mais le client est redirigé vers le bon nœud). De plus cette solution logicielle prend en charge la reprise sur panne.

Un point fondamental du Redis Cluster est qu'il s'agit d'un **réseau maillé** avec des nœuds maîtres et des nœuds esclaves. Cela signifie que pour un cluster Redis de 6 nœuds composé de 3 maîtres et 3 esclaves, chaque nœud, quel que soit son statut de réplication, comporte cinq connexions TCP sortantes et cinq entrantes comme représentée dans la figure ci-après :



Les connexions TCP d'un nœud sont toujours vivantes et lui permettent de répondre continuellement aux pings des autres nœuds du cluster. Chacun de ces pings, appelé *heartbeat paquet*, contient entre autres l'identifiant du nœud émetteur, un numéro de message produit par un compteur, la description de l'espaces de hachage servi par l'émetteur, le port de base la vue de l'émetteur sur l'état du cluster (*up* / *failing* /*failed*) et l'identifiant du nœud maître si l'émetteur est un esclave.

Pour éviter une croissance exponentielle des messages entre les nœuds du cluster, Redis utilise un protocole **gossip** (protocole épidémique en français). Le principe de fonctionnement de ce mécanisme est analogue à celui de la diffusion d'une rumeur : supposons que toutes les heures, des employés de bureau se retrouvent à la machine à café. Chaque employé est associé à un autre, choisi au hasard, et partage les derniers potins. Au début de la journée, Alice lance une rumeur : elle dit à Bob qu'elle croit que Charlie se teint la moustache. Lors de la pause café suivante, Benoît informe David, tandis qu'Alice répète l'idée à Eve. Après chaque pause, le nombre de personnes qui ont entendu la rumeur double (si on ne prend pas en compte le fait que la même personne a pu entendre la même rumeur plusieurs fois). Il s'agit donc d'un mode de diffusion très efficace... Mais avec une qualité de service aléatoire : le protocole ne garantit pas une diffusion complète sous un délai garanti.

Nous n'approfondirons pas plus ici l'étude de Redis Cluster qui est un outil puissant mais présente aussi des limitations non négligeables :

- les bibliothèques clientes Redis ne sont pas toutes compatibles avec ce système ;
- les opérations impliquant plusieurs clés ne sont généralement pas prises en charge. Par exemple, il n'existe pas de façon directe d'effectuer l'intersection entre deux ensembles s'ils sont stockés dans des clés mappées sur différentes instances Redis ;
- les transactions Redis impliquant plusieurs clés ne peuvent pas être utilisées ;
- la granularité du partitionnement est la clé, il est donc impossible de partager un jeu de données avec une seule clé énorme (ex.: *sorted set* volumineux).

10. Mécanisme pub / sub

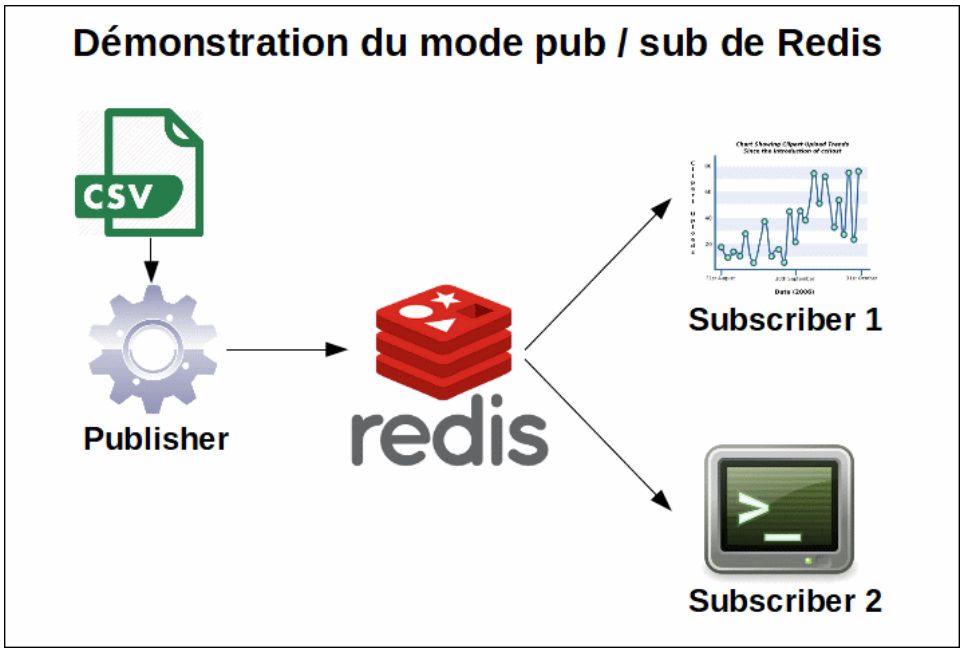
Le mécanisme pub / sub (publish / subscribe, littéralement : publier-s'abonner) permet à un composant logiciel d'émettre un message sur un canal (en anglais "channel", on emploie également le terme "topic"). Le message est ensuite diffusé à tous les composants logiciels qui sont abonnés (i.e surveillent) ce canal. Le système pub /sub présente deux avantages importants :

- c'est un mécanisme évènementiel : un client n'a pas besoin d'interroger cycliquement le serveur pour obtenir les mises à jours des données. Il suffit au client d'être abonné à un canal sur lequel le serveur poste les changements de valeurs pour être notifié automatiquement et en temps réel ;
- l'émetteur et le(s) abonné(s) sont découplés : l'émetteur ne sait pas combien ni qui sont ses abonnés. Réciproquement un abonné n'a pas besoin de connaître l'émetteur (adresse IP, port, système d'exploitation, langage de programmation, etc…) pour se connecter à un canal et recevoir des messages. Cela s'avère particulièrement intéressant en terme de passage à l'échelle (ajout / suppression dynamique d'abonnés) et de souplesse d'utilisation (intégration de systèmes hétérogènes).

Afin d'illustrer la mise en œuvre d'un mécanisme pub / sub avec Redis, nous avons développé un système assez simple avec trois composants logiciels écrit en python :

- un **publisher** qui parcourt le fichier "AAPL_histo_cours.csv" (cf. chapitre 5) et envoie séquentiellement, ligne par ligne, les évolutions du cours de l'action Apple Inc. ;
- un premier abonné (**subscriber 1**) qui affiche les mises à jour de l'action sous forme d'une courbe de tendance ;
- un deuxième abonné (**subscriber 2**) qui affiche les mises à jour de l'action dans une fenêtre de terminal.

L'ensemble est représenté dans le schéma ci-dessous ;



Le code des différents composants est listé ci-dessous :

publisher.py

```
import redis
import csv
import datetime
import time

# connexion à l'instance Redis locale
queue = redis.StrictRedis(host='localhost', port=6379, db=0)
# création du canal sur lequel seront diffusés les évolutions
# du cours de l'action AAPL (lues dans le fichier "AAPL_histo_cours.csv")
channel = queue.pubsub()

# lecture ligne du fichier csv :
# pour chaque ligne lue (correspondant à la cotation de l'action AAPL
# à un instant donné), un message est envoyé sur le canal Redis
with open('AAPL.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    for row in readCSV:
        # row est de la forme :
        #    ['timestamp', 'price']
        # ex :
        #    ['1564771092', '203.8179']
        price = row[1]
        print(datetime.datetime.now(),"PUB", price)
        # envoi du message sur le canal "AAPL_current_price"
        # NB : on envoie uniquemnt le prix de l'action pas le timestamp.
        # Le but est de simuler un système diffusant le cours instantané de l'action
        # en temps réel.
        queue.publish('AAPL_current_price', price)
        time.sleep(0.01)

while 1:
    time.sleep(1000)
```

subscriber1_graphique.py

```
import redis
import time
import matplotlib.pyplot as plt
from drawnow import drawnow

def makeFig():
    # dessine la figure
    plt.plot(xList,yList)
    plt.axis([0,625,195,205])

# rend la figure interactive
plt.ion()
# crée la figure
fig=plt.figure()

# liste des abscisses (timestamps)
xList=list()
# liste des ordonnées (cours de l'action AAPL)
yList=list()

# connexion à l'instance Redis locale
r = redis.StrictRedis(host='localhost', port=6379, db=0)
p = r.pubsub()
# abonnement au canal "AAPL_current_price"
p.subscribe('AAPL_current_price')
```

```
# variable correspondant au timestamp associé au cours de l'action
# pour simplifier le traitement on se contente d'incrémenter un compteur
t=-1

# boucle sur la réception des messages
while True:
    message = p.get_message()
    if message :
        # NB : le 1er message reçu correspond à une notification
        # de connexion au canal Redis. C'est la raison pour laquelle
        # on vérifie si t>=0 (lors du 1er message t==-1)
        if t >= 0  :
            y = float(message['data'])
            # ajoute la nouvelle donnée dans les cordonnées des points
            xList.append(t)
            yList.append(y)
            # actualise la figure contenant la courbe de tendance
            drawnow(makeFig)
        t=t+1
    time.sleep(0.01)
```

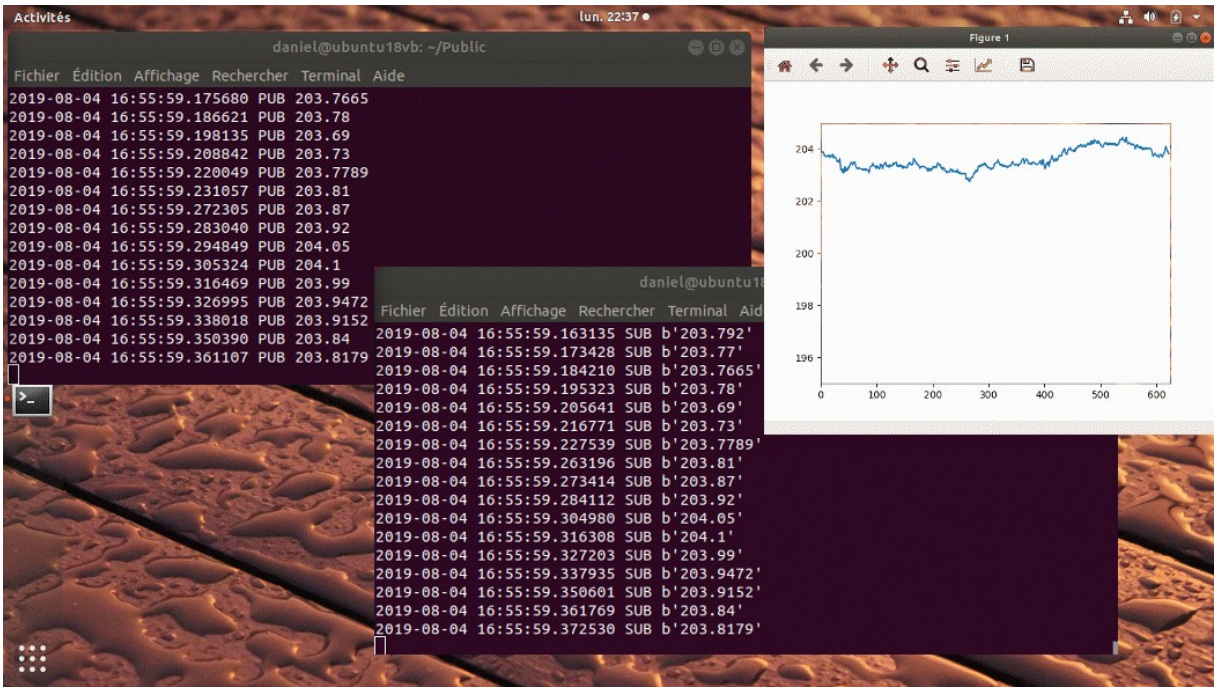
subscriber2_console.py

```
import redis
import time
import datetime

# connexion à l'instance Redis locale
r = redis.StrictRedis(host='localhost', port=6379, db=0)
p = r.pubsub()
# abonnement au canal "AAPL_current_price"
p.subscribe('AAPL_current_price')
# variable correspondant au timestamp associé au cours de l'action
# pour simplifier le traitement on se contente d'incrémenter un compteur
t=-1

# boucle sur la réception des messages
while True:
    message = p.get_message()
    if message :
        # NB : le 1er message reçu correspond à une notification
        # de connexion au canal Redis. C'est la raison pour laquelle
        # on vérifie si t>=0 (lors du 1er message t==-1)
        if t >= 0  :
            # affichage du message sur la sortie standard
            print(datetime.datetime.now(),"SUB", message['data'])
        t=t+1
    time.sleep(0.01)
```

On peut alors exécuter ces différents scripts python (en commençant par les scripts *subscriber*...) comme représenté dans la capture d'écran ci-après :



Deux remarques sur cette démo. :

- On note que le délai entre l'envoi d'un message par le "publisher" et sa réception par un "subscriber" est extrêmement court, par exemple :

Publisher	Subscriber
16:55:59.326995 PUB 203.9472	10:55:59.337935 SUB '203.9472'
16:55:59.338018 PUB 203.9152	10:55:59.350601 SUB '203.9152'
16:55:59.358390 PUB 203.84	16:55:59.361769 SUB '203.84'
16:55:59.361107 PUB 203.8179	16:55:59.372530 SUB '203.8179'

  En fait ce délai correspond essentiellement aux 10 ms de temporisation dans la boucle de lecture des messages reçus par le "subscriber" ;

- L'animation de la courbe de tendance présente une latence importante : en effet la figure est redessinée intégralement à chaque ajout de point. Pour une utilisation réelle, il faudrait optimiser ce mécanisme de rafraîchissement graphique en choisissant une solution plus performante.

11. Conclusion

Redis est non seulement un système NoSQL à part entière extrêmement populaire et performant mais de part sa versatilité est souvent utilisé en complément à d'autres systèmes NoSQL :

- avec MongoDB, Redis permet par exemple d'implémenter des compteurs ou un suivi d'activité ;
- ElasticSearch, Logstash et Kibana, Redis est utilisé comme file de messages entre entre une instance Logstash chargée de collecter les traces et une autre instance Logstash chargée elle de stocker les données dans ElasticSearch.

Redis est aussi fréquemment employé en tant que cache pour des serveurs d'application et/ou des serveurs Web.

Sa légèreté autorise à l'installer pour implémenter une seule fonctionnalité très spécifique (ex : tableau de score dans une application internet).

Une originalité de Redis particulièrement intéressante dans un contexte temps réel est le mécanisme de publication / abonnement de messages. Il s'agit d'une de ses caractéristiques qui fait de Redis un véritable « couteau suisse » dans le monde des systèmes big data.

Cependant, le parti pris du minimalisme et de la légèreté présente forcément quelques revers. Ainsi les solutions Redis de haute disponibilité basées sur le partitionnement sont relativement récentes et n'ont pas forcément le même niveau de maturité que celles d'autres systèmes NoSQL qui les ont intégrées dès le début.

Nul doute cependant que les prochaines évolutions de Redis répondront de mieux en mieux aux besoins d'architectures complexes capable de gérer de façon sécurisé des données distribuées toujours plus volumineuses.