Projet de fin d'études

Ensai - 3ème année

Statistique et Ingénierie des Données

DEBS Grand Challenge 2017

Élèves :
Clément Franger
Quentin Grail
Andréa Knoertzer
Quentin Wachowiak

Encadrants :
Yann Busnel
Nicoló Rivetti

26 Mars 2017

Résumé

Ce rapport a pour objectif de présenter la solution que nous avons développée pour participer au Grand Challenge de la conférence Distributed and Event-Based Systems (DEBS) 2017. Le but du DEBS Grand Challenge 2017 est d'implémenter une application distribuée de détection d'anomalies en temps réel sur des machines en chaîne de production.

Nous avons développé notre solution en Java avec le framework de traitement de données en temps réel Apache Flink. La communication de notre système avec l'extérieur (réception des données en entrée et envoi des anomalies) se fait au travers de files RabbitMQ. Sur chacune des machines que nous surveillons, la détection d'anomalies s'effectue indépendamment sur les données d'une cinquantaine de capteurs. Elle est opérée de manière non supervisée et se décompose en trois grandes étapes. Tout d'abord les données sont séparées dans plusieurs clusters par une méthode de type *K-means*. Ensuite nous entraînons un modèle de Markov sur les données afin de calculer les probabilités de transition entre les différents clusters. Une transition est considérée comme anormale lorsqu'elle passe en dessous d'un certain seuil de probabilité défini par les organisateurs. Notre système est encapsulé dans un conteneur Docker pour pouvoir être envoyé et testé sur la plateforme de *benchmark* HOBBIT, issue d'un projet de recherche européen.

La dernière partie de ce rapport est consacrée à la mesure des performances de notre programme. Au moment de l'écriture de ce rapport, le déploiement de notre système sur quatre machines physiques a été testé mais les vitesses d'exécution précises ont été mesurées uniquement en local. Ces mesures nous ont été extrêmement utiles pour détecter les points faibles et les points forts de notre système. L'ensemble du code Java de ce projet est disponible dans le dépôt Github à l'adresse suivante : `https://github.com/kent930/PFE`

Les images Docker utilisées pour le déploiement sont accessibles à l'adresse suivante : `https://hub.docker.com/u/debs17gcensai/`

Remerciements

Nous tenons à remercier Nicoló Rivetti pour son accompagnement et sa disponibilité tout au long de ce projet. Malgré la distance, il a su répondre avec patience à nos nombreuses questions et nous fournir tous les outils nécessaires pour participer au DEBS Grand Challenge 2017. Nous remercions aussi Yann Busnel qui a su s'assurer du bon déroulement du projet au cours de ces quelques semaines.

Enfin nous remercions les organisateurs du challenge pour cette opportunité de nous confronter à une problématique industrielle et pour tous les moyens mis à disposition des participants. Les enjeux technologiques du développement d'une solution au challenge ont été des défis stimulants pour mettre en œuvre notre démarche de *data scientists*.

Table des matières

Introduction

Le Distributed and Event-Based Systems (DEBS) [4] est une conférence internationale de recherche. Au cours de celle-ci est organisée une compétition de développement autour d'une application distribuée. Ce concours annuel, créé en 2007, a pour but d'encourager la recherche concernant les systèmes distribués. Une partie de la conférence est consacrée à la présentation des meilleurs projets proposés. Les thèmes de la compétition ont été variés mais ont toujours concerné le traitement de données en temps réel.

En 2012, l'objectif du challenge était de fournir des statistiques utiles aux spectateurs et aux managers d'une équipe de football à partir de flux de données issus de capteurs posés sur le ballon et les chaussures des joueurs. En 2015 le challenge avait pour but d'analyser des données envoyées en direct par les taxis de New-York pour déterminer, entre autres, les routes les plus empruntées durant les 30 dernières minutes ou encore les zones les plus profitables pour les taxis.

Cette année, le DEBS Grand Challenge 2017 [5] s'appuie sur des données issues de la plasturgie. L'objectif de la solution à implémenter est de détecter en temps réel les anomalies de fonctionnement sur des machines en chaîne de production. Le sujet a été annoncé le 1er décembre 2016, les équipes peuvent s'inscrire depuis le 17 décembre et doivent rendre leur solution avant le 14 avril 2017.

Les données réelles sont issues de capteurs posés par la société Weidmüller [11] sur deux types de machines industrielles : des machines à injection de matière (*injection molding machines*) et des machines d'assemblage (*assembly machines*). Dans un souci de confidentialité, les données qui sont mises à notre disposition ne sont pas des données réelles, mais des données simulées par AGT International [1] avec un modèle construit à partir des données réelles issues des capteurs Weidmüller. Celles-ci nous parviennent sous la forme d'un flux RDF contenant les valeurs d'une centaine de capteurs pour chaque machine.

Le challenge de cette année est co-organisé par Hobbit [7] qui fournit la plateforme d'évaluation et de benchmark des projets. La plateforme matérielle fournie par Hobbit pour évaluer les différentes solutions se compose de quatre machines virtuelles possédant deux CPU de huit coeurs chacunes. Il convient donc de déployer le calcul de manière distribuée afin de

maximiser l'utilisation des ressources matérielles à disposition.

Dans un premier temps, nous exposerons plus en détail le contexte du DEBS Grand Challenge, les données fournies pour répondre à la problématique ainsi que les résultats attendus. Ensuite, nous présenterons notre démarche, le choix du framework et les étapes du développement. Puis, nous détaillerons l'architecture du système, pour enfin présenter ses performances.

1 Contexte du Challenge

1.1 Détection d'anomalies sur machines industrielles

Pour chaque machine à injection de matière et chaque machine d'assemblage sont récupérées les valeurs d'une cinquantaine à une centaine de capteurs. Le système doit renvoyer les valeurs anormales pour certains de ces capteurs. Les deux principaux algorithmes à implémenter nous sont imposés par le challenge. Passée la phase de démarrage, le système travaille sur une fenêtre de W observations.

La détection d'anomalies se déroule en trois étapes comme illustré par la Figure 1 :

— La première étape consiste à classifier W observations en K clusters avec une méthode de type K-means. Le nombre maximum d'itérations du K-means est fixé à M. Les sorties des différents capteurs d'une même machine sont à traiter indépendamment, donc chaque K-means s'effectue sur W observations d'un même capteur d'une même machine. Si nous surveillons 50 capteurs par machine, à chaque fois que notre système reçoit un groupe d'observations [1], 50 K-means par machine sont lancés indépendamment et en parallèle.

— La deuxième étape est l'entraînement d'un modèle de Markov à partir des différents clusters du K-means.

— La troisième étape calcule, à partir du modèle markovien, les probabilités des chaînes composées des 1 à N dernières transitions et compare chaque probabilité avec un seuil donné dans les métadonnées. Si une probabilité est inférieure au seuil alors la dernière observation rentrée dans le fenêtre est une anomalie et doit être retournée par le système.

1. Un groupe d'observations est l'ensemble des valeurs reçues à un horodatage donné.

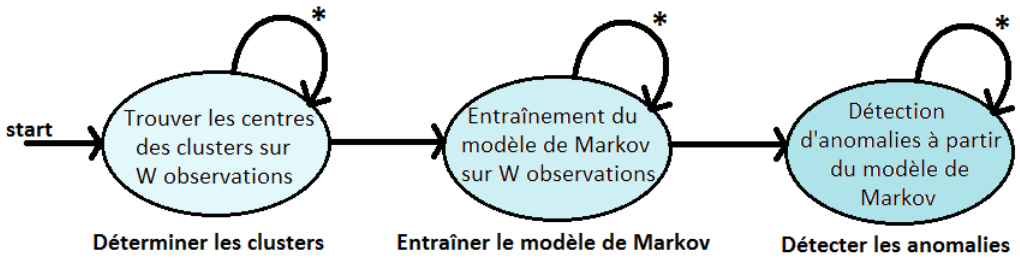


FIGURE 1 – Les trois étapes de la détection d'anomalies

La détection d'anomalies est donc paramétrée par trois valeurs, fixées à la discrétion des organisateurs :

— W le nombre d'observations dans la fenêtre glissante ;

— M le nombre maximum d'itérations pour la convergence du K-means ;

— N le nombre maximum de transitions de la chaîne de Markov considérées pour calculer la probabilité d'anomalie.

Faire varier ces valeurs permettra aux utilisateurs de déterminer la combinaison la plus efficace pour la détection d'anomalies. Deux autres paramètres propres à chaque capteur existent, ils figurent donc dans les métadonnées. Pour chaque dimension :

— K est le nombre maximal de cluster à trouver pour le K-means ;

— Td (*threshold*) est le seuil en deçà duquel une probabilité de transition est déclarée comme anomalie.

1.2 Flux de données

Les données du challenge sont fournies par les sociétés Weidmüller et AGT International. La première fournit un modèle de données issues de capteurs posés sur des machines industrielles fournissant des valeurs de température, pression, fréquence, volume, vitesse, etc. La seconde se charge de simuler un jeux de données pour le challenge à partir du modèle formé par les données de Weidmüller. Si pour le challenge, les participants ne disposent donc pas d'un jeu de données réelles, ils disposent bien d'un jeu de données réalistes.

1.2.1 Format

RabbitMQ [9] est le framework choisi par les organisateurs pour communiquer avec notre système, en entrée et en sortie. C'est un logiciel de messagerie open-source. Les données arrivent par une première file d'attente RabbitMQ, et les anomalies ressortent par une seconde file d'attente. Ces files d'attentes sont créées par HOBBIT et notre solution doit pouvoir écouter celle en entrée et remplir celle en sortie. Le système de messagerie illustré par la figure 2 est simple. La file représentée par le rectangle rouge doit être nommée et fait office de boîte aux lettres. Les messages envoyés y sont stockés, qu'il y ait ou non quelqu'un pour les écouter. La réception consiste à ouvrir cette boîte et lire dans l'ordre d'arrivée les messages qui s'y trouvent.

FIGURE 2 – Schéma de fonctionnement de RabbitMQ

Les données que nous recevons depuis RabbitMQ sont en Turtle (*Terse RDF Triple Language*), une syntaxe qui permet une sérialisation de modèles RDF. Toutes les informations sont donc contenues sous la forme de triplets RDF composés comme ceci : (sujet, prédicat, objet). Les données sont décrites selon l'ontologie OWL déclinées en plusieurs modules [8] (IoT Core, Industry 4.0). Nous verrons plus précisément en troisième partie comment ces triplets sont analysés par notre système.

1.2.2 Entrées

Nous disposons de deux types de données en entrée :

— Des métadonnées qui fournissent des informations sur les caractéristiques des machines sont reçues d'abord dans un ficher séparé, une seule fois avant chaque flux de données réelles. Elles contiennent les numéros de capteurs d'intérêt pour chaque machine, ainsi que certains paramètres utilisés par les algorithmes de détection d'anomalies, comme le nombre

de centres utilisés pour le K-means et le seuil limite en probabilité à partir duquel une transition doit être considérée comme anormale.

— Les données réelles sur lesquelles les calculs s'effectuent sont reçues en mode flux par la suite. Elles contiennent les valeurs prises par l'ensemble des capteurs des machines, ainsi que leurs horodatages.

Le fichier exemple de métadonnées qui nous est fourni est le descriptif pour la détection d'anomalies sur une machine à injection. Sur cette dernière, il y a 55 capteurs d'intérêt, qui sont les seuls à retenir pour procéder à la détection d'anomalies. Le nombre de centres à utiliser pour le K-means est de 13 pour chaque capteur, et une transition est considérée comme anormale lorsque sa probabilité est inférieure à 0.73.

Le premier fichier exemple de données réelles contient les valeurs du total des 117 capteurs d'une machine à injection de matière à 63 horodatages différents. Le second contient les valeurs des 117 capteurs d'une machine à injection de matière à 639 horodatages différents. Dans ces fichiers, les horodatages sont séparés de 10 secondes. Le plus lourd (107,2Mo) correspond donc à la surveillance d'une machine durant environ 1 heure 45.

Ce fichier des données est composé de plusieurs blocs appelés groupes d'observations. Chaque groupe d'observations est caractérisé par l'horodatage auquel l'information a été envoyée par les machines. Ensuite on retrouve une multitude de triplets RDF avec diverses informations sur les 120 capteurs de la machine. Pour plus de précision sur la structure RDF des données en entrée, un schéma est disponible Figure 15 en Annexe A.

1.2.3 Sorties

En sortie du programme, seules sont envoyées dans une queue RabbitMQ les anomalies détectées parmi les valeurs transmises par les capteurs des machines industrielles. Suivant les mêmes directives que pour les données en entrée, les données en sorties sont des tuples RDF écrits avec la syntaxe Turtle. Le schéma précis reliant chaque anomalie retournée au capteur, à la machine et à l'horodatage concernés est fourni Figure 16 de l'Annexe A. Il doit être suivi dans un souci d'homogénéité des résultats pour l'évaluation. Les sorties doivent également être ordonnées chronologiquement.

1.3 Évaluation de la solution

La procédure d'évaluation est entièrement réalisée par la plateforme HOBBIT. Chaque solution est envoyée sur la plateforme pour être soumise à des tests. Une fois ces tests terminés, la plateforme fournit un ensemble d'indicateurs sur les performances obtenues. Il est aussi possible de tester l'exactitude des résultats produits, un test pouvant être correct ou incorrect (ni la liste ni le pourcentage d'erreurs ne sont accessibles). Parmi les solutions correctes, un score est attribué à partir d'une formule annoncée dès le lancement du challenge :

$$total_{score} = \left(\frac{debit_1}{latence_1} + \frac{debit_2}{latence_2} + ...\right)_{scenario1} + \left(\frac{debit_1}{latence_1} + \frac{debit_2}{latence_2} + ...\right)_{scenario2}$$

Le système sera donc confronté à deux **scénarios**, et doit conserver une **latence** la plus faible possible face à des **débits** d'injection de données croissants.

1.3.1 Les deux scénarios de test

Il existe deux scénarios auquel notre système doit s'adapter au mieux :

— Scénario 1 : le nombre de machines est constant sur toute la durée du flux.

— Scénario 2 : le nombre de machines est susceptible de varier dynamiquement durant la réception du flux.

La différence entre les deux scénarios est la variabilité du volume de données reçu en entrée du système. L'existence du scénario 2 valorise les systèmes capables d'adapter dynamiquement leur répartition de charge (ou *load balancing*). Il dissuade de prendre comme unique référence le nombre de machines fixe du scénario 1, ce qui conduirait à "sur-optimiser" l'allocation des ressources de calcul.

1.3.2 Le débit et la latence

Débit
ou vitesse d'injection de données

Le débit correspond au nombre de groupes d'observations envoyés par seconde.

Pour mettre le système à l'épreuve, des tests sont réalisés à débit croissant. Il s'agit du débit d'injection des données, à bien différencier du débit d'arrivée des messages RabbitMQ. Par exemple, si les données proviennent de 10 machines à injection, chacune ayant 120 capteurs renseignés, un seul groupe d'observations correspond à environ 9440 messages RabbitMQ. Avec un débit d'injection à 10 groupes d'observations par minute, le système reçoit toutes les 6 secondes un lot de 9440 messages sur sa file d'entrée. C'est cet intervalle d'attente entre deux groupes d'observations qui est réduit pas à pas sur la plateforme.

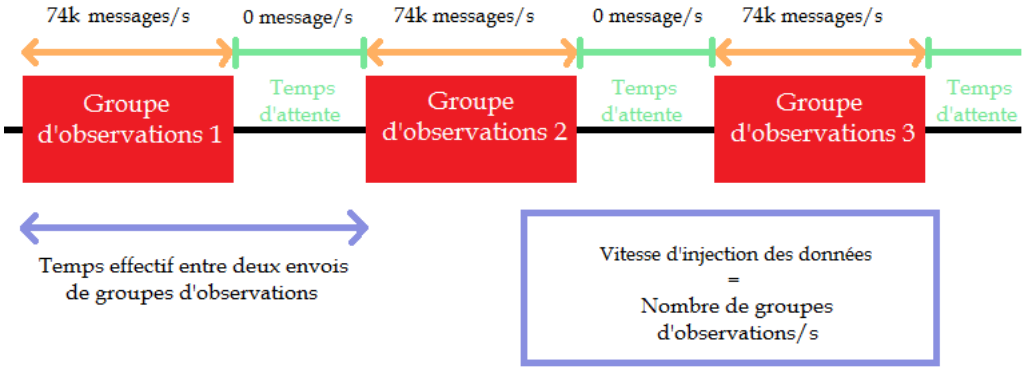


FIGURE 3 – Flux d'entrée des messages

Sur la figure 3, les blocs rouges représentent les phases de réception de message. Durant cette phase, le système reçoit environ 74 000 messages par seconde. Ce débit RabbitMQ est lié à des contraintes techniques d'envoi et réception. La temps d'attente (en vert) est le temps restant avant l'envoi du prochain lot de messages. Si l'envoi d'un groupe d'observations était instantané, notre rythme effectif d'injection des données serait strictement égal au temps d'attente.

Latence totale
ou temps de traitement
Intervalle de temps entre la réception du dernier message d'un groupe d'observations et la sortie correspondante de la détection d'anomalies.

Le critère de performance qui départagera les systèmes de détection d'anomalies est le temps de traitement des données, mesuré par la latence. Ce temps dépend du débit d'entrée des observations.

Pour un débit fixé, la latence est mesurée pour chaque groupe d'observations qui passe dans le système. Il s'agit d'une latence totale et non par tuple : elle représente le temps de traitement du groupe d'observations entier. Une fois le flux terminé, la latence du système est estimée par sa moyenne empirique [2], qui est conservée pour chaque débit testé. Au final, le rapport débit-latence correspond au volume de données traité rapporté au temps de traitement : il s'agit donc de la vitesse de traitement du système.

Les organisateurs du challenge se réservent la possibilité d'examiner les solutions proposées avant de choisir l'ordre de grandeur du volume des données envoyées et de leur rythme d'injection. On peut cependant raisonnablement imaginer que les solutions seront poussées à leurs limites.

2. Il s'agit de la somme des latences mesurées rapportée au nombre de groupes d'observations traités.

2 Démarche

Pour le développement d'une solution, les seules contraintes techniques posées par le Grand Challenge sont liées à la plateforme d'évaluation, concernant la forme des entrées et des sorties.

2.1 Choix du framework : Flink

2.1.1 Départager entre les possibles

L'enjeu du concours est de pouvoir monter en débit tout en conservant une latence suffisamment faible sur des données diffusées en mode flux. Les candidats sont libres de la technologie utilisée, l'esprit du DEBS Grand Challenge est à l'innovation.

Première question : quels frameworks gèrent les données diffusées en continu ?

Une brève recherche sur Internet permet d'apprendre que Apache Spark Streaming [10], Apache Storm [13] et Apache Flink [6] sont les plus reconnus en début d'année 2017. Spark Streaming s'annonce comme une infrastructure libre de calcul distribué qui peut travailler sur la totalité des données en même temps et exécute l'ensemble des opérations en mémoire. Il ne s'appuie sur des disques que lorsque sa mémoire n'est plus suffisante, ce qui réduit les temps de latence entre les traitements et contribue à sa rapidité. Storm est aussi un système libre de calcul distribué qui permet de traiter les données en temps réel. Il s'appuie sur des topologies construites comme des graphes orientés dont les arêtes sont des flux étiquetés contenant des données, et les sommets des étapes de traitement [13]. Flink est une infrastructure libre nativement conçue pour traiter des flux diffusés en continu ou en batch.

Deuxième question : lequel est le plus susceptible de résister à un débit grandissant lors de l'évaluation du challenge ?

Pour faire face à des débits croissants successivement imposés par la plateforme d'évaluation, voilà que Spark est directement écarté puisque, pour

celui-ci, c'est à l'utilisateur de préciser le débit en entrée. Une série de benchmarks organisés par Yahoo ! [14] compare les performances des frameworks. L'article étudie Flink, Storm et Spark et montre que les deux premiers répondent linéairement à l'augmentation du débit, contrairement à Spark dont la latence augmente plus rapidement et par paliers, ce qui conforte notre choix d'écarter ce framework. Entre les deux autres, l'article conclut à une plus grande rapidité de Storm à condition de désactiver son système de suivi, sans lequel il ne peut malheureusement plus ni reporter ni gérer les erreurs.

Troisième question : et en pratique ?

La décision finale est prise à partir d'autres qualités de Flink [15] :

— Il propose un traitement de chaque tuple avec une garantie *exactly-once*, alors que Storm se contente d'une garantie *at-least-once* et permet la redondance.

— Son système de tolérance aux fautes est bon. Il dissémine des *checkpoints* dans le flux et en cas de problème ne re-traite que le petit groupe de données entre deux points désignés ; contrairement à Storm dont un opérateur renvoie un constat à l'opérateur précédent pour tous les enregistrements qu'il a traité.

— Flink a une API de plus haut niveau que Storm. Dans ce dernier, les *spout* (connecteurs à une source de données) et les *bolt* (calculs particuliers) au sein des topologies se font manuellement, tandis qu'il existe de nombreux opérateurs pré-implémentés dans Flink.

— Flink introduit le concept de *streaming windows* [12], bien pratique pour notre sujet où les calculs s'effectuent sur des fenêtres glissantes.

— Tout comme pour Storm, tout programme Flink peut s'implémenter en Java, un avantage significatif pour le passage sur la plateforme Hobbit dont l'API et les exemples sont en Java.

— La compatibilité de Flink avec RabbitMQ est native.

Le principal point faible de Flink est finalement son léger manque de maturité. Moins de documentation et de bibliothèques s'avèrent disponibles. Un argument qui a aussi servi à écarter d'autres jeunes frameworks aux performances apparemment prometteuses comme Apache Samza et Apex.

2.1.2 Présentation d'Apache Flink

Flink est un système de traitement distribué de larges volumes de données qui se veut rapide et facile d'utilisation. Il supporte aussi bien le mode flux que le traitement par lots et peut utiliser les mêmes algorithmes sur les deux modes de diffusion. À l'origine, il a été créé en 2010 à partir du projet *Stratosphere* de l'université technique de Berlin, avant d'être rendu *open source* en 2014 en tant que projet Apache développé en Scala et en Java. Apache Flink est compatible avec une architecture YARN, et peut être utilisé aussi bien à distance que localement ou embarqué dans un cloud.

Des noms célèbres ont été séduits par ce framework et fournissent des exemples de cas d'utilisation. King [3] permet l'analyse de données de jeux en temps réel par ses data scientists via une plateforme interne alimentée grâce à Flink. Zalando [4] utilise Flink pour effectuer des synchronisations massives de données du site vers leur entrepôt de données.

Le moteur d'exécution de Flink possède un certain nombre d'atouts assurant son bon fonctionnement et sa robustesse.

La vitesse de traitement des données de notre solution est essentielle pour l'évaluation, et Flink semble faire de cette caractéristique une de ses priorités. Afin d'améliorer la rapidité des opérations, il dispose d'un système d'itération native. Il utilise, d'une part, les itérations simples qui permettent d'assurer le traitement des flux de données successifs. Et d'autre part, il peut avoir recours aux delta-itérations exécutées dans le cadre de l'amélioration de performances des itérations simples, lorsque l'ensemble des traitements est plus dense et inclut des mises-à-jour à prendre en compte au cours de l'avancée des opérations.

En plus d'être rapide, le moteur Flink se veut sûr. C'est à nouveau une caractéristique primordiale d'un bonne solution au DEBS Grand Challenge,

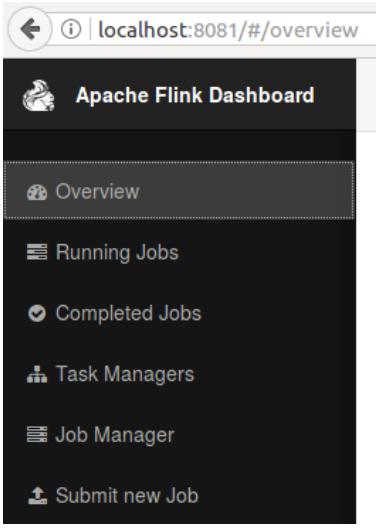
3. King Digital Entertainment, entreprise britannique de jeu vidéo connue notamment pour son jeu *Candy Crush Saga*.

4. Zalando est une entreprise de commerce en ligne allemande, spécialisée dans la vente de chaussures et de vêtements.

elle doit être capable de supporter un débit important même en cas d'alea. Il garantit que, même dans le cas d'une erreur chaque tuple n'est représenté qu'une seule fois au sein du statut de l'opération et sera traité en conséquence. Chaque opération sauvegarde son état régulièrement suivant des *checkpoints*. En cas d'échec, pas besoin de reproduire tous les traitements depuis le début : il suffit de consulter le dernier état sauvegardé (*checkpoint*) par l'opération en échec afin de déterminer les étapes restantes à traiter. Ce mode de gestion des pannes classe Flink en tant que technologie assurant la garantie *exactly once* sur le traitement des messages.

En supplément de ses qualités techniques, ce framework s'attache à conserver une relative facilité d'utilisation. Pour démarrer avec Flink sur votre IDE Java favori il suffit d'ajouter les librairies [5] désignées en tant que dépendances dans le *pom.xml*, ou directement dans le *Build Path*. Dès lors il sera possible d'utiliser, à l'intérieur de classes Java, des fonctions de l'API Flink.

Les jobs Flink peuvent, pour la plupart, être lancés depuis l'IDE Java et les sorties observées directement dans la console. Sinon, il est tout à fait possible de lancer un jar préalablement construit à cet effet depuis un terminal, et d'obtenir les résultats de la log dans un autre terminal.
Pour le suivi des opérations, un tableau de bord interactif est disponible depuis une interface web sur le port 8081 de la machine qui héberge le master. Les jobs en cours et les ressources allouées y sont, entre autres, mis à jour en temps réel. Ce Dashboard est une aide précieuse tant pour les mesures de performance que pour la visualisation de la parallélisation de notre système.



5. Disponibles sur `https://flink.apache.org/downloads.html` pour la version 1.2.0 d'Apache Flink.

2.2 Étapes du développement

Le processus de développement de la solution proposée est le résultat de nombreux compromis avec toujours la vocation d'être flexibles sur les composants implémentés. Le développement a donc été réalisé en cinq étapes distinctes avec différents enjeux.

2.2.1 Implémentation des algorithmes de détection d'anomalies

Les algorithmes correspondent aux trois étapes proposées par le challenge, à savoir : une méthode de clustering (K-means), l'entraînement d'un modèle statistique (chaîne de Markov) et un filtre utilisant les résultats de ces deux étapes.

Les bibliothèques directement compatibles avec Flink contiennent encore peu d'algorithmes et de documentation, et de fait les méthodes qu'elles contiennent ne s'adaptent pas vraiment aux consignes du challenge. Malgré l'existence d'algorithmes en grande partie efficaces dans des librairies d'apprentissage statistique (ou *Machine Learning*) externes, leur import et leur adaptation ne sont pas apparus pertinents. Chaque étape a donc été implémentée spécifiquement pour notre système afin de correspondre immédiatement et exactement au cadre du challenge.

2.2.2 Construction du système dans Flink

L'API de Flink est directement conçue pour traiter un flux de données, à travers un objet `DataStream`. Chaque élément du flux de données reçu depuis une source passe par une succession de traitements. Une des phases du développement a donc consisté en l'ajout successif des différentes étapes de traitement afin de réaliser pour chaque maillon des tests sur l'état du flux, les transformations obtenues, les sorties.

L'objectif était de maîtriser l'utilisation des outils dans l'API Flink. En effet, bien que le framework permette d'excellentes performances, il n'est réellement optimisé qu'à condition d'être correctement utilisé. Parmi les outils dont nous avons besoin, Flink permet nativement de trier un flux de données en définissant une variable-clé, ou encore de réaliser les calculs sur une fenêtre glissante. Les tests réalisés en emboîtant chaque traitement

successivement ont permis d'utiliser proprement ces méthodes, et de mieux préparer la parallélisation du système.

Du point de vue de l'exactitude des résultats, la construction progressive du système a permis de contrôler la bonne transformation des données depuis le flux de triplets RDF. Cela a aussi permis de bien contrôler la validité des résultats sur les étapes statistiques, notamment en faisant varier des paramètres comme le nombre N de transitions calculées pour la chaîne de Markov, ou la taille W de la fenêtre glissante.

2.2.3 Mise à l'épreuve du système en local

Lorsqu'un système est mis en ligne sur la plateforme HOBBIT pour être confronté aux différents *benchmarks* d'évaluation, les retours sur l'exécution sont très opaques et il n'y a pas de possibilité d'accéder aux logs du système. Il est donc utile de créer en local un environnement de test qui soit pratique et aussi proche que possible des conditions d'évaluation de la solution au challenge.

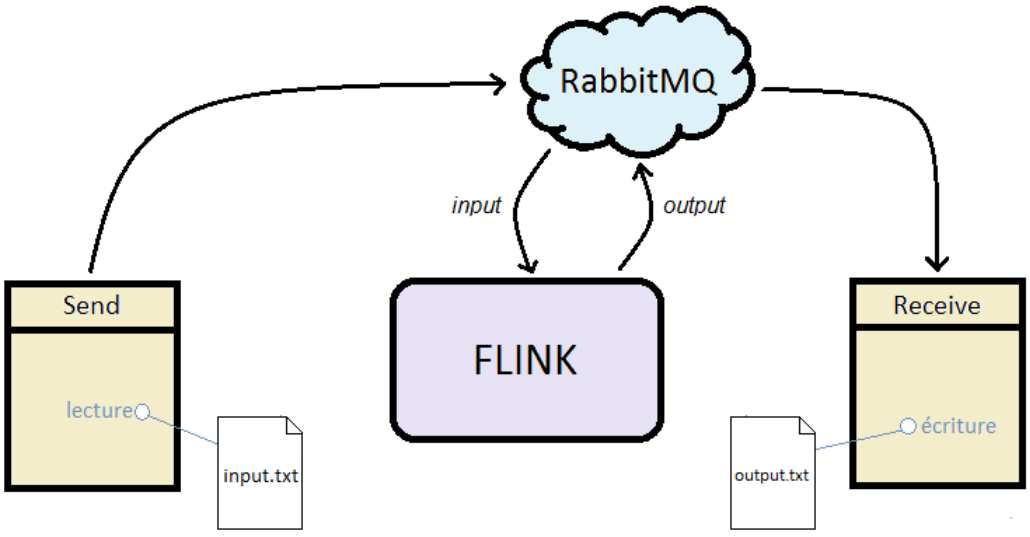


FIGURE 4 – Environnement de test du système de détection d'anomalies

L'environnement de test, représenté par la Figure 4, nécessite dans un pre-

mier temps de lancer le serveur RabbitMQ en local. Le système de détection d'anomalie est une classe Java qui utilise l'API de Flink en se connectant aux files *input* et *output*. Pour générer le flux de données, une classe `Send` lit un fichier texte et envoie séquentiellement son contenu sur la file *input* du serveur RabbitMQ. Inversement, une classe `Receive` écoute la file *output* de RabbitMQ et affiche les messages reçus au fur et à mesure qu'ils arrivent.

Pour imiter la variation du débit d'entrée de la plateforme, la classe `Send` possède un paramètre correspondant au temps d'attente entre l'envoi des lots de données par groupe d'observations. `Send` expédie donc à intervalles réguliers des séries de milliers de lignes contenant des triplets RDF, simulant une fréquence d'observation des machines.

Les premiers échantillons de données ne contiennent qu'une seule machine et ses 55 capteurs. Même en débit maximal (aucun temps d'arrêt), le volume de données envoyé est insuffisant pour mettre Flink à l'épreuve.

Beaucoup de bruit pour rien
Avec une seule machine à injection de matière, nous récupérons à chaque groupe d'observations la valeur de 55 capteurs. Pour ces 55 tuples soumis à la détection d'anomalies, 940 messages sont reçus par RabbitMQ soit 17 fois plus d'entrées que de tuples évalués.

Des fichiers de données plus volumineux ont donc été construits artificiellement pour obtenir jusqu'à 10 machines dans le flux. Quand le débit est fixé à 10 groupes d'observations par minute, ce sont des séries de 9440 messages RabbitMQ qui sont envoyés par la classe `Send` toutes les 6 secondes. Après observation des premières latences, il a fallu optimiser l'envoi du flux de données car la lecture de centaines de milliers de lignes de texte était plus coûteuse que le traitement par Flink lui-même. Pour les tests suivants, la lecture du fichier de données et la transformation des lignes en bytes est faite à l'avance. Ensuite seulement la classe envoie les messages vers RabbitMQ, avec un plus fort débit.

Au moment de la rédaction de ce rapport, les organisateurs n'ont pas fourni de jeu de données avec plus d'une machine. Le système n'a donc pas été testé dans le scénario 2 pour évaluer sa réaction à la variation dynamique du nombre de machines traitées.

Au niveau de la file de sortie, la classe `Receive` permet l'affichage en continu des résultats du traitement au format RDF attendu. Pour observer la sortie de tous les tuples et pas seulement les anomalies, le programme a été testé presque systématiquement avec un seuil [6] d'anomalie supérieur à 1.

2.2.4 Optimisation et parallélisation

L'environnement de test en local entièrement maîtrisé permet de définir un certain nombre de mesures pour comprendre le comportement du système. À cela s'ajoutent les outils de métriques déjà disponibles dans Flink pour décrire l'exécution des traitements. Un effort important a été accordé à la définition de mesures pertinentes permettant de mettre en lumière les opérations les plus coûteuses de nos traitements : où sont les files d'attentes ? Comment les tâches sont-elles parallélisées ? Comment les traitements réagissent à la variation des paramètres externes comme la taille de la fenêtre ou la limite d'itération du K-means ? Le système a donc d'abord été optimisé en local progressivement, en fonction des faiblesses observées ou soupçonnées.

Mais le challenge reste orienté systèmes répartis, les performances sur une seule machine ne sont donc pas suffisantes pour rendre la meilleure solution. D'autres tests ont été menés sur un cluster Flink mis en place sur 4 machines physiques identiques, avec espoir de vérifier la cohérence des résultats et ajuster la parallélisation des traitements. Les machines du cluster sont très fortement inférieures en capacités à celles disponibles sur Hobbit, notamment en mémoire RAM et en connexion. Ces tests n'ont pas abouti à un système distribué fonctionnel.

2.2.5 Déploiement du système

La plateforme Hobbit fournit un cluster de 4 machines pour tester nos solutions. Les nodes sont équipés de processeurs 2×64 bit Intel Xeon E5-2630v3 (8-Cores, 2,4 GHz, Hyperthreading, 20MB Cache), avec 256 GB RAM

6. Rappel : On déclare un événement comme anomalie quand sa probabilité de se produire d'après le modèle de Markov est faible. Toute probabilité étant comprise entre 0 et 1, un seuil d'anomalie plus grand que 1 conduit à déclarer tous les événements comme anomalies.

et 1Gb Ethernet. De plus, l'inscription au challenge donne accès à deux espaces liés au projet HOBBIT :

— **Gitlab** : Espace destiné à l'upload de l'image Docker [3] contenant notre système et d'un fichier turtle de métadonnées (*system.ttl*) le décrivant.

— **Plateforme d'évaluation (Hobbit GUI)** : Permet d'accéder aux benchmarks disponibles et de lancer leur exécution sur notre système. Une fois le benchmark terminé, plusieurs indicateurs clés sont mis à disposition.

La disponibilité de la plateforme a été limitée jusqu'en Mars, et le premier exemple d'adapteur fonctionnel a été proposé le 14 Mars suite à des demandes de participants. Le développement d'un adapteur spécifique pour réaliser les tests sur HOBBIT est assez contraignant. La principale difficulté provient du débuggage, faute d'accès aux logs. Malheureusement, la version d'HOBBIT utilisable en local comprend encore beaucoup de failles (9 *issues* ouvertes sur Github au moment de la rédaction du rapport) qui rendent son usage tout aussi risqué. L'adapteur nécessaire à la soumission de notre solution sur la plateforme n'est donc pas encore implémenté.

3 Architecture du système

La solution développée pour le DEBS Grand Challenge 2017 est composée de plusieurs briques dont l'ensemble forme un système de détection d'anomalies industrielles.

3.1 Architecture générale

Le système de détection d'anomalies est bâti autour d'un programme Flink, mais d'autres composants s'y rattachent pour permettre son intégration et son évaluation sur la plateforme HOBBIT. Ils sont représentés sur la Figure 5, les parties qui suivent sont consacrées à l'explication de chaque composant. Ce sont principalement des outils liés au déploiement du système distribué et des moyens de communication avec l'environnement de test.

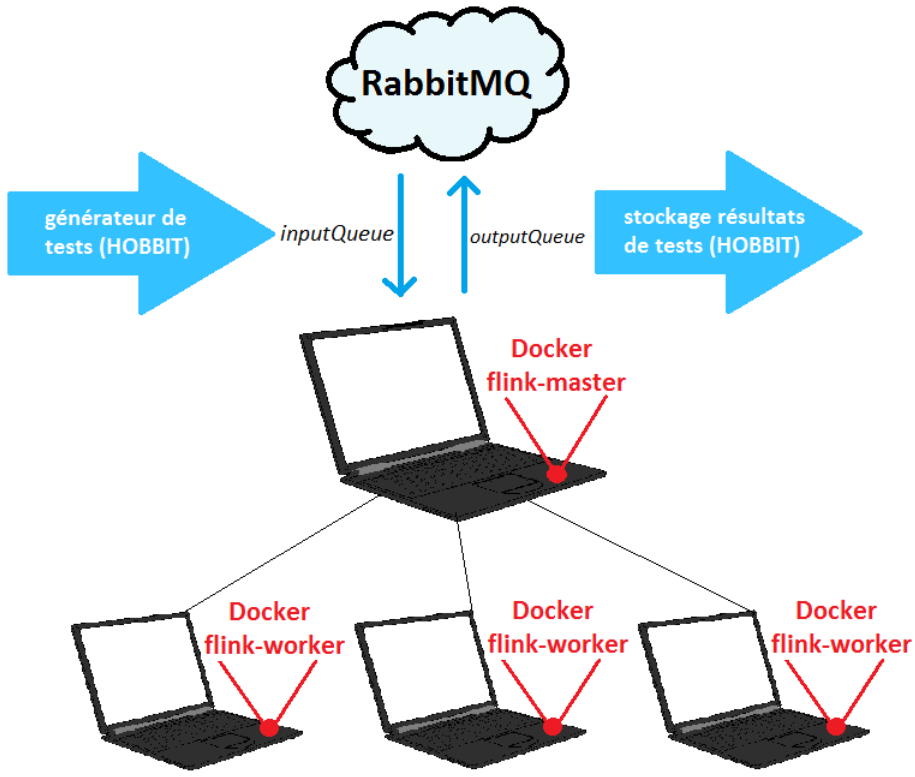


FIGURE 5 – Système complet et interfaces avec la plateforme d'évaluation

3.1.1 Déploiement et distribution sur plusieurs machines

Le déploiement du système s'effectue au moyen de conteneurs Docker [3]. Il s'agit de conteneurs légers qui s'apparentent à des machines virtuelles, permettant d'isoler l'exécution d'un processus dans un environnement adapté. Plutôt que d'embarquer un système d'exploitation, un conteneur Docker est lié à sa machine hôte. Enfin un conteneur est l'instance d'une image Docker, définie dans un script qui indique les commandes pour l'initialiser (paquets à installer, variables d'environnement, ports pour communiquer).

Le système est déployé à partir de deux images Docker différentes selon le pattern maître-esclave. Selon la sémantique de Flink, l'image *flink-master* héberge le *JobManager* et l'image *flink-worker* est instanciée par les *TaskManagers*[7]. Les images sont construites à partir de conteneurs sur lesquels Flink est déjà disponible. Au lancement sur une machine d'un conteneur *flink-worker*, un *TaskManager* s'inscrit auprès du *JobManager* Flink (potentiellement sur une autre machine) pour pouvoir participer aux traitements.

L'intérêt du framework Flink est de gérer seul la plupart des problématiques liées à la distribution du système : tolérance aux fautes, répartition des tâches, cohérence des résultats. Mais il est possible de bien paramétrer la parallélisation des traitements.

Parallélisation par les données

Le système se prête fortement à une division du flux de données dans des *threads* séparés, les capteurs des machines étant indépendants les uns des autres. Le couple (machine, capteur) agit donc comme une clé qui permet de répartir les traitements.

Les tâches pour un tuple unique ne sont pas parallélisées. La taille de la fenêtre glissante étant de l'ordre de la dizaine d'observations, le système effectue surtout un très grand nombre de traitements courts. Les étapes de clustering et d'apprentissage du modèle de Markov pour un tuple nécessitent peu de ressources.

7. La première version de ces images Docker a été reprise à un tutoriel du blog de www.big-data-europe.eu [2]

Allocation des ressources

Le *JobManager* dispose d'un certain nombre de *TaskManagers*, chacun pouvant accomplir un certain nombre de tâches simultanées en fonction du nombre de cœurs disponibles sur le processeur. La première implémentation de notre système repose sur l'optimisation par défaut de Flink pour allouer des cœurs à chaque tâche.

Les tests sur un cluster fonctionnel n'ayant pas abouti à l'heure de la rédaction du rapport, ils demeurent une priorité pour améliorer le système. Une fois le système capable de répartir les traitements correctement sur des machines séparées, il sera nécessaire de mesurer sa latence et de paramétrer l'allocation des *Taskmanagers* en fonction des traitements les plus stratégiques.

3.1.2 Interagir avec la plateforme HOBBIT

Le banc de test fournit par Hobbit est composé d'une multitude de conteneurs Docker communiquant par des files RabbitMQ. Lorsqu'un utilisateur soumet un système à une évaluation, les étapes essentielles sont les suivantes :

— Un conteneur génère des jeux de données qu'il envoie à un générateur de tests.

— Le générateur de tests prépare des tests pour lesquels il génère aussi les réponses attendues. Les réponses sont stockées dans un conteneur dédié. Une fois qu'il est prêt, le générateur de tests expédie le flux de données vers le système testé.

— Le système testé reçoit d'abord un message de top départ, puis les données, et un message d'arrêt une fois le flux terminé. Les résultats des traitements sont expédiés en flux vers le conteneur qui stocke les résultats de test.

— Les résultats sont comparés avec la référence et l'utilisateur reçoit un compte-rendu.

L'initialisation, le déroulement et la fermeture des communications entre le système de détection d'anomalies et le reste de Hobbit sont implémentées dans un adaptateur en Java. Un exemple de celui-ci est fourni par les organisateurs. Il permet surtout de recevoir automatiquement les données depuis les bonnes files de RabbitMQ. En revanche les métadonnées et les paramètres

utilisés pour l'évaluation finale seront donnés peu avant par mail aux participants.

3.2 Architecture du programme Flink

Cette partie est consacrée à l'architecture du programme Flink de détection d'anomalies avec, dans un premier temps, un aperçu global de sa structure. Ensuite chaque étape de la méthode de détection est détaillée à travers les différents choix d'implémentation qui ont été faits.

L'architecture générale de notre système s'articule autour de trois grandes parties : le clustering avec l'algorithme K-means, l'entraînement du modèle de Markov et le filtrage des anomalies. En dehors de ces trois blocs, la structure des données entrantes est transformée pour les adapter au système et inversement pour obtenir le format demandé en sortie. Enfin, le tout communique avec le monde extérieur au moyen de files RabbitMQ en entrée et en sortie de notre système.

Tout se passe dans l'environnement de Flink dédié au *streaming*. Le flux de données qui traverse le système est un objet `DataStream`. Il est initialement créé à la lecture de la sortie de RabbitMQ. Une série de transformations est ensuite appliquée sur les données jusqu'à obtenir le format voulu. Ces transformations modifient les données du `DataStream` en rajoutant, modifiant ou en supprimant des éléments dans le flux mais elles conservent toujours ce contexte de données en flux qui ne permet pas une manipulation classique des données.

Les fonctions appliquées sur le `Datastream` implémentent principalement deux types de transformations (interfaces) :

— `FlatMap` qui pour chaque élément du `DataStream` renvoie un nombre quelconque d'éléments ;

— `Filter` qui permet de filtrer le flux en ne laissant passer que les éléments voulus.

`KeyBy` et `countWindow` sont aussi deux fonctions utilisées pour manipuler le `DataStream`. La première sépare le flux dans plusieurs pipelines, permettant le traitement parallèle des données dans plusieurs chaînes indépendantes de détection d'anomalies. `CountWindow` permet de créer une fenêtre glissante

de taille W dans le `DataStream` sur laquelle s'applique la méthode de K-means.

Flink Plan Visualizer permet de visualiser précisément l'architecture de notre système.

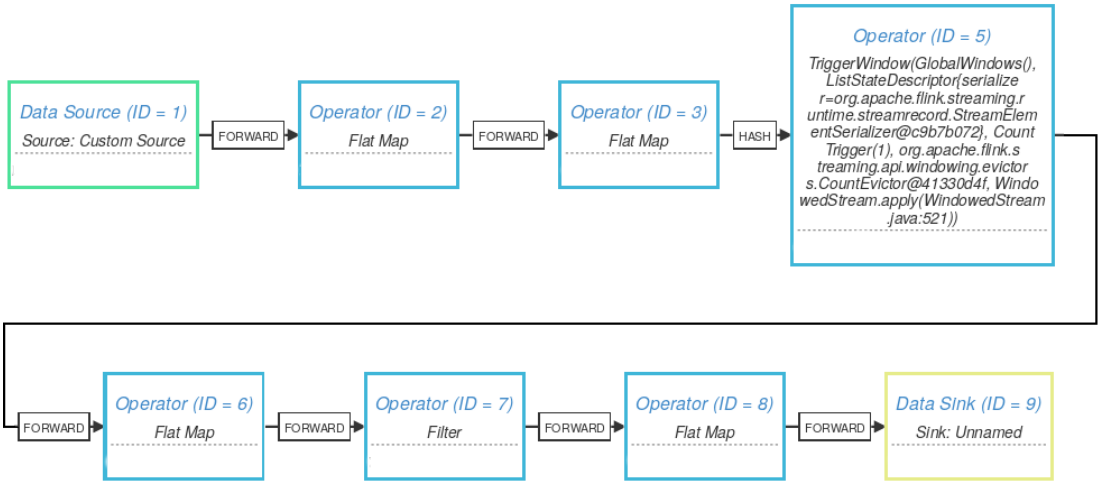


FIGURE 6 – Achitecture Flink

Les données sont lues depuis une file d'attente RabbitMQ (ID = 1). Ensuite une première `FlatMap` (ID = 2) sépare les lignes reçues et forme des tuples de trois chaînes de caractères correspondant aux trois parties des triplets RDF (sujet, prédicat, objet).

Après cette opération une deuxième `FlatMap` (ID = 3) extrait l'information de ces triplets pour former des tuples ne contenant que les données nécessaires. Ces tuples sont séparés par machine et par capteur à l'aide d'un `keyBy` puis envoyés dans les fonctions de clustering (ID = 5) puis Markov (ID = 6). Enfin une fonction de type `Filter` (ID = 7) réalise la détection d'anomalies en ne laissant passer que les tuples déclarés anormaux. Pour terminer une `FlatMap` (ID = 8) formate ces tuples avec la syntaxe de sortie demandée et les envoie dans une file RabbitMQ (ID = 9).

Dans les parties qui suivent, chaque bloc du système est décrit plus précisément.

3.2.1 Traitement et usage des métadonnées

La connaissance des métadonnées par le système est indispensable au déroulement des traitements sur le flux de données reçu.

Que contiennent les métadonnées ?

Les métadonnées contiennent les identifiants ainsi que le type des machines (Molding/Injection) qui seront présentes dans le flux. Elles contiennent aussi les identifiants des capteurs d'intérêt pour chaque type de machine, seule une partie des capteurs étant à considérer pour la détection d'anomalies. Les métadonnées contiennent de plus pour chaque capteur le nombre de clusters à utiliser dans l'algorithme de clustering ainsi que le seuil à partir duquel une transition est considérée anormale pour ce capteur.

Sous quelle forme et comment les métadonnées sont-elles fournies ?

Les métadonnées sont lues directement depuis un fichier texte externe. Pour chaque test, ce fichier est directement fourni par les organisateurs au préalable. Les métadonnées se présentent sous la forme de triplets RDF comme les données des machines. Le mode de transmission n'a été clarifié qu'après le 20 mars, la gestion des métadonnées a donc connu plusieurs versions.

La solution de l'adaptabilité : le fichier de configuration

Pour être moins dépendants du mode d'accès aux métadonnées, le choix a été fait d'intégrer un fichier de configuration. Ce fichier contient les informations du fichier de métadonnées formatées de manière à être facilement interprétable par notre système. Des objets Java `Properties` permettent d'écrire et lire ce ficher de configuration. Dans le programme, la classe `Metadata` gère ce fichier de configuration. Elle classe possède deux méthodes principales :

— `readData()` qui lit un fichier texte de métadonnées et crée le fichier de configuration correspondant pour le système.

— `load()` qui charge les informations des métadonnées dans le système depuis le fichier de configuration.

Ainsi le système fonctionne en duo avec son propre fichier de métadonnées. Si la méthode de réception était amenée à changer au cours du challenge, il suffirait de modifier la fonction `readData()` pour créer le fichier de configuration correctement.

Pour chaque type de machine, les métadonnées sont chargées dans le système dans une `Map` Java. La clef de cette `Map` est un entier représentant l'identifiant des capteurs à surveiller. Les valeurs sont des tuples contenant un entier pour le nombre de centres du cluster de ce capteur et un double pour le seuil de détection d'anomalies.

3.2.2 Traitement du flux de données en entrée

Les données en entrée nous parviennent via RabbitMQ sous la forme de chaînes de caractères contenant soit une ligne, soit un groupe de lignes en Turtle représentant chacunes un triplet RDF. Un exemple d'entrée RDF jusqu'au premier capteur du premier groupe d'observations est disponible Figure 17 de l'Annexe A.

La première tâche consiste à séparer ces lignes lorsque nécessaire, et à les découper en trois selon le schéma (sujet, prédicat, objet). Pour cela la classe `LineSplitter` qui implémente l'interface `FlatMap`[8] a été codée de sorte à opérer les découpes sur les chaînes de caractères en entrée. `LineSplitter` est le premier opérateur Flink dans lequel passent les données. Son implémentation prend en compte la variabilité des lignes en entrées afin de minimiser d'éventuelles erreurs de lecture du fichier qui pourraient se répercuter dans la suite du programme.

L'information utile au sein d'un groupe d'observations est étalée sur plusieurs lignes. La deuxième tâche consiste donc à retenir toutes les informations nécessaires et seulement les informations nécessaires. Les valeurs à conserver sont : le numéro de machine, le numéro de capteur, la valeur correspondante, et son horodatage. Pour les retrouver dans le lot de triplets RDF nous utilisons également le numéro de groupe d'observations et le numéro d'observation. L'opérateur `InputAnalyze` qui implémente l'interface `FlatMap` de Flink, se charge de la sélection et du tri des valeurs avant de les renvoyer

8. Cette interface permet d'appliquer un ensemble de fonctions définies par l'utilisateur sur chaque élément d'un DataStream. Elle peut retourner un nombre arbitraire de sorties pour chaque élément en entrée par l'implémentation de sa fonction `flatMap()`.

dans le flux sous la forme de `Tuple4` de type $\langle n^o machine, n^o capteur, valeur, horodatage \rangle$. Le choix a été fait d'implémenter `InputAnalyze` comme une succession des scénarios différents en fonction du contenu du tuple en entrée. Ainsi, l'opérateur trie rapidement et n'agit que si l'entrée contient une information nécessaire. Remarquons qu'il sort bien moins de tuples de cette fonction que ce qu'il y entre. En effet, d'une part elle rassemble les données de plusieurs lignes en un seul tuple, et d'autre part elle les trie selon que les capteurs aient ou non été signalés comme intéressants dans les métadonnées. À partir de là, le flux est formaté pour les calculs et a un volume moins important.

3.2.3 Calcul du K-means

En amont du K-means

A ce stade du système, le `DataStream` en amont est un flux de `Tuple4` avec comme composantes l'identifiant de machine, l'identifiant de capteur, l'horodatage et la valeur mesurée par le capteur. Comme expliqué précédemment la détection d'anomalies se fait de manière indépendante sur chaque capteur de chaque machine. Le flux est donc séparé par machine et par capteur avec la fonction `keyBy(id_machine,id_capteur)`.

Une fenêtre glissante est ensuite obtenue grâce à l'application de la fonction `countWindow(W,1)` sur le `DataStream`. La fenêtre est de taille W et est mise à jour à chaque arrivée d'un tuple. Cette fenêtre crée un environnement fixe dans lequel l'ensemble des éléments de la fenêtre est accessible via un objet de type `Iterable` et sur lequel est appliqué l'algorithme de clustering via la méthode `apply()`.

La méthode des k-moyennes

L'algorithme des k-moyennes (ou K-means) est un algorithme de partitionnement de données. Dans cet algorithme l'objectif est de minimiser la distance d'un point au centre du cluster auquel il appartient. Voici les différentes étapes de son fonctionnement :

1. Initialisation de K centres de cluster.
2. Affectation de chaque point de l'ensemble au cluster dont le centre est le plus proche.

3. Mise à jour des centres vers les nouveaux barycentres de cluster.
4. Tant que les centres des clusters se déplacent suffisamment [9], répétition des étapes 2 et 3.

La solution trouvée n'est pas unique et donc pas forcément optimale. Cependant plusieurs contraintes imposées par le challenge rendent cet algorithme déterministe.

— Les centres ne sont pas initialisés aléatoirement mais correspondent aux K premiers points distincts de la fenêtre.

— Lors de l'affectation d'un point à un cluster, s'il se trouve à égale distance de plusieurs clusters, alors il est affecté au cluster donc la valeur du centre est la plus élevée [10].

— Le seuil de distance à partir duquel on estime que les centres n'ont pas bougé et le nombre maximal d'itérations de l'algorithme sont fournis.

Dans notre système l'algorithme K-means s'appuie sur trois classes représentées sur le diagramme de classes de la figure 7.

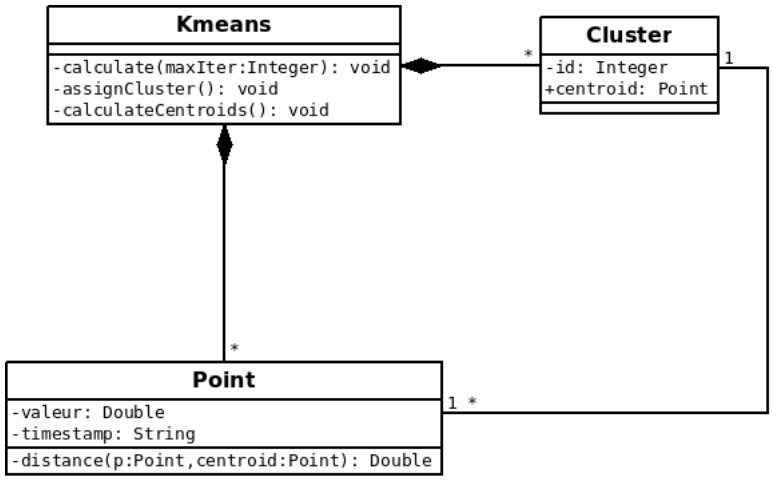


FIGURE 7 – Diagramme de classes de l'algorithme K-means

9. La distance à partir de laquelle on considère que les centres ne se sont pas déplacés nous est fournie dans les paramètres

10. Les points traités sont les valeurs mesurées par les capteurs. On peut ordonner la valeur des centres de clusters car ils se situent dans une espace à une dimension.

Chaque `Kmeans` est composé d'un ensemble de `Point` et de `Cluster`. La méthode générale `calculate()` applique successivement les méthodes `assignCluster()` et `calculateCentroids()` pour effectuer les étapes 2 et 3 de l'algorithme. Deux raisons peuvent nous amener à stopper l'algorithme :

— L'algorithme à convergé, c'est à dire qu'entre deux itérations successives les centre ne se sont pas (ou très peu) déplacés.

— L'algorithme n'a pas convergé mais le nombre maximal d'itérations M a été atteint.

Remarque : un `Point` possède un attribut nommé `timestamp`. Bien qu'il soit facile dans une fenêtre de récupérer l'identifiant du capteur et de la machine (identique pour tous les tuples), tous les horodatages sont différents. Pour conserver l'information lors du K-means, il nécessaire que chaque `Point` la maintienne.

En aval du K-means

Chaque élément du flux en sortie du K-means est une liste de tuples de taille W, en fonctionnement continu. En effet l'opérateur `Kmeans` renvoie une liste contenant tous les tuples de la fenêtre en leur associant leur numéro de cluster. Il est indispensable pour la suite et notamment pour l'entraînement du modèle de Markov de connaître l'identifiant de cluster de tous les tuples pour une fenêtre, car les clusters sont entièrement recalculés à chaque nouvelle observation traitée.

3.2.4 Calcul du modèle de Markov et filtrage des anomalies

L'objectif dans cette étape est d'entraîner le modèle statistique qui sert de référence pour le comportement d'un capteur, en utilisant les résultats du clustering. C'est une fois le modèle entraîné qu'une probabilité de réalisation peut être attribuée à l'événement que le système analyse.

Processus de Markov à temps discret

Un processus de Markov est un processus stochastique possédant la propriété de Markov : toute l'information utile pour la prédiction du futur est contenue dans l'état présent. Une chaîne de Markov est un processus de Markov à temps discret, qui évolue donc d'état en état successivement. D'après

la propriété de Markov, la probabilité de transition d'un état présent vers le suivant est indépendante du chemin emprunté pour atteindre l'état présent.

Grâce à cette propriété, on peut définir la loi de la chaîne de Markov à partir de sa loi initiale (loi du premier état) et de sa matrice de transition (ou noyau de transition). Dans cette matrice, le coefficient p_{ij} est la probabilité d'aller vers l'état j sachant qu'on se trouve dans l'état i. Quand le noyau de transition est identique dans le temps, il représente la loi stationnaire du processus.

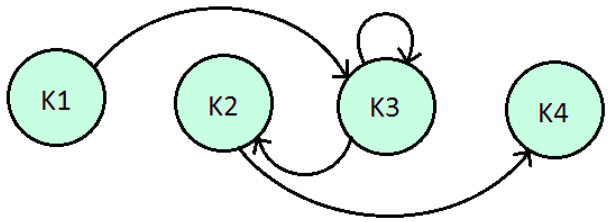


FIGURE 8 – Chaîne de Markov à 4 états

Pour un tuple arrivé dans le système (valeur d'un capteur d'une machine au temps t), l'espace des états de la chaîne de Markov est défini par l'ensemble des K centres de clusters obtenus par le K-Means sur les W observations précédentes. Sur la figure 8, le nombre de cluster K est égal à 4. Le noyau de transition est donc de taille $K * K$, soit en général $13^2 = 169$ probabilités de transition à considérer (d'après la valeur du fichier exemple de données).

La matrice de transition permet aussi de calculer la probabilité d'un chemin de taille quelconque. Soit X une chaîne de Markov, X_t l'état de la chaîne au temps t et $E = (e_i)_{i \in \{1,\dots,K\}}$ son espace d'états. Alors :

$$P(X_0 = e_i, X_1 = e_j, X_2 = e_k, X_3 = e_l) = p_{ij} \times p_{jk} \times p_{kl}$$

Ce résultat s'appuie sur la propriété de Markov, qui pose l'indépendance entre les transitions successives. La probabilité obtenue ici est la probabilité de réalisation d'un chemin à 3 transitions. Pour le Grand Challenge, le système calcule la probabilité de réalisation des chemins de 1 à N transitions, N étant donné en paramètre de la requête par les organisateurs. D'après les notations ci-dessus, les probabilités calculées pour N = 3 sont :

(i) p_{kl}

(ii) $p_{jk} \times p_{kl}$

(iii) $p_{ij} \times p_{jk} \times p_{kl}$

Chaque état entraîne donc le calcul de N probabilités de réalisation.

Implémentation de l'apprentissage d'une chaîne de Markov

L'entraînement du modèle de Markov pour un tuple qui traverse le système est réalisé dans un opérateur `FlatMap`, un tuple entré conduisant à N tuples sortis. En entrée, ce traitement s'applique à un flux de listes de tuples. Une liste de tuples correspond à un tuple du système accompagné des $W - 1$ éléments le précédant, l'ensemble étant la fenêtre utilisée pour le calcul du clustering.

Pour entraîner le modèle on dispose de W observations du processus de Markov. La chaîne de Markov est supposée irréductible, c'est-à-dire que tout état est a priori accessible à partir de n'importe quel autre état. D'après la loi forte des grands nombres, il est possible d'approcher la loi stationnaire d'une chaîne de Markov irréductible et à espace d'états finis par sa distribution empirique. L'idée principale de l'algorithme d'apprentissage est donc d'utiliser la distribution empirique sur les W observations.

En pratique, la liste des observations est parcourue dans l'ordre chronologique d'arrivée et le compte des transitions entre états est maintenu. Pour éviter de remplir une matrice de transition de taille $K * K$, les occurrences sont incrémentées dans une `HashMap` contenant uniquement les transitions effectivement réalisées. On conserve aussi dans une liste de taille K le nombre de passages par chaque état (sauf pour la dernière observation, qui ne donne pas lieu à une transition). Pour deux états i et j, l'élément p_{ij} du noyau stationnaire est estimé par :

$$\widehat{p_{ij}} = \frac{\text{Nombre de transitions de i vers j}}{\text{Nombre de passages par i}}$$

Du point de vue de notre solution au challenge, le modèle de Markov est considéré comme entraîné dès que les transitions ont été comptées. Les N derniers $\widehat{p_{ij}}$ sont calculés directement (et uniquement ceux-là). Les produits

sont calculés dans l'ordre conforme à l'exemple page 34, et chacun donne lieu à la sortie d'un tuple.

La dernière étape de la détection d'anomalie est alors un `Filter` qui ne renvoie que les tuples pour lesquels le modèle de Markov a indiqué une probabilité inférieure au seuil d'anomalie [11]. On remarque donc qu'un même tuple peut être déclaré jusqu'à N fois comme anomalie au même instant. Si la probabilité du chemin à k transitions est sous le seuil de détection, alors celle du chemin à $k+1$ transitions l'est aussi.

3.2.5 Mise en forme des sorties

Le dernier opérateur appliqué au flux est `SortieTransfo`. Il implémente à nouveau l'interface `FlatMap`. Pour chacun des `Tuple5` de la forme $< n^o machine, n^o capteur, valeur, horodatage, proba\ de\ transition >$, il retourne une chaîne de caractère rassemblant, sur plusieurs lignes, l'ensemble des tuples RDF nécessaires à la caractérisation de l'anomalie. Un exemple de sortie pour une anomalie est disponible Figure 18 de l'Annexe A.

Malgré la production de groupements de sept lignes pour chaque anomalie, et puisque les anomalies devraient rester épisodiques, le volume en sortie est moindre comparé au volume en entrée.

11. Pour rappel, le seuil d'anomalie est propre à chaque capteur des machines et parvient au système grâce au fichier de métadonnées.

4 Performances du système

4.1 Méthode(s) de mesure des performances

Mise à part la validité mathématique de la solution, les organisateurs du projet DEBS s'intéressent particulièrement aux performances informatiques du système. Comme le montre la formule d'évaluation, le débit et la latence sont les deux mesures essentielles pour l'évaluation. L'idée est de pouvoir supporter un débit du flux de données maximal, tout en gardant une latence minimale au traitement. Pour ce faire, il est nécessaire de récupérer différentes mesures de performances à travers le système pour espérer identifier un problème et le corriger.

4.1.1 Groupes de mesures et opérateurs

Le traitement des données par Flink est effectué par la suite d'opérateurs : `LineSplitter`, `InputAnalyse`, `Kmeans`, `Markov` et `Filter`. La première étape est d'identifier parmi eux lequel affiche des performances d'exécution à améliorer. Pour ce faire, Flink met nativement à disposition un système de mesures. Le principe est le suivant : chaque fonction utilisateur qui hérite de la classe abstraite `AbstractRichFunction`, y compris indirectement en héritant de `RichMapFunction`, `RichFilterFunction` ou `RichWindowFunction`, dispose de son propre groupe de métriques. Il a donc été nécessaire de changer le type des opérateurs concernés.

Chaque groupe de métriques met à disposition plusieurs mesures par défaut, comme par exemple `numRecordsIn` qui mesure le nombre d'éléments qui rentrent dans la fonction concernée. De surcroît, Flink met à disposition des outils pour créer ses propre mesures, comme l'objet `Counter` qui permet notamment de compter quelque chose, grâce à ses méthodes d'incrémentation ou de décrémentation. Pour personnaliser un groupe de mesures, il suffit de modifier la configuration de la classe héritant d'`AbstractRichFunction`. En implémentant les compteurs voulus dans la méthode `open()` de cette classe, il sont appelés à l'exécution de la même manière que les méthodes de l'opérateur.

Des compteurs ont été implémentés de manière à calculer le temps total [12] d'exécution de chaque opérateur du système. Pour ce faire, le compteur est décrémenté de la valeur du temps actuel au commencement de la fonction, puis incrémenté du nouveau temps actuel à la fin de la fonction.

Voici l'allure d'une classe type pour un opérateur dans lequel est utilisé un groupe de mesures :

```
public static final class MyExempleFunction
extends RichMapFunction<TypeEntrée, TypeSortie>> {

     private Counter counterExemple;

     @Override
     public void open(Configuration config) {
          this.counterExemple = getRuntimeContext()
                                .getMetricGroup()
                                .counter("Exemple de Counter");
     }
     @Override
     public TypeSortie map(TypeEntrée input) {
          this.counter.dec(System.currentTimeMillis());
          // PROGRAM //
          this.counter.inc(System.currentTimeMillis());
          return TypeSortie output;
     }
}
```

Toutes les mesures effectuées sont consultables en temps réel sur le *dashboard* de Flink.

4.1.2 Outil de mesure des files d'attente

La deuxième étape pour comprendre les performances du système est d'étudier le comportement de Flink à l'extérieur des opérateurs. De cette manière, des phénomènes de file d'attentes peuvent être mis en évidence. Ces files d'attentes, additionnées au temps d'exécution des opérateurs en

12. Temps passé en cumulé pour tous les tuples du ficher exemple de données en entrée.

eux-mêmes, permettent de comprendre précisément les résultats de latence.

Les groupes de mesures de Flink ne permettent pas d'avoir accès à des mesures extérieures aux opérateurs. Il a été nécessaire de créer son propre outil de mesure. La solution envisagée se propose d'utiliser la fonction Java `System.currentTimeMillis()` et de l'associer aux éléments du système. Cette méthode de mesure des performances du système permet cette fois de connaître individuellement le comportement de chaque élément du flux de données.

Le plugin log4j d'apache permet d'afficher dans le journal d'information les éléments du flux de données. De cette manière, il est possible de récupérer le temps auquel chacun d'eux passe certains *checkpoints*. Pour faire apparaître les mesures dans la log, un objet de type `Logging` est instancié puis le code suivant se charge de placer un *checkpoint* aux endroits voulus du système :

```
log.info(Element_Du_Stream + "," + System.currentTimeMillis());
```

Ces derniers sont positionnés avant chaque fin d'opérateur et après chaque début d'opérateur. De cette façon ils couvrent l'intervalle de temps à l'extérieur des opérateurs. Cet intervalle correspond à celui représenté entre les barres rouges sur la figure 9 avec l'exemple des opérateurs `Kmeans` et `Markov`.

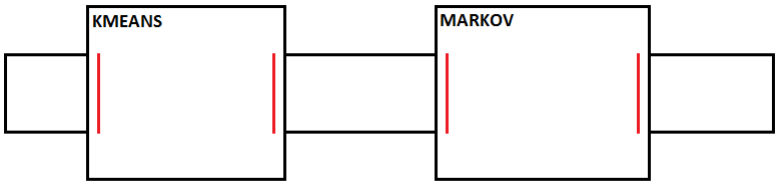


FIGURE 9 – Placement des logs destinées aux mesures

Les données issues du journal d'information peuvent être exportées dans R pour y être exploitées.

4.2 Performances après la première implémentation en local

Les différentes mesures qui sont présentées dans cette partie ont été réalisées en local sur un ordinateur possédant un processeur i7 et 8GB RAM. Ce processeur est aussi composé de 8 cœurs entre lesquels Flink peut paralléliser les tâches du système.

Le jeu de données qui a été mis à l'épreuve correspond à un fichier RDF contenant l'information de 10 machines industrielles à 120 capteurs chacune, et ce pour 64 horodatages différents. Seulement 55 de ces 120 capteurs sont retenus comme pertinents par le challenge. Sur tout le jeu de données, il y a un total de 604 160 lignes d'information qui sont envoyées dans le système Flink. Seuls 34 560 éléments ressortent d'`InputAnalyze` et traversent les opérateurs `Kmeans` et `Markov`. Enfin, seules les anomalies atteignent l'opérateur `SortieTransfo`.

4.2.1 Temps d'exécution des opérateurs

Les résultats des groupes de mesures sont calculés automatiquement par Flink. Ils apparaissent sur le Dashboard où les graphiques sont construits en temps réel. Un exemple de graphique construit en temps réel dans le Dashboard est présenté Figure 19 en Annexe B. Ces graphiques ont permis de rassembler les mesures présentes dans le tableau suivant.

Fonction	**Temps cumulé (ms)**	**Nombre d'input**	**Temps d'exécution moyen (μs)**
LineSplitter	2 070	604 160	3.426
InputAnalyze	884	604 160	1.463
Kmeans	706	34 560	20.428
Markov	104	34 560	3.009
SortieTransfo	83	16 203	5.123

FIGURE 10 – Synthèse des mesures sur les opérateurs

`LineSplitter` est l'opérateur qui prend le plus de temps d'exécution dans le système. Ce constat s'explique par le fait qu'il reçoit environ 17 fois plus d'entrées que les autres. Son temps d'exécution moyen par tuple n'est pas le plus important, mais reste deux fois supérieur à celui d'`InputAnalyse`.

La lenteur relative des opérations effectuées par `LineSplitter` est une faiblesse considérant son importance dans le système, il sera prioritaire durant la phase d'optimisation.

Le deuxième constat porte sur `Kmeans` qui présente un temps d'exécution moyen environ 4 fois supérieur aux autres opérateurs. Cela s'explique par la nature de l'algorithme K-means : itératif, il dépend du nombre d'éléments à agréger et de la rapidité de convergence. De plus, si la taille W de la fenêtre s'élargit, elle présente aussi le risque de multiplier la complexité de l'algorithme.

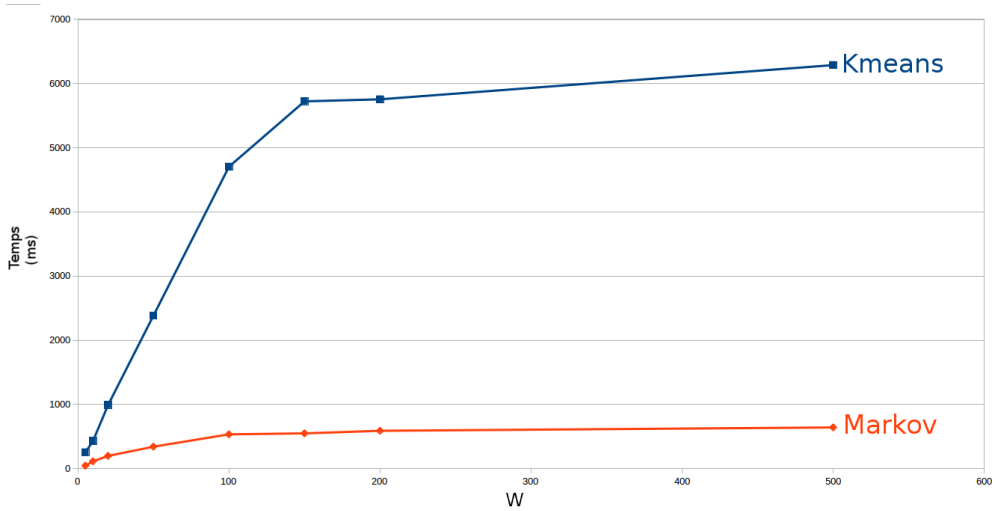


FIGURE 11 – Évolution des temps d'exécution des algorithmes K-means et Markov selon W

En suivant la courbe bleue du graphique, la complexité algorithmique du K-means semble être linéairement proportionnelle à W jusqu'à un certain seuil, puis stagne. Cela est dû au fait que le jeu de données sur lequel ont été effectué ces mesures a été rallongé jusqu'à 190 groupes d'observations pour pouvoir observer de grandes valeurs de W, et qu'à partir de cette valeur le fichier n'a plus suffisamment de données pour mesurer une quelconque différence.

À la suite de ces résultats, l'effort d'optimisation semble devoir se concentrer sur `LineSplitter` et `Kmeans` pour permettre au système de supporter plus longtemps la croissance du débit de données à l'évaluation du challenge.

4.2.2 Files d'attente

Après l'analyse de la part de la latence causée par le traitements des données à l'intérieur des opérateurs, il convient de se pencher sur la part de la latence due au phénomène de file d'attente.

Le journal d'information fournit le temps de passage de chaque élément du flux de données. Quelque soit le débit initial, il n'existe aucune file d'attente entre les opérateurs, à l'exception de l'entrée de `LineSplitter`. Cela signifie que tout les éléments qui sortent de `LineSplitter`, `InputAnalyze`, `Kmeans`, `Markov` ou `Filter` se retrouvent immédiatement dans la fonction suivante, du moins à la milliseconde près.

L'analyse de la latence entre les opérateurs met, à nouveau, en cause `LineSplitter`. Le ralentissement occasionné par ce traitement est illustré par la Figure 12. Le débit auquel ces mesures ont été produites est maximal. Il correspond à un temps d'attente de 0s entre l'envoi de deux groupes d'observations. Ainsi, un groupe d'observations complet est envoyé toutes les 130ms environ.

Plusieurs phénomènes sont mis en évidence sur la Figure 12.

Premièrement, la latence des premiers éléments, entre 0 et 10^5 entrées, est plus importante que celle des autres. Appelé phénomène *bootstrap*, le pic correspond au temps pris par les différentes initialisations du système au démarrage comme la mise en place des relations, ou encore l'instanciation des objets, etc.

Deuxièmement, même une fois stabilisée, la courbe conserve une allure en dents de scie. Cela s'explique par l'utilisation du micro-batching par Flink pour réguler les files d'attente. En cas de goulot d'étranglement, de petits paquets de données se remplissent à l'entrée de la fonction avant d'y entrer, et ce de façon cyclique.

Troisièmement, la partie stabilisée de la courbe peut être interprétée comme un palier de la vitesse de traitements des informations par `LineSplitter`. À très fort débit, la file d'attente est constamment pleine, les données s'accumulent.

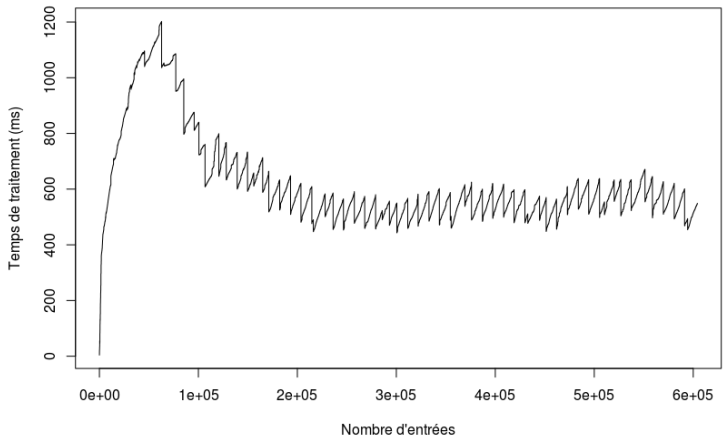


FIGURE 12 – Temps de latence d'un élément entre l'envoi à fort débit dans le système et son entrée dans LineSplitter

Pour aller plus loin, le débit a été diminué pour observer son impact sur ces files d'attente. La Figure 13 représente les mêmes mesures avec un débit plus faible, correspondant à un temps d'attente de 1s entre deux groupes d'observations.

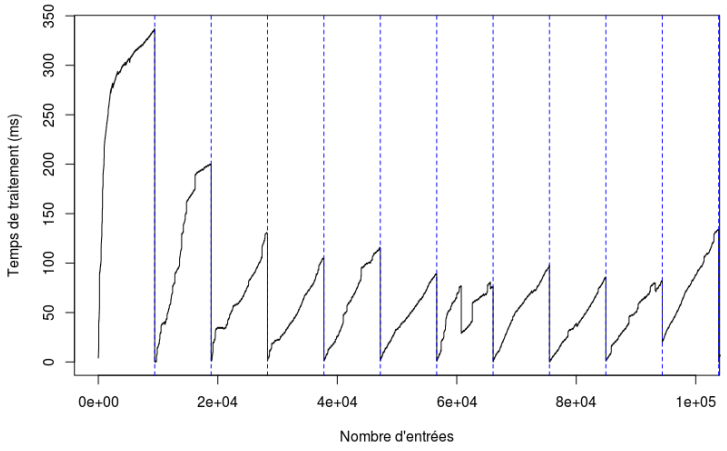


FIGURE 13 – Temps de latence d'un élément entre l'envoi à faible débit dans le système et son entrée dans LineSplitter [14]

14. les lignes verticales en pointillés délimitent les groupes d'observations, et seuls quelques uns ont été représentés.

La différence est flagrante entre la Figure 13 et l'expérience à haut débit de la Figure 12 : les pics successifs correspondent cette fois aux délimitations entre les différents groupes d'observations. À chaque nouvel envoi espacé d'1s, la file d'attente à eu le temps de se vider. En effet, l'entrée des premiers éléments du groupe d'observations a une latence proche de 0ms. Elle augmente progressivement autour de 100ms en situation stationnaire. La latence moyenne pour chaque élément du flux de données est environ 5 fois plus faible que dans la situation précédente. La file d'attente se remplit malgré tout, mais seulement de façon périodique, elle a le temps de se vider avant le prochain envoi.

4.2.3 Latence globale

Il a été montré que le phénomène de file d'attente provoque des délais qui s'additionnent au temps de traitement des données dans les méthodes elles-mêmes. Mais, pour mesurer la robustesse du système vis à vis du challenge, c'est la latence globale du système qui importe.

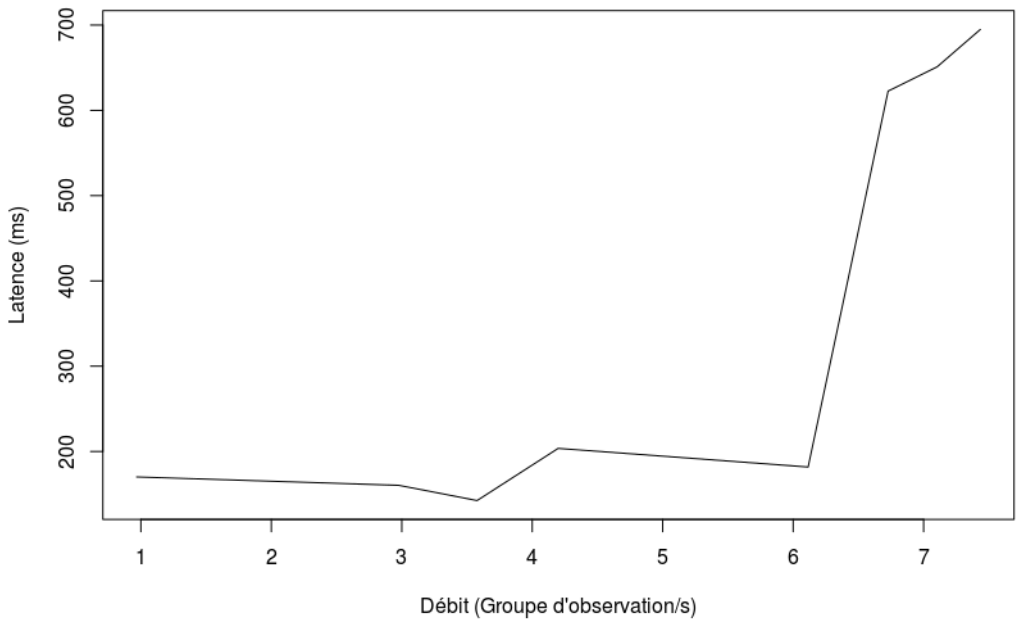


FIGURE 14 – Latence globale moyenne d'un groupe d'observations

Les résultats de latence globale présentés par la Figure 14 ne sont que la visualisation des phénomènes décrits précédemment. En effet, d'une part la faible latence moyenne des groupes d'observations pour des débits faibles est le résultat d'une file d'attente très courte ou vide. D'autre part, l'explosion de la latence moyenne des groupes d'observations pour des débits supérieurs à 7 est le résultat d'une longue file d'attente. La rupture se produit à partir du moment où le débit d'envoi est supérieur au temps de traitement d'un groupe d'observations par le système. Elle semble correspondre à un temps d'attente de 160ms entre deux groupes d'observations, soit environ 6 groupes d'observations par seconde. Cela correspond effectivement à la latence moyenne d'un groupe d'observations à un fort débit.

4.3 Optimisation et perspectives d'amélioration

Il a été observé que les optimisations prioritaires pour la mise à l'échelle du système concernent les opérateurs `LineSplitter` et `Kmeans`. Cependant, il faut garder à l'esprit que toutes les mesures réalisées jusqu'à présent ont été faites en local sur une seule machine physique. Le changement de configuration en faveur d'un cluster de machines qui permet la parallélisation du système est une optimisation en soi.

4.3.1 Optimisation des méthodes de traitement des données

Optimisation du K-means

Le principal problème de `Kmeans` est son important temps d'exécution. La fonction a été implémentée d'une telle manière qui ne permet pas la parallélisation de celle-ci. Or l'algorithme K-means est un algorithme qui se prête bien au principe de la parallélisation grâce au map reduce. Le calcul du centre le plus proche, ainsi que la mise à jour des moyennes des cluster, peuvent facilement se mapper sur plusieurs travailleurs. De cette manière,le K-means peut être optimisé en étant réécrit par des fonctions `map` et `reduce`.

Optimisation du LineSplitter

L'opérateur `LineSplitter` gère beaucoup d'entrées puisqu'il traite l'ensemble les lignes du fichier Turtle pour en ressortir des triplets RDF. Il est

d'autant plus intéressant à optimiser.

Il utilise la méthode `split()` du package `java.lang.String` de façon successive pour séparer des chaînes de caractères en fonction d'une expression régulière. Or, il existe une fonction plus efficace pour séparer une chaîne de caractères : la fonction `split()` du package `StringUtils`. Elle gagne du temps en n'utilisant pas les expressions régulières, mais ne peut être utilisée que pour séparer une chaîne en fonction d'un seul caractère. Les fonctions ont été remplacées à chaque fois que c'était possible, semblant occasionner une légère amélioration.

Pour optimiser le traitement effectué par `LineSplitter`, il a aussi été envisagé de le découper en plusieurs opérateurs n'effectuant chacun qu'une partie des fonctions sur les chaînes de caractères. Un opérateur complexe unique se change alors en une succession d'opérateurs plus simples. A priori ce procédé permet à Flink de mieux allouer les ressources en faveur de ces opérations. Et un élément du flux de données n'attend plus l'exécution de multiples `split()` dans `LineSplitter` pour être traité, mais plus qu'un.

Néanmoins, aucune amélioration n'a pu être observée en local. On peut imaginer que la parallélisation au sein d'un cluster de plusieurs machines physiques permettrait à cette méthode de révéler son potentiel.

4.3.2 Distribution des tâches sur plusieurs machines pour améliorer la vitesse de traitement

À l'heure de ce rapport, le système n'a pas fonctionné de manière satisfaisante de manière distribuée. De nombreux tests de déploiement ont été menés sur nos propres machines, les *workers* (*TaskManagers*) étant soit dans des conteneurs Docker soit directement sur les machines. Le système peut fonctionner avec plusieurs *TaskManagers* dans des conteneurs Docker sur une seule machine. De même le système fonctionne avec une machine comme *JobManager* et une autre comme *TaskManager*, et ce avec les paramètres souhaités de parallélisation.

Cependant des problèmes ont systématiquement émergé lorsque les tâches sont réparties sur des *TaskManagers* sur des machines différentes. Qu'il s'agisse de problèmes de réseau, de paramètres de Flink ou de spécificités de Do-

cker, les nombreuses combinaisons doivent être testées méthodiquement pour aboutir à un système fonctionnel.

Le passage sur la plateforme Hobbit et son cluster à 4 machines de 8 cœurs chacunes est prévu. Mais à ce jour le déploiement sur Hobbit est peu documenté et les organisateurs ont même fini par retirer de la plateforme le test qui permettait de vérifier l'exactitude des résultats. La date butoir du challenge a été décalée au 14 avril 2017 notamment à cause de difficultés techniques et de questions entourant cette plateforme de test.

4.3.3 Exactitude de la solution

Trois jours avant le rendu de ce rapport, les organisateurs du challenge ont mis en ligne les données nécessaires à la vérification de la justesse des résultats. Notre programme traite bien ces nouveaux fichiers de données test, mais ne retourne pas les anomalies attendues par le challenge pour les paramètres indiqués. Néanmoins, après quelques vérifications disponibles en Annexe C, il retourne bien les anomalies que nous attendions.

Le problème n'est pas d'ordre algorithmique, ce n'est pas une mauvaise implémentation du but visé. Il s'agit d'un doute sur l'interprétation de la consigne, au niveau de la méthode mathématique de détection d'anomalies. D'après celle-ci, une anomalie doit être retournée pour toute séquence jusqu'à N transitions ayant une probabilité en deçà du seuil T [15]. Nous avons alors envisagé, tester et calculer de nouvelles interprétations de la consigne, mais sans jamais obtenir un résultat s'approchant de celui du fichier destiné à la vérification.

Le 24 mars 2017 nous avons envoyé une question aux organisateurs afin de préciser leurs critères de détection. Leur réponse nous permettra de déterminer l'origine de la difficulté et de parfaire notre solution.

15. Dans le texte : « Output an alert about a machine, if any sequence of up to N state transitions for that machine is observed that has a probability below T. »

Conclusion

Au terme de l'implémentation de notre solution au DEBS Grand Challenge 2017, nous proposons un système fonctionnel de détection d'anomalies capable de traiter plusieurs dizaines de milliers de triplets RDF par seconde. Le choix d'Apache Flink comme framework de calcul distribué sur des données en flux a tenu ses promesses. Il a permis d'économiser une partie de la complexité technique de l'implémentation, nous laissant apporter plus de soin à la chaîne de détection d'anomalies elle-même et à son optimisation.

La réflexion sur les mesures de performances du système a été l'occasion de mieux analyser le comportement des méthodes implémentées. Le choix et l'obtention de mesures pertinentes furent des défis qui auraient parfois mérité plus d'informations sur les conditions réelles d'évaluation du système. Lors de la rédaction de ce rapport, nous n'avons connaissance ni de l'ordre de grandeur du volume de données (nombre de machines) ni de leur débit d'entrée. Néanmoins les performances actuelles du système nous semblent d'autant plus correctes qu'elles sont réalisées par un seul ordinateur, dont les capacités sont très largement inférieures à celles de ceux fournis pour l'évaluation.

Nous regrettons de ne pas encore pouvoir garantir l'exactitude des anomalies détectées et espérons une clarification prochaine de la méthode des organisateurs. Cependant les résultats du système étant actuellement exactement conformes à notre interprétation de la méthode demandée, nous sommes confiants pour obtenir des résultats valides dès que la consigne sera clarifiée.

La distribution du système s'est avérée plus complexe que nous ne l'imaginions. Malgré de nombreux tests, nous n'avons fait qu'approcher un fonctionnement satisfaisant. Le déploiement sur plusieurs machines physiques ou virtuelles est donc l'objectif majeur pour continuer le challenge. Nous espérons en tirer une meilleure marge de manœuvre pour l'optimisation de nos temps de traitement.

À l'heure de l'écriture de ce rapport, il reste 19 jours avant la soumission définitive des solutions candidates au Grand Challenge. À l'image de la dernière semaine, les échanges avec les organisateurs sont fréquents et beaucoup d'interrogations persistent sur les attentes du challenge. En réalité, les organisateurs ont déclarés que celles-ci vont être ajustées progressivement

en prenant en compte les participations des candidats. Notre système de détection d'anomalies est donc encore vraiment susceptible d'évoluer et nous prenons part activement à ces échanges pour proposer la meilleure candidature possible au Grand Challenge.

Références

[1] AGT international. http://www.agtinternational.com/.

[2] Big-Data-Europe. https://github.com/big-data-europe/docker-flink. Demo of Apache Flink with Docker.

[3] Docker. https://www.docker.com/.

[4] DEBS. http://www.debs2017.org/.

[5] DEBS grand challenge. http://www.debs2017.org/call-for-grand-challenge-solutions/.

[6] Flink. https://flink.apache.org/.

[7] HOBBIT project. https://project-hobbit.eu/.

[8] Ontologies RDF. https://ckan.project-hobbit.eu/dataset/debs-grand-challenge-2017.

[9] RabbitMQ. https://www.rabbitmq.com/.

[10] Spark streaming. http://spark.apache.org/streaming/.

[11] Weidmüller. http://www.weidmueller.de/de/startseite.

[12] Fabian Hueske. *Introducing Stream Windows in Apache Flink*. 2015.

[13] Rémy Saissy. *Qu'est-ce que Storm ?* 2013.

[14] Bobby Evans Reza Farivar Tom Graves Mark Holderbaugh Zhuo Liu Kyle Nusbaum Kishorkumar Patil Boyang Jerry Peng Sanket Chintapalli, Derek Dagit and Paul Poulosky. *Benchmarking Streaming Computation Engine at Yahoo !* 2015.

[15] Petr Zapletal. *Comparison of Apache Stream Processing Frameworks*. 2016.

Annexes

A. Structures de données

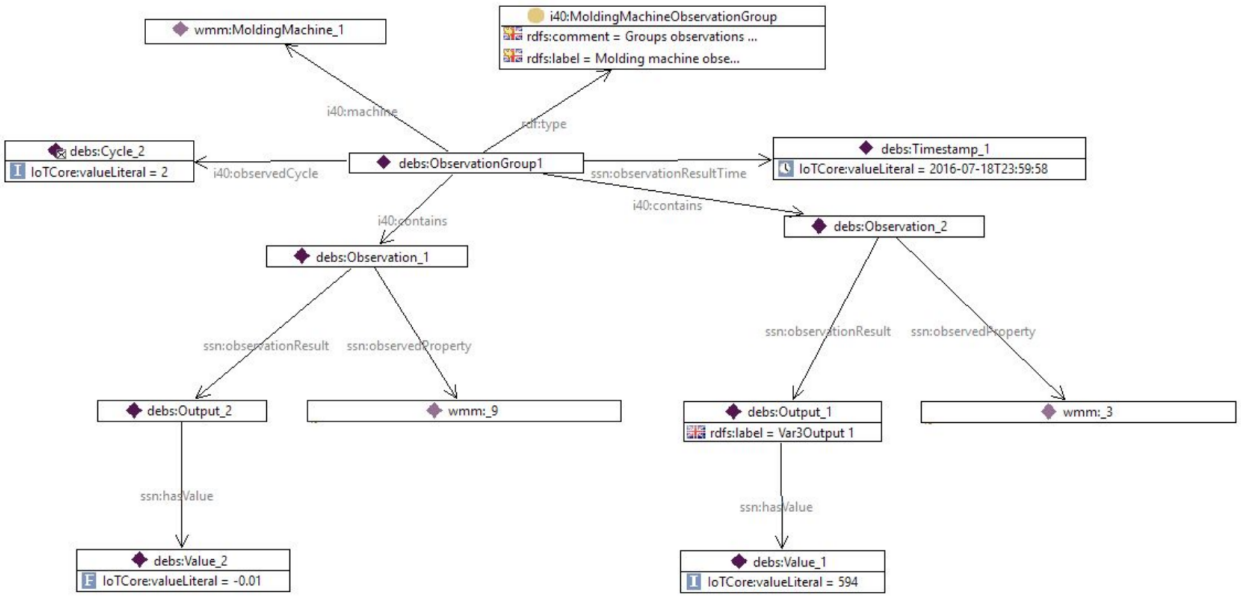


FIGURE 15 – Structure du Turtle en entrée

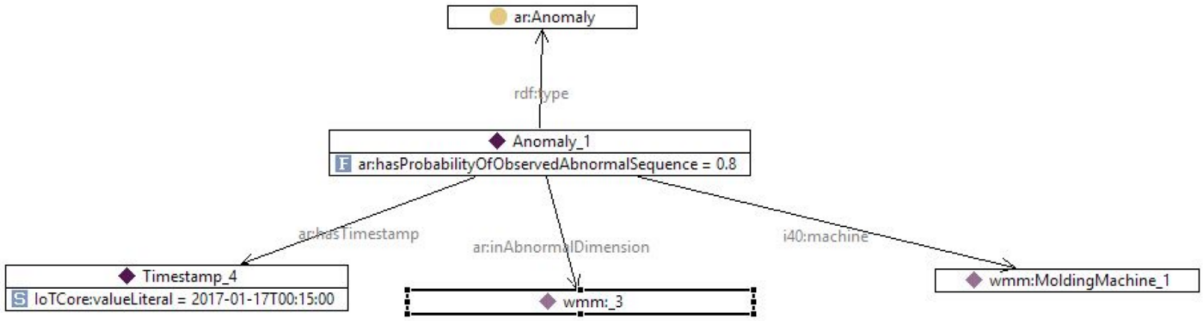


FIGURE 16 – Structure du Turtle en sortie

```
<http://project-hobbit.eu/resources/debs2017#ObservationGroup_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/I4.0#HoldingMachineObservationGroup> .
<http://project-hobbit.eu/resources/debs2017#ObservationGroup_0> <http://purl.oclc.org/NET/ssnx/ssn#observationResultTime> <http://project-hobbit.eu/resources/debs2017#Timestamp_0> .
<http://project-hobbit.eu/resources/debs2017#ObservationGroup_0> <http://www.agtinternational.com/ontologies/I4.0#machine> <http://www.agtinternational.com/ontologies/HeidmullerMetadata#Machine_57> .
<http://project-hobbit.eu/resources/debs2017#ObservationGroup_0> <http://www.agtinternational.com/ontologies/I4.0#observedcycle> <http://project-hobbit.eu/resources/debs2017#Cycle_0> .
<http://project-hobbit.eu/resources/debs2017#Cycle_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/I4.0#Cycle> .
<http://project-hobbit.eu/resources/debs2017#Cycle_0> <http://www.agtinternational.com/ontologies/IoTCore#valueLiteral> "13"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://project-hobbit.eu/resources/debs2017#Timestamp_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/IoTCore#Timestamp> .
<http://project-hobbit.eu/resources/debs2017#Timestamp_0> <http://www.agtinternational.com/ontologies/IoTCore#valueLiteral> "2017-01-01T01:00:10+01:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
<http://project-hobbit.eu/resources/debs2017#ObservationGroup_0> <http://www.agtinternational.com/ontologies/I4.0#observationGroupOutput> <http://project-hobbit.eu/resources/debs2017#Output_0> .
<http://project-hobbit.eu/resources/debs2017#Observation_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/I4.0#HoldingMachineObservation> .
<http://project-hobbit.eu/resources/debs2017#Observation_0> <http://purl.oclc.org/NET/ssnx/ssn#observationResult> <http://project-hobbit.eu/resources/debs2017#Output_0> .
<http://project-hobbit.eu/resources/debs2017#Observation_0> <http://purl.oclc.org/NET/ssnx/ssn#observedProperty> <http://www.agtinternational.com/ontologies/HeidmullerMetadata#_3> .
<http://project-hobbit.eu/resources/debs2017#Output_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://purl.oclc.org/NET/ssnx/ssn#SensorOutput> .
<http://project-hobbit.eu/resources/debs2017#Output_0> <http://purl.oclc.org/NET/ssnx/ssn#hasValue> <http://project-hobbit.eu/resources/debs2017#Value_0> .
<http://project-hobbit.eu/resources/debs2017#Value_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/I4.0#NumberValue> .
<http://project-hobbit.eu/resources/debs2017#Value_0> <http://www.agtinternational.com/ontologies/IoTCore#valueLiteral> "9433.11"^^<http://www.w3.org/2001/XMLSchema#double> .
```

FIGURE 17 – Exemple d'entrée en RDF, jusqu'au premier capteur du premier *observation group*. Les éléments extraits pour les calculs sont surlignés.

```
<http://PFEensai/outputs/debs2017#Anomaly_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/DEBSAnalyticResults#Anomaly> .
<http://PFEensai/outputs/debs2017#Anomaly_0> <http://www.agtinternational.com/ontologies/DEBSAnalyticResults#hasProbabilityOfObservedAbnormalSequence> "0.5"^^<http://www.w3.org/2001/XMLSchema#float> .
<http://PFEensai/outputs/debs2017#Anomaly_0> <http://www.agtinternational.com/ontologies/DEBSAnalyticResults#hasTimestamp> <http://project-hobbit.eu/resources/debs2017#Timestamp_0> .
<http://PFEensai/outputs/debs2017#Anomaly_0> <http://www.agtinternational.com/ontologies/DEBSAnalyticResults#hasMachineID> "HeidmullerMetadata#_8"> .
<http://PFEensai/outputs/debs2017#Anomaly_0> <http://www.agtinternational.com/ontologies/I4.0#machine> <http://www.agtinternational.com/ontologies/HeidmullerMetadata#HoldingMachine_3"> .
<http://PFEensai/outputs/debs2017#Timestamp_0> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.agtinternational.com/ontologies/IoTCore#Timestamp> .
<http://PFEensai/outputs/debs2017#Timestamp_0> <http://www.agtinternational.com/ontologies/IoTCore#valueLiteral> "2017-01-01T01:00:10+01:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
```

FIGURE 18 – Exemple de sortie en RDF

B. Métriques d'évaluation

La courbe suivante présente des paliers car les résultats affichés ont été réalisés avec un temps d'attente de 1s entre deux groupes d'observations par soucis de lisibilité.

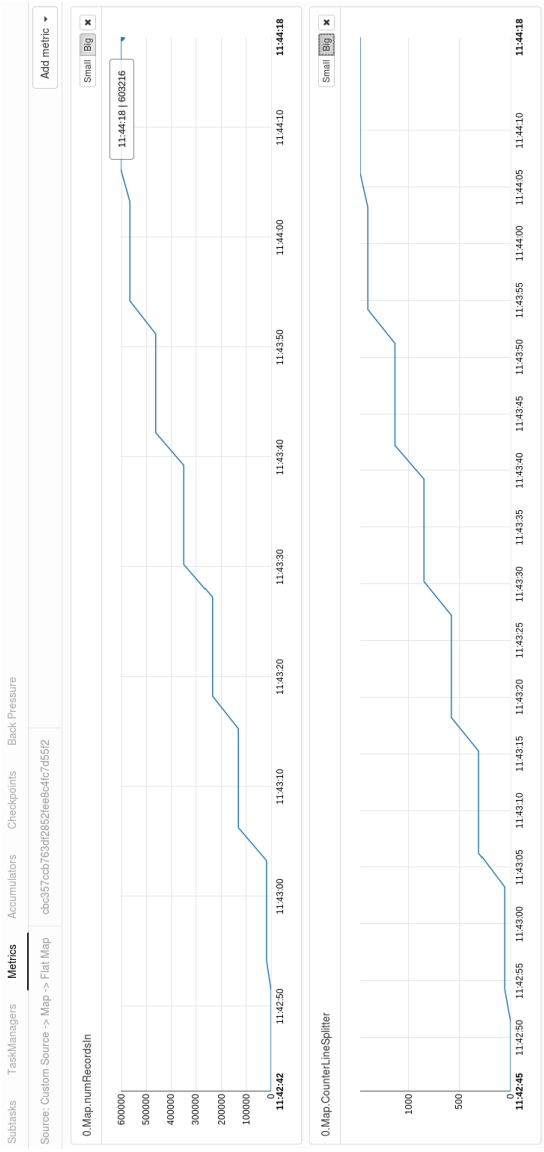


FIGURE 19 – Traffic dans LineSplitter

C. Question aux organisateurs sur le test de validité

Contexte

Les organisateurs ont proposé un jeu de données qui a été traité par leur solution de référence. Leur système détecte seulement 4 anomalies, toutes au 5$^{\text{ème}}$ horodatage (Timestamp_4). Parmi ces anomalies, deux concernent le capteur 31 de la machine 59. Nous détaillons ici la sortie attendue selon nous pour ce capteur.

Anomalies déclarées par les organisateurs :

— Machine 59, capteur 31, Timestamp_4, probabilité : 0.004115226337448559

— Machine 59, capteur 31, Timestamp_4, probabilité : 0.0035555555555555557

Calcul détaillé des probabilités

On s'intéresse aux données du capteur 31 de la machine 59, des temps 0 à 4. Les données observées sont :

Timestamp_0	1.2
Timestamp_1	1.19
Timestamp_2	1.2
Timestamp_3	1.2
Timestamp_4	1.2

K-Means

La séquence ne comprend que deux valeurs différentes. On note A et B les deux clusters obtenus : $A = 1.2$ et $B = 1.19$. La sortie de l'algorithme de clustering peut donc s'écrire :

Timestamp_0	1.2	cluster A
Timestamp_1	1.19	cluster B
Timestamp_2	1.2	cluster A
Timestamp_3	1.2	cluster A
Timestamp_4	1.2	cluster A

Apprentissage du modèle de Markov

On compte les transitions entre états au fil de la séquence d'observations. La liste des transitions observées est :

Horodatages	Transition	Nombre total d'occurrences
Timestamp_0 à Timestamp_1	A à B	1
Timestamp_1 à Timestamp_2	B à A	1
Timestamp_2 à Timestamp_3	A à A	2
Timestamp_3 à Timestamp_4	A à A	2

Notre interprétation de la méthode de détection attendue

La probabilité de transition d'un état i à un état j est estimée à partir de sa distribution empirique. On obtient donc les probabilités suivantes :

$P(A \to B) = \frac{1}{3}$; $P(A \to A) = \frac{2}{3}$; $P(B \to A) = \frac{1}{1}$; $P(B \to B) = 0$

Avec N = 5, le système effectue la détection sur les probabilités suivantes :

$P(A \to A) = 0.66666...$

$P(A \to A \to A) = 0.44444...$

$P(B \to A \to A \to A) = 0.44444...$

$P(A \to B \to A \to A \to A) = 0.14814...$

Nous n'avons trouvé aucun contexte de chaîne de Markov ou de K-means qui permette d'obtenir pour une de ces probabilités calculées les valeurs de 0.004115226337448559 et 0.0035555555555555557.

Remarque : Les résultats de notre système sont exactement ceux obtenus mathématiquement dans cette explication.