M2 TIW/DS - Big Data Analytics
TP Vélocité
Traitement de flux via *Apache STORM*

UCBL - Département Informatique de Lyon 1 – 2017

L'objectif de ce TP est de vous familiariser avec les topologies STORM permettant de traiter des flux de données et de vous sensibiliser aux problèmes de congestions d'opérateurs qui peuvent apparaître en fonction de l'évolution du débit du flux en entrée.

---

1 Préliminaire

1.1 Organisation du TP

Ce TP se déroule en trois temps.
Le premier temps consiste à prendre en main Apache STORM. Pour cela, vous pourrez exécuter la requête continue qui vous est proposée *via* la topologie *TopologyT1* qui vous est fournie.
Une fois cette prise en main faite, vous allez pouvoir créer vos propres requêtes. Il s'agit de la partie

sur la course des tortues (section 2.2.4). L'objectif est de voir différents types d'opérateurs avec ou sans état et avec ou sans fenêtrage. Le débit du flux en entrée restera normalement gérable pour votre application.
La dernière partie du TP, s'intéresse au calibrage d'Apache STORM pour retarder la congestion éventuelle de certains opérateurs du fait de l'augmentation du débit du flux en entrée. Il s'agit de la partie sur la course de lièvres. L'objectif est de voir comment il est possible de configurer Apache STORM pour modifier le degré de parallélisme de certains opérateurs ainsi que l'utilisation de ressources supplémentaires.

Pour réaliser ce TP, vous avez besoin de récupérer sur TOMUSS :
— l'adresse IP de votre VM principale : **"STORM_IP_Princ"**
— votre numéro de course : **"STORM_Course"**
— votre numéro de dossard : **"STORM_Dossard"**
— l'adresse IP de vos VM secondaire : **"STORM_IP_Sec"**

Le TP peut être réalisé au choix en JAVA ou en SCALA.

1.2 Recommandation

Le déploiement de vos topologies sur le cluster STORM peut prendre du temps. La gestion des logs distribués et le caractère peu ergonomique de l'interface web peuvent être un frein au débugage de votre code. Il est fortement recommander de regrouper vos méthodes au sein d'une classe (par exemple *TortoiseManager*) et d'effectuer des tests unitaires nécessaires pour valider votre code avant de le déployer.

1.3 Modalités de rendu

Pour ce TP, il vous est demandé de fournir un bref rapport dont le canevas est disponible ici : http://liris.cnrs.fr/nicolas.lumineau/teaching/bda/tpStorm/nom1-nom2-RenduTPstorm.zip. Ce rapport devra être complété et le PDF généré sera à déposer sur TOMUSS avant le **17/12/2017**.

2 Prise en main d'Apache STORM

2.1 Brefs rappels sur Apache STORM

Il vous est rappelé ci-dessous quelques éléments/concepts importants pour comprendre le fonctionnement d'une application reposant sur Apache STORM.

La requête continue que vous souhaitez mettre en place pour interroger un flux est représentée par une **topologie**. Cette topologie est un *workflow* d'opérateurs qui vont s'enchaîner pour générer le flux de sortie de votre topologie. Ces opérateurs sont appelés **spout** quand ils servent de connecteur aux flux d'entrée et **bolt** quand ils servent à implémenter un traitement. La coordination de la ou des topologies se fait grâce à un **nimbus** qu'il faut voir comme un *master*. Ce nimbus gère les allocations des opérateurs sur les nœuds de traitements appelés **supervisors**. Pour gérer la coordination, le master a besoin de monitorer ce qui se passe au niveau des supervisors, pour cela il exploite les informations issues de **zookeeper**. Vous, en tant qu'utilisateur/développeur, vous avez besoin de visualiser l'activité de votre application, pour cela, Apache STORM propose une interface web appelée **ui**.

Ainsi, pour lancer une application Apache STORM, vous avez besoin de lancer :

— un **zookeper** pour le monitoring
— un **nimbus** pour la coordination
— un **ui** pour la visualisation
— un **supervisor** pour l'execution
— une **topologie** comprenant un ou plusieurs **spouts** et **bolts** qui exprime une requête

2.2 Mise en place d'une topologie

2.2.1 Environnement

Vous trouverez la documentation sur Apache STORM à l'adresse :
http://storm.apache.org/releases/1.0.3/index.html

Dans ce TP vous disposez d'une VM sur le cloud du département dont l'adresse IP qui vous a été attribué en début de séance et spécifiée sur TOMUSS dans la cellule "STORM_IP_Princ".

Pour vous connecter à votre VM vous utiliserez une connexion *ssh* avec la clé *pedabdcloud* qui vous a été transmis par mail :

```
ssh -i <keyPath>/pedabdcloud ubuntu@192.168.**.***
```

Comme il va être nécessaire de lancer plusieurs applications, il est conseillé d'utiliser *screen*[1] pour le lancement de STORM et d'éviter un nombre trop important de terminaux ouvert en *ssh*.

Vous pouvez créer les sessions suivantes :
— *screen -S zook* : pour créer une session dédiée à *zookeeper*
— *screen -S nimb* : pour créer une session dédiée au *nimbus* de *Storm*
— *screen -S ui* : pour créer une session dédiée à l'interface *ui* de *Storm*
— *screen -S super* : pour créer une session dédiée à un *supervisor* de *Storm*
— *screen -S topo* : pour créer une session dédiée à la topologie que vous souhaiter exécuter sur *Storm*
— *screen -S startS* : pour créer une session dédiée au lancement de votre flux
— *screen -S stopS* : pour créer une session dédiée à l'arrêt de votre flux

Pour rappel, il vous suffira de faire *screen -r <nom_de_session>* pour vous rattacher à la session que vous souhaitez.

De plus, pour faciliter la navigation dans les répertoires, vous disposez des alias suivants dans votre *.bashrc* :
— alias cdst='cd /lib/apache-storm-1.0.2/bin'
— alias cdzk='cd /lib/zookeeper-3.3.6/bin'

Dans le répertoire */home/ubuntu* de votre VM, vous avez :
— Pour le traitement de flux :
  — le répertoire *./lib/zookeeper-3.3.6* contenant les fichiers de l'application de monitoring ZOOKEEPER.
  — le répertoire *./lib/apache-storm-1.0.2* contenant les fichiers de l'application de Apache STORM.
  — le programme *./stormTP-0.1-jar-with-dependencies.jar* contenant une topologie qui sera exécutée par*Apache STROM* (c'est ce jar que vous aurait à générer pour soumettre une topologie).

1. https://doc.ubuntu-fr.org/screen

— l'archive *Storm.zip* contenant le projet maven avec les classes JAVA qui vous permettront de définir vos topologies *Apache STROM* pour traiter les flux.

— Pour le partage de ressources (CPU, RAM) avec un tiers :

— le répertoire *./forOthers/apache-storm-1.0.2* contenant les fichiers de l'application de Apache STORM utilisable par d'autres que vous.

— Pour le monitoring du flux de sortie :

— le script *main.js* qui permet de lancer une application NODE.JS pour écouter du flux de sortie (sur le port 9002) généré par votre topologie STORM.

En local sur votre machine, vous pouvez récupérer l'archive *StormJAVA.zip* ou *StormSCALA.zip* en fonction du langage choisi pour développer vos topologies. A partir d'une de ces archives, vous pouvez importer le projet MAVEN dans votre IDE Eclipse. Le *pom.xml* de votre projet est configuré pour générer une topologie STORM dans le programme *stormTP-0.1-jar-with-dependencies.jar* que vous trouverez dans le répertoire *target*. Après toute modification de vos classes définissant votre topologie, il sera nécessaire de générer le *stormTP-0.1-jar-with-dependencies.jar* pour JAVA ou SCALA via la commande *'mvn assembly :assembly'* puis de transférer le .jar générée sur votre VM (via *scp*).

Remarque : Certains tests fournis seront à commenter le temps que vous implémentiez les fonctionnalités.

2.2.2 Paramètres de configuration

Avant toute configuration, il est nécessaire de vous assurer que le fichier **/etc/hosts** soit correct.

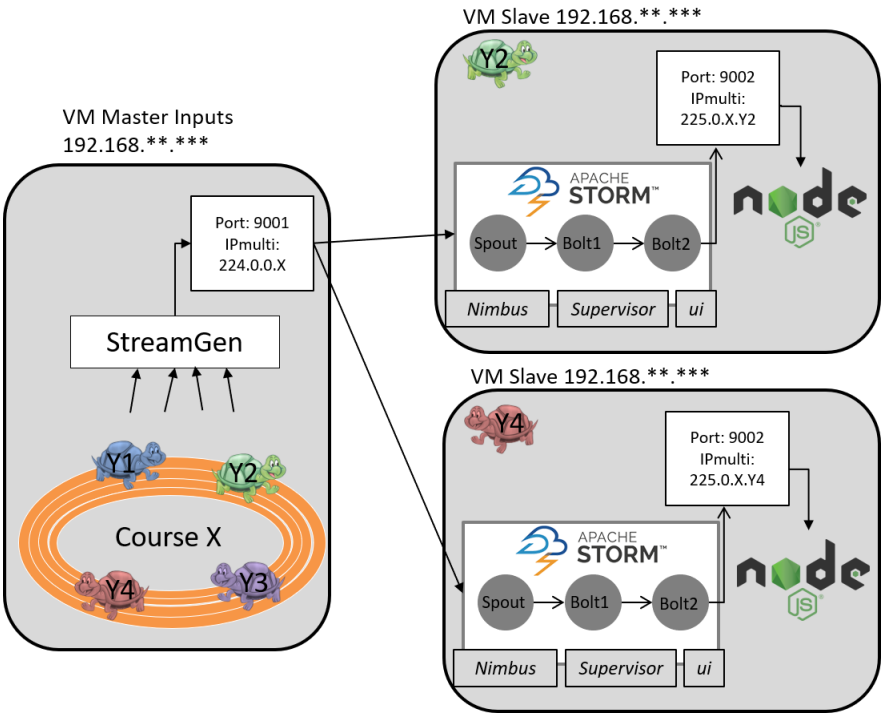

FIGURE 1 – Architecture globale

Ports et IP multicast

Comme le montre la Figure 1 , pour pouvoir configurer votre VM, vous avez besoin :

— de l'adresse IP de votre VM

— de l'adresse IP multicast d'émission du flux de votre course. Il s'agit de l'adresse 224.0.0.X sur le port 9001, avec X qui correspond à votre numéro de course.

— de l'adresse IP multicast du flux de sortie que vous allez émettre. Il s'agit de l'adresse 225.0.X.Y sur le port 9002, avec X qui correspond à votre numéro de course et Y qui correspond à votre numéro de dossard.

Ainsi, si sur TOMUSS vous avez **1** pour STORM_Course et **4** pour STORM_Dossard, votre adresse IP multicast d'émission du flux d'entrée est **224.0.0.1** et votre adresse IP multicast d'émission du flux de sortie est **225.0.1.4**.

STORM
En ce qui concerne la configuration de STORM, elle est spécifiée dans le fichier *./lib/apache-storm-1.0.2/conf/storm.yaml*.
La documentation est disponible ici :
http://storm.apache.org/releases/1.0.3/Setting-up-a-Storm-cluster.html

2.2.3 Lancement des applications

Lancement de STORM

— Dans le répertoire *./lib/zookeeper-3.3.6/bin*, exécuter :

```
./zkServer.sh start
```

— Dans le répertoire *./lib/apache-storm-1.0.2/bin*,

- pour lancer votre master Nimbus, exécuter :

```
./storm nimbus
```

- pour lancer votre interface de monitoring de Storm [2] sur la page *http ://<votreIP> :8080*, exécuter :

```
./storm ui
```

- pour lancer un nœud de traitement sur votre VM, exécuter :

```
./storm supervisor
```

Lancement de votre topologie Comme indiqué précédemment, vous avez une topologie qui vous est proposée dans le fichier
/home/ubuntu/stormTP-0.1-jar-with-dependencies.jar. Cette topologie est composée de 3 opérateurs :

— *StreamSimSpout* qui écoute le port 9001 sur votre IP multicast du flux d'entrée et bufferise les messages UDP reçus.

— *TestStatelessBolt* qui ne fait que transmettre le message reçu à l'opérateur suivant.

— *ExitBolt* qui émet à l'adresse IP multicast du flux de sortie sur le port 9002 les messages reçus.

Pour lancer cette topologie, exécuter dans le répertoire *./lib/apache-storm-1.0.2/bin* :

```
./storm jar /home/ubuntu/stormTP-0.1-jar-with-dependencies.jar
      stormTP.topology.TopologyT1 <X> <Y>
```

où *<X>* correspond à votre numéro de course et *<Y>* correspond à votre dossard dans la course.

Remarque : Vous disposez dans le package *stormTP.operator.test* de quatre exemples de classes implémentant respectivement un opérateur *stateless*, *stateful*, *stateless* avec fenêtrage, *stateful* avec fenêtrage.

2. http://www.malinga.me/reading-and-understanding-the-storm-ui-storm-ui-explained/

Lancement du monitoring Pour vous assurer que tout se passe correctement, vous disposez de deux moniteurs.

Pour pouvoir visualiser les tuples en sortie de votre topologie, exécuter dans votre *home* :

```
node main.js <ipMulticastOUT>
```

où *<ipMulticastOUT>* correspond à votre IP multicast du flux de sortie.

Pour visualiser le trafic dans votre topologie, vous avez l'interface *ui* comme représenté sur la Figure 2 et la Figure 3.

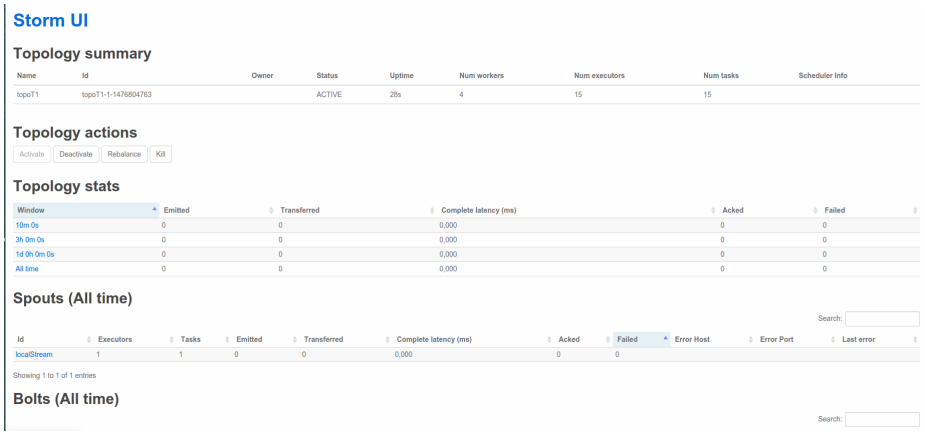

FIGURE 2 – Interface Storm UI : Page d'accueil

Sur la page d'accueil et dans la section "Topology summary", vous devez avoir *topoT1* qui apparaît. En cliquant sur *topoT1*, vous accédez à l'interface dédiée à cette topologie représentée sur la Figure 3. Les deux boutons importants dans cette interface, sont les boutons :

— **Kill** : qui vous permet d'arrêter votre topologie.

— **Show visualization** : qui permet de visualiser graphiquement votre topologie.

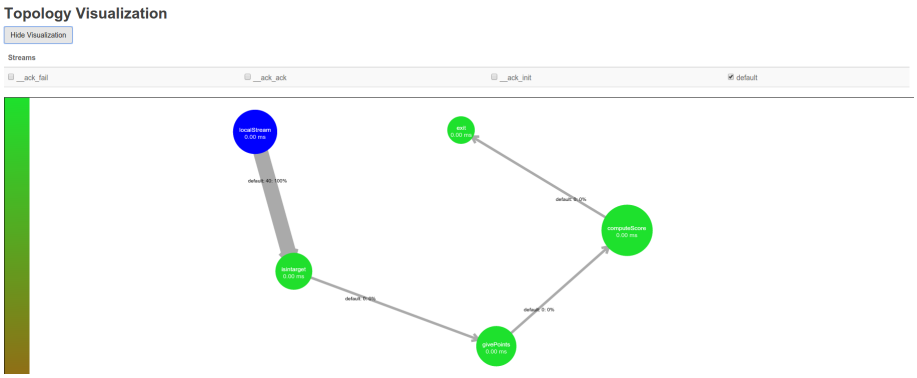

FIGURE 3 – Interface Storm UI : Visualisation d'une topologie

2.2.4 Bilan

A ce niveau du TP, vous devez pourvoir voir :

— le trafic entre les trois opérateurs (*MasterInputStreamSpout*, *NoFilterBolt*, *ExitBolt*) de votre topologie via l'interface ui de STORM

— les tuples émis par votre topologie dans le terminal où vous avez exécuté *main.js*

3 Course des tortues

Pour cette partie du TP, nous considérons une pseudo piste d'athlétisme. Cette piste est composée de couloirs fragmentés en cellule. Chaque couloir est composée de 254 cellules numérotées. Cette piste est le terrain d'une course des tortues. Une tortue ne peut que rester sur la cellule où elle se trouve ou avancer sur la cellule suivante. L'évolution de la course est décrite par un flux de données dont le schéma est (id, top, position, nbAvant, nbApres, nbTotal) avec *id* un entier correspondant au dossard de la tortue qui l'identifie, *top* un entier qui indique le numéro d'observation des tortues sur la piste, *position* un entier qui correspond à la cellule courante où se trouve la tortue (Attention, la position ne permet pas de déterminer le classement de la tortue, car la piste est circulaire et qu'une tortue peut avoir au moins un tour d'avance), *nbDevant* un entier qui indique le nombre de tortues se trouvant devant la tortue dans le classement, *nbDerriere* un entier qui indique le nombre de tortues se trouvant derrières dans le classement et *total* indique le nombre total de tortues en piste.

Remarque importante : pour ne pas gaspiller les ressources (CPU, RAM) de votre VM, et sauf contre indication, il sera important de stopper les topologies précédentes avant de lancer une nouvelle topologie (cf bouton KILL sur l'interface UI de STORM) [3].

3.1 Filtrer sa tortue

Il s'agit ici d'implémenter un opérateur *stateless*.
Définir un bolt, nommé "MyTortoiseBolt" qui récupère dans le flux la tortue qui vous a été attribuée en début de séance. Les tuples retournés par ce bolt ont pour schéma (id, top, nom, position, nbAvant, nbApres, nbTotal). Ce qui correspond au schéma en entrée augmenté du nom de la tortue (correspondant à nomBinôme1-nomBinôme2) qui vous a été attribuée.

Par exemple, si vous avez le dossard 1 et que vous êtes le binôme 'Yves Atrovite' et 'Ella Paltan', à partir de l'objet JSON reçu :

{ tortoises :[
{"id" :0,"top" :523,"position" :4,"nbDevant" :8,"nbDerriere" :1,"total" :10},
{"id" :1,"top" :523,"position" :11,"nbDevant" :4,"nbDerriere" :4,"total" :10},
{"id" :2,"top" :523,"position" :15,"nbDevant" :1,"nbDerriere" :7,"total" :10},
{"id" :3,"top" :523,"position" :5,"nbDevant" :7,"nbDerriere" :2,"total" :10},
{"id" :4,"top" :523,"position" :11,"nbDevant" :4,"nbDerriere" :4,"total" :10},
{"id" :5,"top" :523,"position" :14,"nbDevant" :3,"nbDerriere" :6,"total" :10},
{"id" :6,"top" :523,"position" :248,"nbDevant" :0,"nbDerriere" :9,"total" :10},
{"id" :7,"top" :523,"position" :8,"nbDevant" :6,"nbDerriere" :3,"total" :10},
{"id" :8,"top" :523,"position" :15,"nbDevant" :1,"nbDerriere" :7,"total" :10},
{"id" :9,"top" :523,"position" :1,"nbDevant" :9,"nbDerriere" :0,"total" :10}
] }
vous devez produire l'objet JSON :
{"id" :1,"top" :523, "nom" :"Paltan-Atrovite", position" :11,"nbDevant" :4,"nbDerriere" :4,"total" :10}

A partir de *ExitBolt*, créer le bolt *Exit2Bolt* qui prend en entrée des tuples de schéma (id, top, nom, position, nbDevant, nbDerriere, total) et qui produit en sortie un tuple de schéma (json) dont la valeur retournée correspond l'objet JSON attendu.

Définir la topologie TopologyT2, qui permettra de tester votre bolt "MyTortoiseBolt" avec *MasterInputStreamSpout* et *Exit2Bolt*.

3. Mettre 0 comme valeur dans le dialogue et valider pour arrêter la topologie immédiatement

Tester votre topologie TopologyT2 (après avoir arrêté votre topologie TopologyT1).

3.2 Calcul du rang

Il s'agit ici d'implémenter un opérateur *stateless*.
Définir un bolt, nommé "GiveRankBolt" qui détermine le classement de votre tortue sur la piste. Les tuples retournés par ce bolt ont pour schéma (id, top, nom, rang, nbTotal). L'*id* correspond à l'identifiant de votre tortue, le *top* correspond au top d'observation de votre tortue, le *nom* correspond au nom de votre tortue et le *rang* correspond à la chaîne de caractère indiquant le rang de la tortue. En cas d'égalité, le rang des tortues *ex æquo* sera suffixé par le mot 'ex'.

A partir de *ExitBolt*, créer le bolt *Exit3Bolt* qui prend en entrée des tuples de schéma (id, top, nom, rang, nbTotal) et qui produit en sortie un tuple de schéma (json) dont la valeur retournée correspond l'objet JSON attendu.

Ainsi pour le tuple reçu (1, 5, 'Toto',2 ,0 ,9 ,10), vous devrez générer un tuple (1, 5, 'Toto', '1', 10). Pour le tuple reçu (1, 10, 'Toto', 6, 2, 6, 10), vous devrez générer un tuple (1, 10, 'Toto', '3ex', 10).

Définir la topologie TopologyT3, qui permettra de tester votre bolt "GiveRankBolt" avec *MasterInputStreamSpout*, *MyTortoiseBolt* et *Exit3Bolt*.

Tester votre topologie TopologyT3 (après avoir arrêté votre topologie TopologyT2).

3.3 Affectation des points bonus

Il s'agit ici d'implémenter un opérateur *stateful*.
Définir un bolt, nommé "ComputeBonusBolt" qui calcule le nombre de points bonus cumulés par votre tortue. Les tuples retournés par ce bolt ont pour schéma (id, top, nom, score). L'affectation des points bonus se fait le de la manière suivante : tous les 15 tops, le classement de la tortues est transformé en point correspondant au nombre total de participants moins le rang dans le classement. Ainsi, pour 10 participants, le ou les premiers auront 9 points supplémentaires, le ou les seconds auront 8 points supplémentaires et ainsi de suite.

A partir de *ExitBolt*, créer le bolt *Exit4Bolt* qui prend en entrée des tuples de schéma (id, top, nom, points) et qui produit en sortie un tuple de schéma (json) dont la valeur retournée correspond l'objet JSON attendu.

Ainsi pour les tuples reçus {(1, 15, 'Toto', '1', 10), (1, 30, 'Toto', '3ex', 10), (1, 45, 'Toto', '2', 10), (1, 60, 'Toto', '3', 10)} vous devrez générer les tuples : (1, 15, 'Toto', 9), (1, 30, 'Toto', 15), (1, 45, 'Toto', 23) puis (1, 60, 'Toto', 30).

Définir la topologie TopologyT4, qui permettra de tester votre bolt "ComputeBonusBolt" avec *MasterInputStreamSpout*, *MyTortoiseBolt*, *GiveRankBolt* et *Exit4Bolt*.

Tester votre topologie TopologyT4 (après avoir arrêté votre topologie TopologyT3).

3.4 Vitesse moyenne

Il s'agit ici d'implémenter un opérateur *stateless* avec fenêtrage.
Définir un bolt, nommé "SpeedBolt, qui détermine la vitesse moyenne de la tortue exprimée en cellule par top calculée sur 10 tops et ce, tous les 5 tuples reçus. Les tuples retournés par ce bolt ont pour schéma (id, nom, tops, vitesse), avec *tops* une chaîne de caractères de la forme "$t_i - t_{i+9}$" où t_i et t_{i+9}

correspondent respectivement au premier et au dernier top considérés dans le calcul.

Ainsi pour les tuples reçus { (1,1,'Toto',1,0,9,10) ; (1,2,'Toto',1,0,9,10) ; (1,3,'Toto',1,0,9,10) ; (1,4,'Toto',2,0,9,10) ; (1,5,'Toto',3,0,9,10) ; (1,6,'Toto',3,0,9,10) ; (1,7,'Toto',3,0,9,10) ; (1,8,'Toto',4,0,9,10) ; (1,9,'Toto',4,0,9,10) ; (1,10,'Toto',4,0,9,10) ; (1,11,'Toto',4,0,9,10) ; (1,12,'Toto',5,0,9,10) ; (1,13,'Toto',5,0,9,10) ; (1,14,'Toto',5,0,9,10) ; (1,15,'Toto',6,0,9,10) ; },
vous devez générer les tuples (1, 'Toto', '1-10', 0.40) puis (1, 'Toto', '6-15', 0.3) ;

A partir de *ExitBolt*, créer le bolt *Exit5Bolt* qui prend en entrée des tuples de schéma (id, top, nom, vitesse) et qui produit en sortie un tuple de schéma (json) dont la valeur retournée correspond l'objet JSON attendu.

Définir la topologie TopologyT5, qui permettra de tester votre bolt "SpeedBolt" avec *MasterInputStreamSpout* et *Exit5Bolt*.

Tester votre topologie TopologyT5 (après avoir arrêté votre topologie TopologyT4).

3.5 Evolution du rang

Il s'agit ici d'implémenter un opérateur *stateful* avec fenêtrage.
Définir un bolt, nommé "RankEvolutionBolt", qui détermine l'évolution de la moyenne du rang ("En progression", "Constant" ou "En régression") de la tortue calculée sur une fenêtre de 10 secondes. Les tuples retournés par ce bolt ont pour schéma (id, nom, tops, evolution). L'*id* correspond à l'identifiant de la tortue, le *nom* correspond au nom de votre tortue, le *tops* correspond à la concaténation du plus petit top observé dans la fenêtre avec le plus grand top observé dans la même fenêtre et le *evolution* correspond à la chaîne de caractères indiquant si la tortue est :

- en progression, pour traduire qu'il a gagné au moins une place dans le classement au bout de 10 secondes
- constant, pour traduire qu'il a la même place au bout de 10 secondes
- en régression, pour traduire qu'il a perdu au moins une place dans le classement au bout de 10 secondes

Ainsi pour les tuples reçus { (1,1,'Toto','1ex',10) ; (1,2,'Toto','1',10) ; (1,3,'Toto','1',10) ; (1,4,'Toto','2',10) ; (1,5,'Toto','1',10) ; (1,6,'Toto','1',10) } pendant 30 secondes, vous devez générer un tuple (1, 'Toto', '1-6', 'Constant') ;

A partir de *ExitBolt*, créer le bolt *Exit6Bolt* qui prend en entrée des tuples de schéma (id, top, nom, points) et qui produit en sortie un tuple de schéma (json) dont la valeur retournée correspond l'objet JSON attendu.

Définir la topologie TopologyT6, qui permettra de tester votre bolt "RankEvolutionBolt" avec *MasterInputStreamSpout*, *MyTortoiseBolt*, *GiveRankBolt* et *Exit6Bolt*.

Tester votre topologie TopologyT6 (après avoir arrêté votre topologie TopologyT5).

Remarque :L'erreur de type 'java.lang.IllegalArgumentException : topology.bolts.message.id.field.name is not set...' avec l'utilisation d'une 'TumblingWindow' peut se résoudre via la déclaration de l'identifiant du tuple (.withMessageIdField).

4 Course des lièvres

Pour cette partie du TP, nous conservons le contexte de course mais cette fois les tortues sont remplacées par des lapins. Dans ce contexte, les observations sont bien plus nombreuses et fréquentes.

Dans la partie précédente, les ressources étaient suffisantes pour absorber le débit du flux. Dans cette partie, ce ne sera plus le cas. Certains opérateurs vont se retrouver dans un état de congestion, c'est-à-dire que le nombre de tuples à traiter est plus important que le nombre de tuples qu'il est en capacité de traiter. Pour résoudre ce problème, vous allez dans un premier temps modifier le degré de parallélisme de l'opérateur congestionné pour avoir plusieurs *thread* et ainsi exécuter l'opérateur en parallèle. Ensuite, vous évaluerez l'impact d'ajouter un *supervisor* supplémentaire.

4.1 Podium

Afin que les performances réseaux ne biaisent pas les tests que vous aurez à effectuer, vous utiliserez le spout *HareSpout* qui vous est fourni dans le package *stormTP.operator*. Les objets JSON émis par ce spout sont quasi identiques à ceux émis par le *MasterInputStreamSpout* avec un nom en plus.

Nous allons dans un premier temps définir un bolt qui effectue une opération coûteuse. Définir un bolt de type 'stateless', nommé "ComputePodiumBolt", qui pour chaque tuple reçu, détermine le podium correspondant (c'est-à-dire, les trois meilleurs rangs). Dans le cas où plusieurs lièvres sont sur la même marche du podium, les noms s'afficheront triés par ordre alphabétique. Les tuples émis par ce bolt ont pour schéma (json) et représentent des objets JSON de la forme :

{ "top" :"long", "marcheP1" :[{"nom" :"string"}],
"marcheP2" :[{"nom" :"string"}], "marcheP3" :[{"nom" :"string"}]}

Par exemple, à partir de l'objet JSON reçu :
{ "rabbits" :[
{"id" :0,"top" :123,"nom" :"RogerRabbit","position" :4,"nbDevant" :8,"nbDerriere" :1,"total" :10},
{"id" :1,"top" :123,"nom" :"BugsBunny","position" :11,"nbDevant" :4,"nbDerriere" :4,"total" :10},
{"id" :2,"top" :123,"nom" :"Panpan","position" :15,"nbDevant" :1,"nbDerriere" :7,"total" :10},
{"id" :3,"top" :123,"nom" :"Caerbannog","position" :5,"nbDevant" :7,"nbDerriere" :2,"total" :10},
{"id" :4,"top" :123,"nom" :"Oswald","position" :11,"nbDevant" :4,"nbDerriere" :4,"total" :10},
{"id" :5,"top" :123,"nom" :"Jojo","position" :14,"nbDevant" :3,"nbDerriere" :6,"total" :10},
{"id" :6,"top" :123,"nom" :"Coco","position" :248,"nbDevant" :0,"nbDerriere" :9,"total" :10},
{"id" :7,"top" :123,"nom" :"JudyHopps","position" :8,"nbDevant" :6,"nbDerriere" :3,"total" :10},
{"id" :8,"top" :123,"nom" :"LapinBlanc","position" :15,"nbDevant" :1,"nbDerriere" :7,"total" :10},
{"id" :9,"top" :123,"nom" :"Basil","position" :1,"nbDevant" :9,"nbDerriere" :0,"total" :10}
] }
vous devez produire l'objet JSON :
{"top" :"123","marcheP1" :[{"nom" :"Coco"}],"marcheP2" :[{"nom" :"LapinBlanc"}, {"nom" :"Panpan"}],
"marcheP3" :[{"nom" :"Jojo"}] }

Lancer la topologie TopologyE1 pour tester votre bolt "ComputePodiumBolt" avec *HareSpout* et *ExitInLogBolt*.

En regardant dans l'ui l'affichage de votre topologie, que remarquez-vous sur l'état du noeud correspondant au bolt "ComputePodiumBolt" (si vous ne voyez rien d'anormal... attendez un peu ! Des 'failed' devraient apparaître).

4.2 Déphasage des tuples

Pour tester les performances de notre bolt "ComputePodiumBolt", nous allons calculer le nombre de tuples déphasés (i.e. qui n'ont pas pu être traité dans le temps imparti). Pour cela, ajouter une instruction dans la méthode *fail* du spout pour pouvoir journaliser un message du type :
"[PodiumFail] Failure (msg : x) after y seconds"
avec x le numéro du message source de l'échec et y le temps en seconde depuis le lancement de la topologie. Vous pourrez récupérer l'information via un grep dans les différents fichiers worker.log des

répertoires 6700, 6701, 6702 et 6703 dans logs/workers-artifacts/ttt où ttt correspond au nom de votre topologie.

Par la suite, nous partons du principe que la topologie est congestionné à partir de 10 échecs observés.

En combien de temps votre topologie est congestionnée ?

4.3 Gestion de la congestion

4.3.1 Modification du parallélisme

Dans un premier temps, nous allons modifier le parallélisme de l'opérateur "ComputePodiumBolt". Modifier votre topologie TopologyE1, pour associer 4 workers et 4 Executors à votre opérateur.

Qu'observez-vous ?

4.3.2 Ajout d'un supervisor distant

Pour finir, configurer le storm du répertoire "forOthers" qui vous a été attribué comme VM secondaire (**"STORM_IP_Sec"** dans TOMUSS). Pour cela, modifier le conf/storm.yaml du slave et affecter le nom complet de votre VM (stormXXX.novalocal) au nimbus.seeds. Il est aussi nécessaire de modifier le /etc/hosts du slave pour déclarer votre VM. Il sera également nécessaire de modifier le /etc/hosts de votre VM pour déclarer le slave. Lancer un supervisor qui sera connecté à votre nimbus. Vérifier sur l'ui que le nouveau supervisor est bien répertorié dans la section "Summary supervior".

Modifier votre topologie *TopologyE1* pour qu'elle puissent s'exécuter sur 8 workers (et non plus 4).

Qu'observez-vous ?

=>=>=>=>=>=>F=>=>=>=>=>=>\
=>=>=>=>=>=>I=>=>=>=>=>=>=>=>=|=>=>=>=>=>FIN
=>=>N=>=>=>=>=>=>=>=>=>=>=>=>/