Συγκριτική ανάλυσης επίδοσης Flink vs. Samza για την διαχείριση μεγάλων δεδομένων ροής από φωτοβολταϊκές κυψέλες

Flink vs. Samza for the management of streaming big data from photovoltaic panels

ΤΜΗΜΑ ΗΛΕΚΤΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική εργασία του
Ρεβυθά Ιωακείμ

Επιβλέποντες:
Δημήτριος Κατσαρός Τσουκαλάς Ελευθέριος

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την ημερομηνία εξέτασης.

(Υπογραφή) (Υπογραφή)

....

Δημήτριος Κατσαρός Τσουκαλάς Ελευθέριος

Επίκουρος Καθηγητής Π.Θ. Καθηγητής Π.Θ.

Βόλος, Ιούνιος 2016

Σύνοψη

Στο κεφάλαιο ένα και δύο, γίνεται μια εισαγωγή στις βασικές αρχές και αρχιτεκτονικές συστημάτων κατανεμημένης επεξεργασίας και υψηλής απόδοσης. Γίνεται μια σύντομη αναφορά στο Hadoop και Storm, τα οποία αποτελούν τις βάσεις για τα μεταγενέστερα συστήματα όπως είναι το Flink και το Samza, τα οποία θα αναλύσουμε σε βάθος.

Μετέπειτα, στο κεφάλαιο τρία αναλύουμε αρχιτεκτονικά και προγραμματιστικά το Apache Flink framework, και δίνουμε βασικά παραδείγματα κατανόησης του συστήματος.

Συνεχίζοντας, αναλύουμε στο κεφάλαιο τέσσερα το Apache Kafka, το οποίο είναι ένα πρωτόκολλο για διαχείριση μηνυμάτων υψηλής απόδοσης σε φυσικά κατανεμημένο περιβάλλον, το οποίο συνεργάζεται άψογα με το Samza. Αυτός είναι ο λόγος που το αναλύουμε σε βάθος προγραμματιστικά και αρχιτεκτονικά

Στην συνέχεια, θα αναλυθεί το Apache Samza, το οποίο έχει πολλά κοινά στην αρχιτεκτονική από το Kafka όπως θα δούμε αφού έχει δημιουργηθεί με βάση αυτό.

Τέλος, κάνουμε μια πειραματική μελέτη συγκρίνοντας τα δύο αυτά συστήματα τόσο σε απόδοση, όσο και σε ταχύτητα παραγωγής κώδικα. Δημιουργούμε το μοντέλο producer-consumer το οποίο το εφαρμόζουμε και για τα δύο αυτά συστήματα και τα τρέχουμε σε ένα server για να βγάλουμε τους χρόνους σύγκρισης.

Λέξεις Κλειδιά

Apache Flink, Apache Samza, Apache Kafka, Cloud computing, Hadoop, Storm, Distributed computing, topics, producer-consumer

Abstract

In the first and second chapter, we introduce basic concepts like Hadoop and Storm. From those systems we have to understand the main idea, in order to build a system on Samza or Flink.

Next chapter, referred on Apache Flink concepts. We analyze system architecture and programming recipes also. In the end of this chapter, we quote basic examples.

Continue in the next chapter, we will analyze Apache kafka system, as a message query protocol. We refer on that because it is very important on Samza architecture. Someone can characterize Kafka as the “father” of Samza.

Chapter number five, has all basic concepts and architectures about Samza. We are giving some basic example again.

Final, we make an programming experiment. We build a model producer-consumer in both systems, and compare those in a server. We quote the time results and finally we can decide which system are better.

Keywords

Apache Flink, Apache Samza, Apache Kafka, Cloud computing, Hadoop, Storm, Distributed computing, topics, producer-consumer

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες κ. Δημήτριο Κατσαρό και κ. Τσουκαλά Ελευθέριο για την εμπιστοσύνη που μου έδειξαν και την βοήθεια που μου έδωσαν. Με την ευκαιρία αυτή θα ήθελα να ευχαριστήσω εξίσου τον κ. Χρήστο Γεωργόπουλο, κ. Θωμά Αρβανίτη και γενικότερα την εταιρεία Inaccess Networks, για την πολύτιμη βοήθεια και τεχνογνωσία που μου παρείχαν καθ' όλη την διάρκεια εκπόνησης την διπλωματικής μου εργασίας.

Ρεβυθάς Ιωακείμ, Βόλος, 2016

Περιεχόμενα

Σύνοψη

Λέξεις Κλειδιά

Abstract

Keywords

Ευχαριστίες

Περιεχόμενα

1.1 Εισαγωγή

1.2 Σκοπός

1.3 Δομή

Βασικά Συστήματα Επεξεργασίας Δεδομένων

2.1 Hadoop

2.2 Apache Storm

- 2.2.1 Βασικά Χαρακτηριστικά
- 2.2.2 Αρχιτεκτονική
- 2.2.3 Τοπολογίες
- 2.2.4 Ομαδοποίηση ροών

Apache Flink

3.1 Εισαγωγή στο Flink

3.2 Apache Flink Components

3.3 Μοντέλο Αρχιτεκτονικής και Διαχείρισης Καθηκόντων

3.4 Δομικά Στοιχεία Για Stream Processing

- Basics
- Handling State
- Application Development
- Large Deployments

3.5 Βασικά Χαρακτηριστικά

3.5.1 Χρονοσφραγίδες

3.5.2 Παράθυρα

3.6 Προγραμματιστική Υλοποίηση

3.6.1 DataStreams & DataTypes

Map

Flatmap

Filter

3.6.2 DataSources & DataSink

3.6.3 Windows

Apache Kafka

4.1 Αρχιτεκτονική

4.2 Υλοποίηση

4.3 Υλοποίηση ενός Kafka Consumer

4.4 Υλοποίηση ενός Kafka Producer

Apache Samza

5.1 Γενικά για το Samza

5.2 Αρχιτεκτονική του Samza

5.2.1 Streams and Jobs

5.3 Βασικό παράδειγμα κατανόησης

5.4 Προγραμματιστική υλοποίηση

5.5 Metrics

5.6 Checkpointing

5.7 Windowing

6.1 Συμπεράσματα

6.2 Samza Producer

6.3 Samza Consumer

6.4 Flink Producer

6.5 Flink Consumer

6.6 Τελικές μετρήσεις - χρόνοι

A.1 Zookeeper

Αναφορές

“The future of distributed computing is based on streams.”

ΚΕΦΑΛΑΙΟ 1

1.1 Εισαγωγή

Στις μέρες μας ο ρυθμός ανάπτυξης ηλεκτρονικών δεδομένων αυξάνεται με πολύ μεγάλους ρυθμούς. Όπως υπολογίζεται, από το 2005 μέχρι το 2020 τα ψηφιακά δεδομένα θα έχουν έναν συντελεστή ανάπτυξης 300, ήτοι από 130 exabytes σε 40.000 exabytes [1]. Το μέγεθος αυτό όλο και αυξάνεται αφού αυξάνονται αναλόγως κοινωνικά δίκτυα, εργαστηριακές μελέτες και γενικότερα οι ηλεκτρονικές συσκευές. Τέτοιες είναι από απλά smartphones χρηστών έως ιατρικά μηχανήματα. Το γεγονός-επανάσταση όμως της ύπαρξης του cloud, οδήγησε στην εκρηκτική ανάπτυξη δεδομένων καθώς σχεδόν όλες οι εταιρείες "ανεβάζουν" τα δεδομένα τους στο cloud για ασφάλεια, διαθεσιμότητα, επεκτασιμότητα, καθώς και οι απλοί χρήστες οι οποίοι χρειάζονται όλο και περσσότερο χώρο τον βρίσκουν εύκολα διαθέσιμο στο cloud. Αυτό καθιστά σαφές την ύπαρξη μηχανών επεξεργασίας οι οποίες θα μπορούν να διαχειρίζονται τεράστιο όγκο δεδομένων σε πραγματικό χρόνο (real-time streaming). Εφόσον το cloud είναι κατανεμημένο, θα πρέπει και οι τρόποι επεξεργασίας να δουλεύουν κατανεμημένα και αξιόπιστα. Η Google πρώτα είχε βγάλει έναν τρόπο επεξεργασίας μεγάλου όγκου δεδομένων ίσως όχι τόσο κοντά στο real-time processing, αλλά εύκολα κλιμακώσιμο. Αυτό είναι το γνωστό μοντέλο του Map-Reduce το οποίο πια έχει ενσωματωθεί στο Apache Hadoop framework [2].

1.2 Σκοπός

Ο σκοπός της παρούσας διπλωματικής θα είναι να γίνει σύγκριση απόδοσης ανάμεσα σε μηχανές επεξεργασίας δεδομένων πραγματικού χρόνου. Τέτοιες μηχανές επεξεργασίας μπορεί να είναι συστήματα τα οποία έχουν φτιαχτεί με την βοήθεια του framework Apache Storm, Spark, Samza και του πιο πρόσφατου Apache Flink. Στην παρούσα διπλωματική δεν θα αναλυθεί το Spark framework, καθώς δεν συνίσταται για streaming processing αλλά περισσότερο για batch processing. Για να πραγματοποιηθεί λοιπόν η σύγκριση των τριών αυτών συστημάτων, θα πρέπει πρώτα να γίνει η υλοποίησή τους. Αυτή θα πραγματοποιηθεί στα επόμενα κεφάλαια.

1.3 Δομή

Στο πρώτο και δεύτερο κεφάλαιο γίνεται μια σύντομη αναφορά στις βασικές αρχές των συστημάτων επεξεργασίας δεδομένων μεγάλης κλίμακας και ιδιαίτερα του Storm, που θεωρείται ο "πατέρας" τέτοιων συστημάτων.

Στο τρίτο κεφάλαιο αναλύουμε το Apache Flink, αρχιτεκτονικά και υλοποιούμε βασικές εφαρμογές του.

Στο τέταρτο κεφάλαιο αναλύουμε το Kafka σύστημα, το οποίο είναι ένα πολύ βασικό component για το Apache Samza το οποίο θα ακολουθήσει.

Στο πέμπτο κεφάλαιο αναλύουμε το δεύτερο σκέλος της διπλωματικής μας, το οποίο είναι το Apache Samza, τόσο αρχιτεκτονικά όσο και προγραμματιστικά.

Στο τελευταίο κεφάλαιο, κάνουμε μια πειραματική μελέτη, στην οποίο θα συγκρίνουμε τα δύο αυτά συστήματα και θα κάνουμε ανάλυση επίδοσης και θα παρουσιάσουμε τα αποτελέσματα.

Ακολουθούν μετά αναφορές και παραρτήματα.

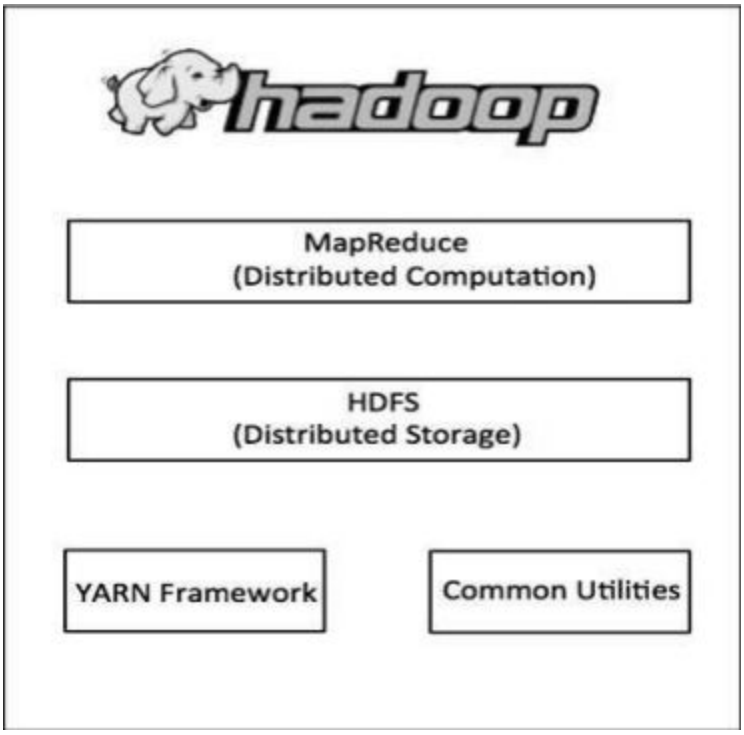
ΚΕΦΑΛΑΙΟ 2

Βασικά Συστήματα Επεξεργασίας Δεδομένων

2.1 Hadoop

Εφόσον θα αναλύσουμε το Storm framework, αξίζει να κάνουμε μια σύντομη αναφορά πρώτα στο Hadoop και αυτό διότι υπάρχει μία άμεση συσχέτιση μεταξύ τους. Το Storm ουσιαστικά είναι η έκδοση του Hadoop για real time & streaming processing σε αντίθεση με το Hadoop το οποίο είναι για batch processing.

Το Hadoop, όπως φαίνεται και στο παρακάτω σχήμα έχει την εξής απλή δομή:



Στο οποίο:

- το Common Utilities είναι όλες οι βιβλιοθήκες και εργαλεία τα οποία απαιτούνται για την σωστή λειτουργία του συστήματός μας
- το YARN Framework είναι υπεύθυνο να κάνει την χρονοδρομολόγηση των εργασιών σε daemons και για την σωστή κατανομή των πόρων στο σύστημα
- το HDFS (Distributed Storage) είναι ένα κατανεμημένο και κλιμακόσιμο (scalable) σύστημα αρχείων
- το MapReduce είναι ένα framework για την επεξεργασία μεγάλου όγκου δεδομένων το οποίο θα αναλυθεί παρακάτω

Το MapReduce λοιπόν, κρύβει από πίσω της την απλή ιδέα του διαίρει και βασίλευε (divide and conquer). Επομένως το αρχικό-μεγάλο πρόβλημα σπάσει σε πολλά μικρά προβλήματα στα οποία βρίσκει λύσεις πιο εύκολα για να συνθέσει την αρχική λύση του προβλήματος.

Όταν δώσουμε λοιπόν ένα input λοιπόν στο Hadoop framework, τότε το input μεταφέρεται στο κατανεμημένο σύστημα HDFS όπου σπάει σε κομμάτια εγγραφών <key, value> στα μηχανήματα. Ένα από αυτά τα μηχανήματα θα πρέπει να παίξει τον ρόλο του master, ο οποίος είναι υπεύθυνος για τον χρονοπρογραμματισμό και την χρονοδρομολόγιση των εργασιών στα υπόλοιπα μηχανήματα που συμμετέχουν στον υπολογισμό της λύσης του προβλήματος. Οι εργασίες που θα αναθέτει ο master κόμβος θα είναι είτε της φύσης Map είτε της φύσης Reduce. Οι συναρτήσεις Map & Reduce καθορίζονται κάθε φορά από τον εκάστοτε προγραμματιστή. Πρέπει να επιλέγονται έτσι, ώστε να εκμεταλλεύονται το βασικό πλεονέκτημα του framework, το οποίο είναι η δυνατότητα πλήρους παραλληλότητας του συστήματος.

Το μοντέλο αυτό πρέπει να έχει ανοχή στα σφάλματα γιατί αναπόφευκτα θα δημιουργούνται αρκετά λόγω της φύσης του συστήματος. Για να πετυχαίνεται λοιπόν αυτή η ανοχή στα σφάλματα (fault tolerance), ο master κόμβος παρακολουθεί “στενά” τους slave κόμβους για τυχόν αποτυχίες και πράττει αναλόγως.

2.2 Apache Storm

2.2.1 Βασικά Χαρακτηριστικά

Το σύστημα Apache Storm σε αντίθεση με το Hadoop επιτρέπει την κατανεμημένη επεξεργασία δεδομένων σε πραγματικό χρόνο. Τα πέντε χαρακτηριστικά που κάνουν το Strom να ξεχωρίζει είναι ότι:

- μπορεί να επεξεργαστεί πάνω από ένα εκατομμύριο πλειάδες ανά κόμβο ανά δευτερόλεπτο

- είναι κλιμακώσιμο (scalability) αρκεί να προστεθούν περισσότερα μηχανήματα στο cluster. Σε αυτό απαιτείται η χρήση του Zookeeper το οποίο είναι ένα εργαλείο συγχρονισμού και συντονισμού κατανεμημένων συστημάτων

- έχει μεγάλη ανοχή στα σφάλματα (fault tolerance) καθώς, εάν κάποιος κόμβος εργάτης σταματήσει να λειτουργεί τότε το Storm θα επανεκκινήσει την διεργασία αυτή σε κάποιον άλλον που λειτουργεί

- εγγύηση ότι όλα τα δεδομένα τα οποία θα εισαχθούν στο σύστημα θα επεξεργαστούν στο ακέραιο (data processing guarantee)

- ευκολία στην χρήση, την επίβλεψη και στην εγκατάσταση του καθώς υποστηρίζει σχεδόν όλες τις γλώσσες και παρέχει γραφικό περιβάλλον για την παρακολούθηση του όλου συστήματος

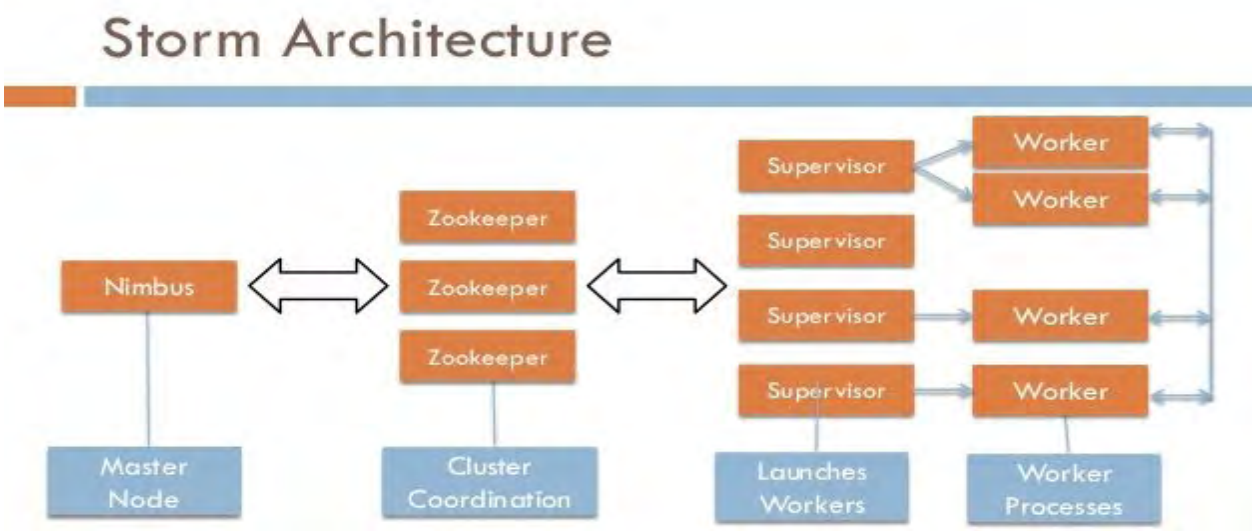
Στο Storm, υπάρχουν clusters υπολογιστών οι οποίοι αποτελούν μία τοπολογία (topology). Η τοπολογία δεν σταματά να υπολογίζει εφόσον η ροή των δεδομένων είναι συνεχής και άπειρη, σε αντίθεση με το Hadoop, όπου σταματάει μόλις ολοκληρωθεί η επεξεργασία των δεδομένων εισόδου.

2.2.2 Αρχιτεκτονική

Το cluster του Storm ακολουθεί την αρχιτεκτονική του master-slave. Υπάρχει λοιπόν ένας κόμβος master, ο οποίος ονομάζεται δαίμονας Nimbus, ο οποίος είναι

υπεύθυνος για να μοιράζει και να χρονοδρομολογεί τις εργασίες στους Supervisor (slaves) μέσω του Zookeeper.

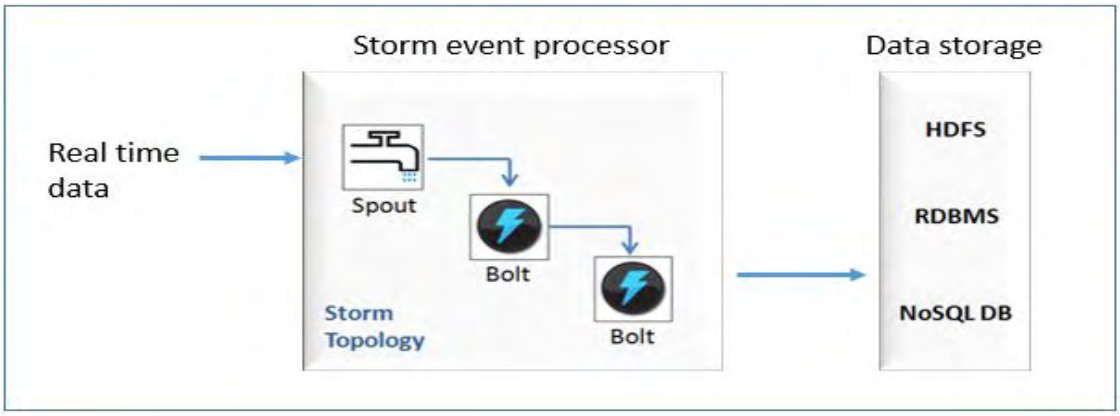
Οι Supervisors λοιπόν είναι κόμβοι-δαίμονες, οι οποίοι είναι υπεύθυνοι για την έναρξη, την επίβλεψη και τον τερματισμό των workers. Workers είναι ουσιαστικά threads, τα οποία εκτελούν υποσύνολα διεργασιών. Στο παρακάτω σχήμα φαίνεται η βασική αρχιτεκτονική.



2.2.3 Τοπολογίες

Όπως αναφέραμε και πριν, το Storm cluster τρέχει μία ή και περισσότερες τοπολογίες. Μία τοπολογία αναπαριστάται από άκυκλους γράφους, οι οποίοι αποτελούνται από 2 ειδών κόμβων, από ένα spout ή ένα bolt. Ένα spout αποτελεί

το κύριο σημείο εισαγωγής δεδομένων στο σύστημα. Ένα spout συνήθως διαβάζει ταυτόχρονα από διαφορετικά συστήματα ουράς δεδομένων όπως το RabbitMQ ή το Kafka. Από την άλλη πλευρά υπάρχουν τα bolts, τα οποία διαβάζουν τα δεδομένα από τα spouts, και εφαρμόζουν one step συναρτήσεις επεξεργασίας. Τέτοιες είναι το φιλτράρισμα, αποθήκευση δεδομένων ή και επικοινωνία με εξωτερικούς παράγοντες. Το παρακάτω σχήμα είναι ένα χαρακτηριστικό παράδειγμα.



Ανάλογα με τον ρυθμό άφιξης των δεδομένων στο σύστημά μας το Storm μας δίνει την δυνατότητα να κάνουμε όλο και περισσότερη παράλληλη κατανεμημένη επεξεργασία. Αυτό προγραμματιστικά γίνεται πολύ εύκολα αρκεί να υπάρχουν διαθέσιμα μηχανήματα τα οποία έχουν επεξεργαστική ισχύ διαθέσιμη. Ένα παράδειγμα είναι το εξής:

```
builder.setBolt("id_bolt", new ονομα_κλασης(), #αριθμος_παραλλιλισμου)
        .shuffleGrouping("string_componentID ");
```

Ο αριθμός παραλληλισμού που δίνουμε είναι άμεσα συνδεόμενος με τους executors(thread) που δουλεύουν παράλληλα για ένα task της τοπολογίας.

2.2.4 Ομαδοποίηση ροών

Ένα από τα πιο σημαντικά πράγματα που πρέπει να λάβουμε υπόψιν μας όταν σχεδιάζουμε μία τοπολογία, είναι με ποιόν τρόπο θα γίνει η ομαδοποίηση των ροών (stream grouping). Μία ομαδοποίηση ροών ουσιαστικά, καθορίζει τον ακριβή τρόπο με τον οποίο θα διανέμονται και θα καταναλώνονται οι πλειάδες (tuples) από τα bolts. Ένας κόμβος, (bolt ή spout) μπορεί να εκπέμπει παραπάνω

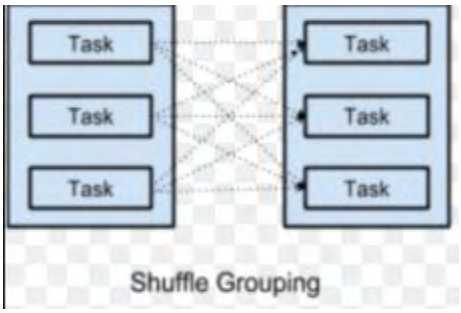
από ένα stream δεδομένων. Τότε, με το stream grouping, είμαστε σε θέση να επιλέγουμε κάθε φορά ποιο stream δεδομένων θέλουμε να πάρουμε.

Ένα stream grouping ορίζεται ακολούθως στον κώδικα:

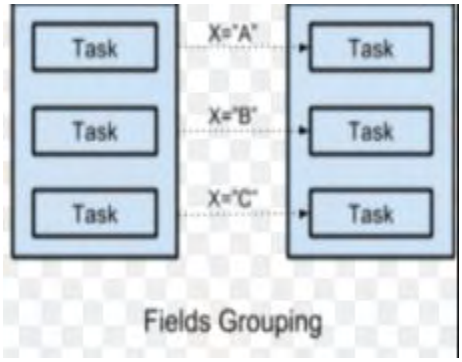
```
builder.setBolt("id_bolt", new ονομα_κλασης(), #αριθμος_παραλλιλισμου)
        .shuffleGrouping("string_componentID");
```

Το Storm μας παρέχει 7 διαφορετικούς τρόπους grouping. Τέτοιοι είναι οι εξής:

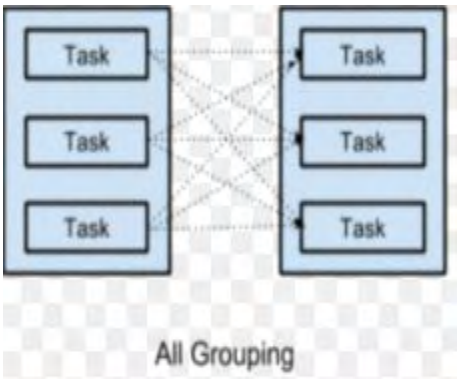
- Shuffle Grouping: Είναι ίσως ο πιο κοινός τρόπος grouping. Και αυτό διότι αυτή είναι μία τεχνική η οποία προσπαθεί να κρατήσει ισορροπία στην κατανομή του φόρτου. Ουσιαστικά μας εγγυάται ότι όλα τα bolts θα δεχτούν σχεδόν ίσες πλειάδες ταυτόχρονα. Αυτό είναι μία καλή τεχνική για ατομικές λειτουργίες. Αυτή η τεχνική όμως μπορεί να γίνει μειονέκτημα σε κάποιες περιπτώσεις γιατί στέλνει τις πλειάδες τυχαία και κατανεμημένα.



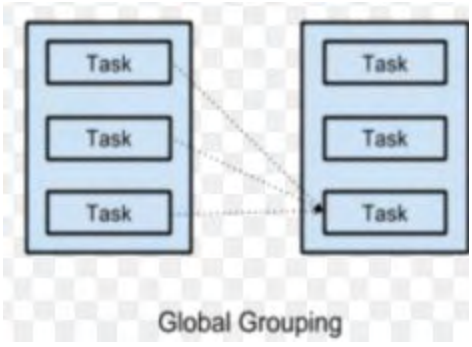
- Field Grouping: Σε αυτό το είδος, οι πλειάδες στέλνονται στα bolts ανάλογα με το όνομα που δίνουμε ως παράμετρο. Δηλαδή, η ποή κατηγοριοποιείται και στέλνεται σε κάθε bolt ανάλογα με το είδος της κατηγορίας (field) που πρέπει.



- All Grouping: Εδώ στέλνεται ένα αντίγραφο της κάθε πλειάδας σε όλα τα instances των bolts. Αυτή η τεχνική χρησιμοποιείται συνήθως για να στέλνει σήματα, πχ. σήματα acknowledgments.



- Global Grouping: Σε αυτήν την τεχνική, όλες οι πλειάδες όλων των instances στέλνονται σε έναν στόχο bolt. (Αυτό με το μικρότερο ID)



- Direct Grouping: Αυτό είναι ένα ειδικό είδος grouping στο οποίο ο παραγωγός της πλειάδας αποφασίζει σε ποιο bolt θα σταλεί για επεξεργασία η πλειάδα αυτή.

- Custom Grouping: Εδώ μας δίνεται η δυνατότητα να φτιάξουμε το δικό μας πρωτόκολλο επικοινωνίας των bolts. Για να γίνει αυτό πρέπει να υλοποιήσουμε το **type.storm.grouping.CustomStreamGrouping** interface.

- None Grouping: Προς το παρών (Storm-0.10.0 release) λειτουργεί όπως το shuffle grouping.

ΚΕΦΑΛΑΙΟ 3

Apache Flink

3.1 Εισαγωγή στο Flink

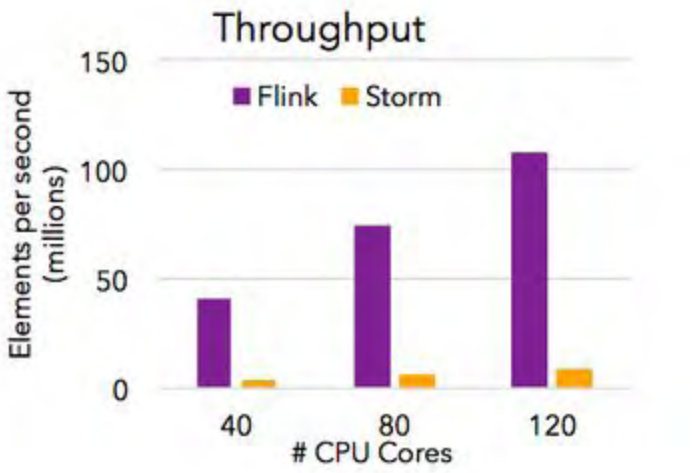
Το Flink (το οποίο πριν αγοραστεί από την Apache ονομαζόταν Stratosphere), είναι ένα σύστημα κατανεμημένης επεξεργασίας δεδομένων, τόσο δεδομένων συνεχής ροής όσο και δεδομένων batch. Στην παρούσα διπλωματική μας ενδιαφέρει η επεξεργασία ροής και εκεί είναι που θα εμβαθύνουμε και θα κάνουμε τα πειράματά μας. Το Flink, έχει δημιουργηθεί με βάση την ιδέα του Hadoop MapReduce, παρόλα αυτά έχει το δικό του Runtime και αρχιτεκτονική. Εν γένει, συνεργάζεται άψογα με το Hadoop Distributed File System (HDFS) για να διαβάζει και να γράφει δεδομένα, γεγονός που απαιτεί μόνο ορισμένες βιβλιοθήκες και όχι ολόκληρη την πλατφόρμα Hadoop. Αξίζει να αναφερθεί ότι το Flink υπερέχει σε ταχύτητα στο streaming processing (low latency) έναντι του batch processing (high latency).

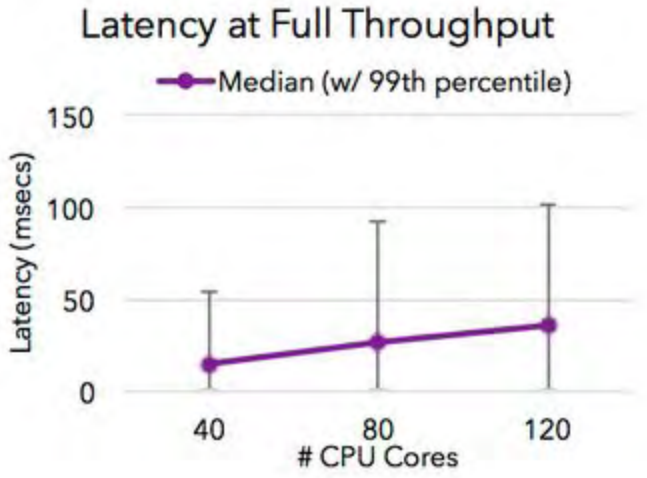
Από την άλλη πλευρά όμως, έχουμε το Storm, το οποίο έχει πολλές ομοιότητες με το Flink. Τόσο το Storm όσο και το Flink μπορούν να αναπαραστήσουν το dataflow τους σε γράφους. Ομοιότητα ακόμα υπάρχει και στα σημεία εισόδου και επεξεργασίας, αφού εκεί στα spouts αντιστοιχούν τα maps, και στα bolts τα flatMaps. Παράλληλα, το Flink προσφέρει ένα μηχανισμό εγγύησης ότι τα δεδομένα θα επεξεργαστούν στο ακέραιο, αντίστοιχο με αυτό που έχει το Storm.

Αυτό που κάνει το Flink να ξεχωρίζει στο streaming processing είναι τα εξής 4 χαρακτηριστικά:

- Low latency
- High throughput
- Low overhead στον μηχανισμό ανοχής βλαβών
- consistency, πραγματοποιώντας περιοδικά στιγμιότυπα Lamport

Συγκριτικά με το Storm έχουμε τις παρακάτω μετρικές:



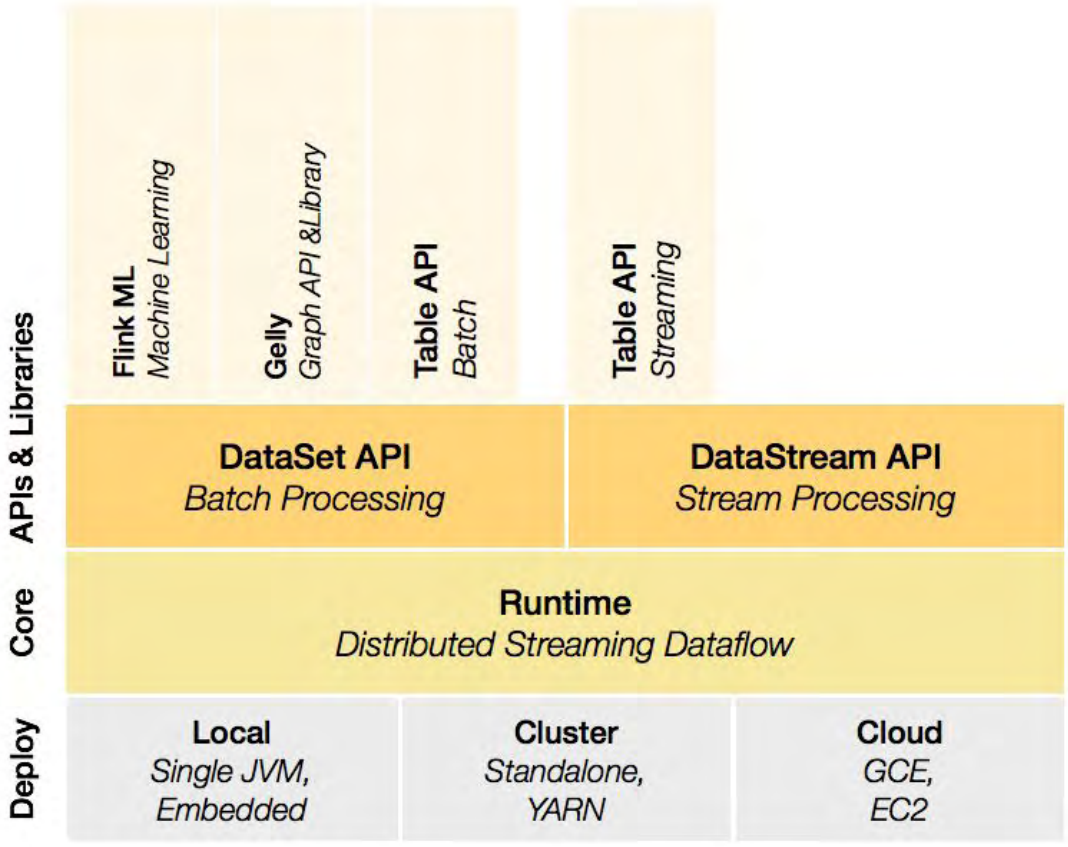


3.2 Apache Flink Components

Το Apache Flink δίνει ένα high-level API (Trident) για επεξεργασία σε:

- Machine Learning (Flink ML)
- Graphs (Gelly)
- Relational Data (Table API)

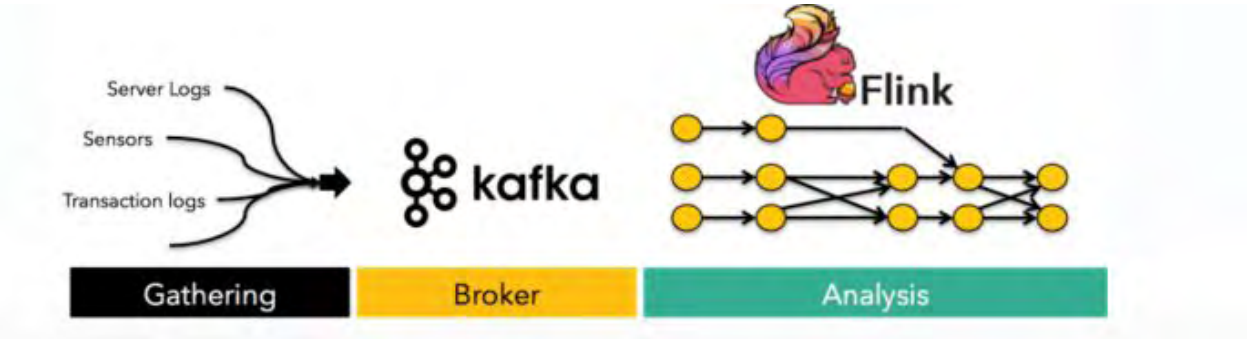
Στην παρακάτω εικόνα φαίνεται τα components του Flink.



Αξίει να σημειωθεί ότι από την έκδοση “1.1.0” και μετά του Apache Flink υπάρχει η δυνατότητα στο ML component να γίνει πρόβλεψη των δεδομένων με βάση την τρέχουσα συμπεριφορά. Για να επιτευχθεί μία καλή πρόβλεψη πρλεπει να θέτουμε

σωστά όρια στα παράθυρα και να έχουμε αρκετό όγκο δεδομένων που έχουν ήδη επεξεργαστεί.

Το Apache Flink όμως, συνήθως δεν βρίσκεται μόνο του μέσα σε ένα σύστημα. Συνήθως συνεργάζεται με Apache Kafka ή RabbitMQ. Ένα συμβολικό σχήμα που παρουσιάζει αυτήν την αρχιτεκτονική φαίνεται παρακάτω.

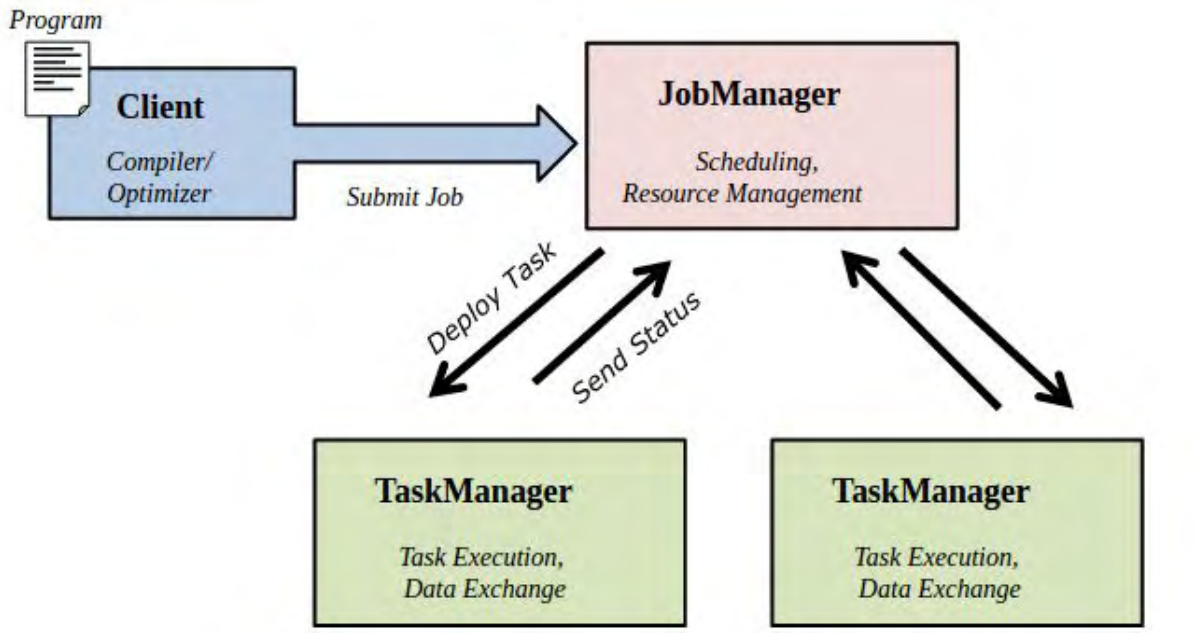


Όπως παρατηρούμε, δεδομένα έρχονται από κατανεμημένες πηγές στο Kafka σύστημά μας, (το οποίο θα αναλυθεί παρακάτω), και αφού αποθηκευτούν στα σωστά topics, το Flink θα τα κάνει streaming και θα τα αναλύσει.

3.3 Μοντέλο Αρχιτεκτονικής και Διαχείρισης Καθηκόντων

Όταν το Flink σύστημα ξεκινάει, "ξυπνάει" ταυτόχρονα τον JobManager και έναν ή περισσότερους TaskManagers. Ο JobManager είναι οι συντεταγμένες του συστήματός μας, ενώ οι TaskManagers είναι οι εργάτες οι οποίοι εκτελούν κομμάτια κώδικα τα οποία είναι σε παράλληλα προγράμματα ή νήματα. Όταν ξεκινήσει έναν σύστημα τοπικά, δηλαδή σε local mode, ο JobManager και οι TaskManagers τρέχουν τοπικά στο ίδιο JVM.

Αξίζει να σημειωθεί ότι για να γράψουμε ένα πρόγραμμα για Flink σύστημα, το γράφουμε είτε σε Scala είτε σε Java. Αφού γράψουμε το πρόγραμμά μας, το ανεβάζουμε στον server του Flink όπου δημιουργείται εκείνη την στιγμή ένας client ο οποίος κάνει μια προ-επεξεργασία στην οποίο το πρόγραμμα που έχουμε γράψει σπάει σε παράλληλα και ανεξάρτητα modules στο data flow τα οποία εκτελούνται από τον JobManager και τους TaskManagers. Αυτή η διαδικασία φαίνεται στο επόμενο γράφημα που ακολουθεί.

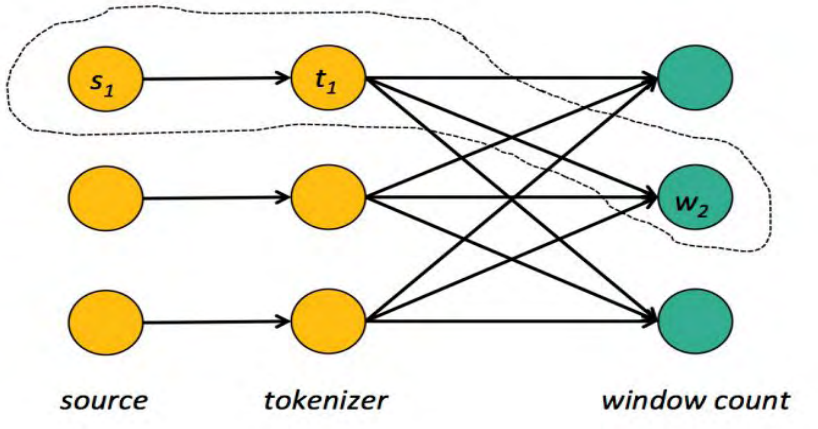


3.4 Δομικά Στοιχεία Για Stream Processing

Για να "στηθεί" το Flink ως stream processor, πρέπει να αναγνωριστούν 8 δομικά στοιχεία τα οποία είναι απαραίτητα για να θεωρήσουμε το σύστημά μας ως τέτοιο. Αυτά τα 8 στοιχεία [3] κατηγοριοποιούνται σε 4 κατηγορίες, όπου η κάθε κατηγορία εμπεριέχει 2 στοιχεία:

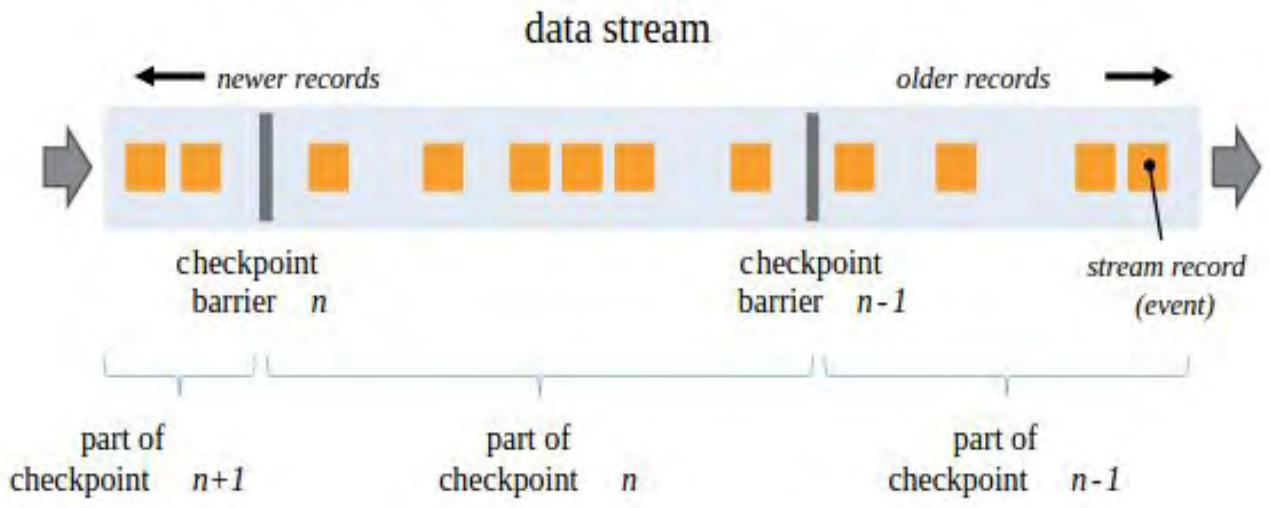
Basics

Pipelining: το οποίο σημαίνει ότι όλα τα tasks πρέπει να είναι online την ίδια στιγμή και να μπορούν να περάσουν ελεύθερα μέσα από το σύστημα χωρίς καθυστερήσεις ακόμα και αν βρίσκονται σε διαφορετικούς κόμβους. Όπως φαίνεται και στο παρακάτω σχήμα, τα 3 παράλληλα tasks s1, t1 και w2, πρέπει να τρέξουν ταυτόχρονα και να επιτρέπουν τα δεδομένα σε μια συνεχείς ροή μέσω του pipeline.



Replay: Ένα data streaming σύστημα πρέπει να είναι σε θέση να κάνει "replay" διάφορα μέρη της ροής του συστήματος αναπαράγοντας αποτελέσματα για τον χρήστη. Το replay είναι σημαντικό και διάφορους λόγους, κυρίως όμως είναι σημαντικό για ανοχή στα σφάλματα, και ανάκτηση από τις πιθανές αστοχίες του

συστήματος. Όπως φαίνεται και στο σχήμα, το Flink περιοδικά εισάγει "checkpoint barriers" μέσα στα data streams.



Handling State

Operator state: το οποίο σημαίνει ότι σε αντίθεση με το batch processing το οποίο βασίζεται σε αμετάβλητους μετασχηματισμούς σε σύνολα δεδομένων, το stream πρέπει να διατηρεί κατάσταση μέσω συναρτήσεων μετασχηματισμού όπως είναι η map() ή όπως είναι η filter().

State backup and restore: το οποίο σε συνεργασία με το Replay βοηθάει στην ανοχή στα σφάλματα και τις αστοχίες του συστήματος.

Application Development

High-level APIs and Integration with batch sources: βοηθάει για αποδοτικότερη δημιουργία εφαρμογών συνδυάζοντας streams και static sources.

Large Deployments

High Availability: το οποίο χρειάζεται όταν οι streaming διεργασίες θα πρέπει να τρέχουν για μεγάλες χρονικές περιόδους.

Automatic scale-out and scale-in: το οποίο χρειάζεται για να αλλάζουμε την παραλληλότητα του συστήματος χωρίς να χάνουμε την λειτουργικότητά του. Για να επιτευχθεί αυτό κλείνουμε το dataflow με ένα checkpoint και επαναφέρουμε το dataflow μετά με διαφορετική παραλληλότητα.

3.5 Βασικά Χαρακτηριστικά

Για να υλοποιήσουμε ένα κατανεμημένο πρόγραμμα σε Apache Flink, πρέπει να λάβουμε υπόψιν μας κάποια βασικά χαρακτηριστικά τα οποία πρέπει να εμπεριέχει το πρόγραμμά μας. Τέτοια είναι το είδος του παραθύρου που θα χρησιμοποιήσουμε ή ακόμα και οι χρονοσφραγίδες.

3.5.1 Χρονοσφραγίδες

Για να βάλουμε στο σύστημά μας τις χρονοσφραγίδες πρέπει να ενσωματώσουμε πρώτα στον κώδικά μας το Event Time, το οποίο δίνει μία τιμή (χρονοσφραγίδα) σε κάθε στοιχείο (element) του stream. [4] Για να αναθέσουμε χρονοσφραγίδες υπάρχουν 2 τρόποι:

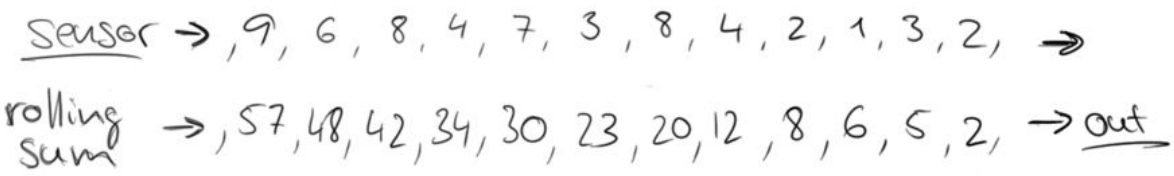
- Απευθείας στο data stream source
- Μέσω του TimestampAssigner

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

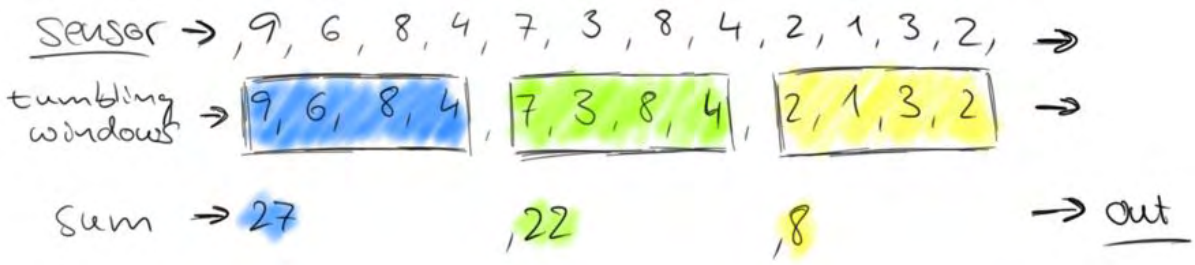
3.5.2 Παράθυρα

Τα παράθυρα είναι ίσως ένα από τις πιο βασικές παραμέτρους που πρέπει να λάβουμε υπόψιν μας όταν χτίζουμε το σύστημά μας. Τα παράθυρα μπορεί να είναι atomic, tumbling είτε sliding.

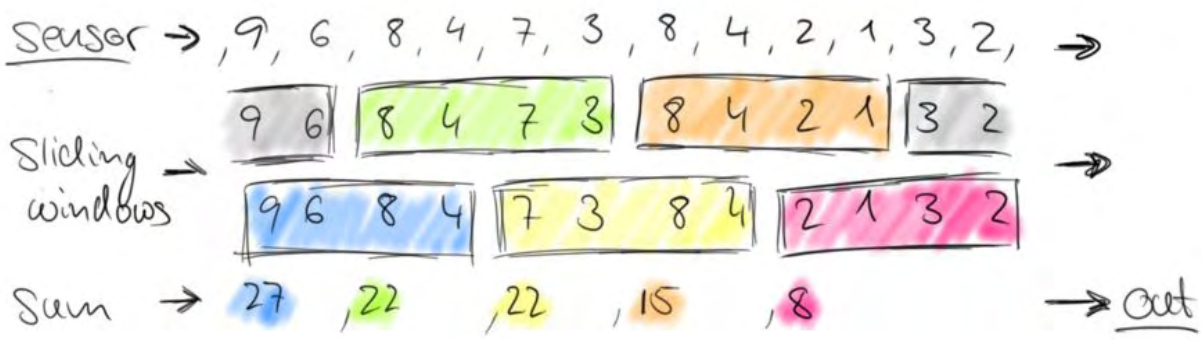
Atomic παράθυρο, είναι ουσιαστικά ο εκφυλισμός παραθύρου καθώς μιλάμε για ένα παράθυρο με μία και μόνο τιμή. Το παράθυρο αυτό είναι στην πραγματικότητα ένας μετρητής ο οποίος προσθέτει τις τιμές real-time. Στο ακόλουθο παράδειγμα, έχουμε τιμές από τον αισθητήρα μας από ένα φωτοβολταϊκό panel. Ο μετρητής προσθέτει τις τιμές και υπολογίζει το άθροισμα καθώς φτάνουν στο σύστημα. Το μειονέκτημα σε αυτήν την τεχνική όμως είναι ότι εφόσον δουλεύουμε σε κατανεμημένο σύστημα θα υπάρχουν και αποκλίσεις στον χρόνο λόγω δικτύου, πράγμα το οποίο είναι μη-διαχειρίσιμο με αυτήν την τεχνική.



Ένα άλλο παράθυρο όπως αναφέραμε είναι το Tumbling, το οποίο κάνει μια σταθερή ομαδοποίηση σε συγκεκριμένο χρονικό διάστημα το οποίο είναι σταθερό. Έτσι, δημιουργεί για παράδειγμα σταθερά παράθυρα και μη-επικαλυπτόμενα, για παράδειγμα κάθε ένα λεπτό και υπολογίζει το άθροισμα όπως φαίνεται στο παρακάτω παράδειγμα.



Το τρίτο και τελευταίο παράθυρο είναι το sliding παράθυρο, το οποίο είναι ένα πιο ελαστικό παράθυρο το οποίο δεν είναι επικαλυπτόμενο εάν χρειαστεί. Για αυτόν τον λόγο είναι και το πιο δημοφιλές. Για παράδειγμα, εάν θέλουμε να υπολογίζουμε το άθροισμα της τιμής του φωτοβολταϊκού αισθητήρα το τελευταίο λεπτό κάθε 30 δευτερόλεπτα, το παράθυρο θα φαινόταν κάπως έτσι:



3.6 Προγραμματιστική Υλοποίηση

3.6.1 DataStreams & DataTypes

Μια από τις βασικές αναφορές που πρέπει να κάνουμε, είναι στους τύπους δεδομένων (Data Types) του Flink framework οι οποίοι αποτελούν βασικό στοιχείο για την υλοποίηση των προγραμμάτων. Το Flink μπορεί να συνεργαστεί με πολλούς και διαφορετικούς τύπους δεδομένων, απλούς ή σύνθετους. Απλοί τύποι μπορεί να είναι ένα Array, ένα String ή ακόμα ένα Integer. Από την άλλη πλευρά υπάρχουν οι σύνθετοι τύποι οι οποίοι είναι και πιο συχνά χρησιμοποιούμενοι. Γνωστή δομή σύνθετων τύπων είναι τα Tuples, τα οποία είναι ο πιο απλός και πιο ελαφρύς τρόπος για να ενθυλακώσουμε δεδομένα στο Flink. Είναι ένας εύκολος και ευέλικτος τρόπος αφού το framework μας δίνει την δυνατότητα να χρησιμοποιήσουμε 25 διαφορετικά Tuples.

Ακολουθεί ένα παράδειγμα στο οποίο χρησιμοποιεί ένα Tuple για να κρατάμε 4 βασικές τιμές κάθε φορά από έναν αισθητήρα. Θα κρατάμε την τοποθεσία, Id, τιμή και αν λειτουργεί το μηχάνημα.

```
Tuple4<String, String, Double, Boolean> sensor =new Tuple4<>("Greece",
                                                    "1234-dfrg-7677-hjh5", 0.00256, true);
String location = sensor.f0;
String sensorID = sensor.f1;
Double value = sensor.f2;
Boolean working = sensor.f3;
```

Ένα άλλο βασικό σημείο που πρέπει να αναφερθεί είναι αυτό των μετασχηματισμών. Υπάρχουν αρκετοί τρόποι μετασχηματισμού δεδομένων, αλλά αυτοί που ξεχωρίζουν είναι:

Map

Η map είναι η κλασσική υλοποίηση μιας map function. Δέχεται μία τιμή και παράγει μια άλλη (one-to-one mapping).

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);
```

```
// Regular Map - Takes one element and produces one element
DataStream<Integer> doubleIntegers = integers.map(new MapFunction<Integer, Integer>() {
        @Override
        public Integer map(Integer value) {
                return value * 2;
        }
});
doubleIntegers.print();


> Result from Terminal: 2, 4, 6, 8
```

Flatmap

Η FlatMap, είναι μια map function η οποία μοιάζει με την map την οποία μόις περιγράψαμε με την διαφορά ότι δεν έχουμε τον περιορισμό να επιστρέψουμε ένα στοιχείο. Έχουμε την πολυτέλεια να επιστρέψουμε κανένα, ένα ή και περισσότερα στοιχεία. Χαρακτηριστικό παράδειγμα μπορεί να θερωρηθεί το ακόλουθο:

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);
DataStream<Integer> doubleIntegers2 = integers.flatMap(new FlatMapFunction<Integer,
Integer>() {
        @Override
        public void flatMap(Integer value, Collector<Integer> out) {
                out.collect(value * 2);
        }
});
doubleIntegers2.print();


> Result from Terminal: 2, 4, 6, 8
```

Filter

Το Filter, είναι πού σημαντική και χρήσιμη transformation function ειδικά για streaming συστήματα που θέλουν επεξεργασία on-the-fly. Ουσιαστικά δέχεται δεδομένα και τα φιλτράρει. Επιστρέφει τα δεδομένα τα οποία πληρούν την προϋπόθεση την οποία έχουμε θέσει εμείς με δική μας “@Override” συνάρτηση.

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);
DataStream<Integer> filtered = integers.filter(new FilterFunction<Integer>() {
        @Override

        public boolean filter(Integer value) {
                return value != 3;
        }}
);
filtered.print();


> Result from Terminal: 1, 2, 4
```

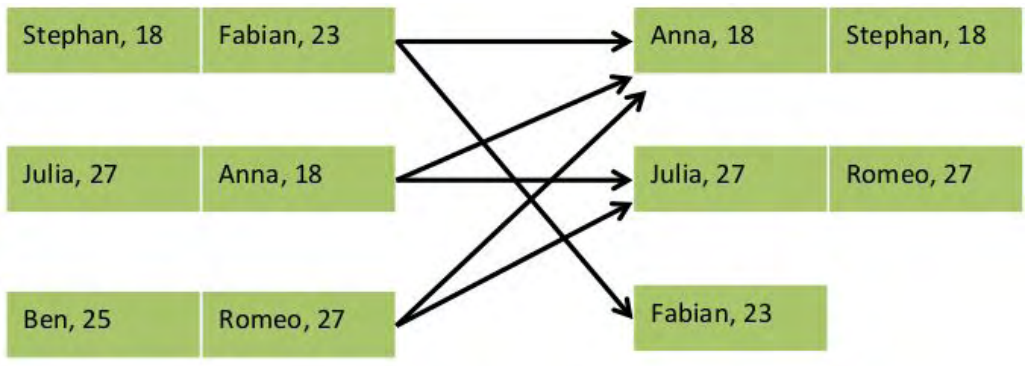
Αξίζει να σημειωθεί ότι κάνουμε δικές μας υλοποιήσεις στις map functions, flatmap και filters τις οποίες τις κάνουμε Override.

Συνεχίζοντας, πρέπει να αναφερθούμε στον λογικό μετασχηματισμό, ο οποίος συμβαίνει με την KeyBy συνάρτηση. Αυτή, χωρίζει τα δεδομένα ενός DataStream σε partiotions με το ίδιο "κλειδί". Το κλειδί μπορεί να είναι μια τιμή από την πλειάδα μας, π.χ. το "name" ή ακόμα μπορεί να δημιουργήσουμε μια δική μας συνάρτηση μέσω του KeySelector που να αποφασίζει ποιο key θα διαλέξει κάθε φορά. Ένα χαρακτηριστικό παράδειγμα κώδικα είναι το ακόλουθο, στο οποίο παραθέτεται και μια σχηματική υλοποίηση για περαιτέρω κατανόηση.

```
// We choose "age" as a key

// (name, age) of employees
DataStream<Tuple2<String, Integer>> passengers = ...

// group by second field (age)
DataStream<Integer, Integer> grouped = passengers.keyBy(1)
```



3.6.2 DataSources & DataSink

Το Flink προσφέρει διάφορους μηχανισμούς για είσοδο στο σύστημα. Η είσοδος μπορεί αν έρχεται από ένα αρχείο, από ένα socket, από μία Source Function την οποία θα δημιουργήσουμε εμείς για να παράγει πλασματικό δεδομένα ή από κάποιο σύστημα ουράς όπως είναι το RabbiMq ή το Kafka. Να τονιστεί ότι παρ' όλο που είμαστε σε streaming system η είσοδος από αρχείο είναι απολύτως αποδεκτή.

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// read text socket from port
DataStream<String> socketLines = env       .socketTextStream("localhost", 9999);

// read a text file ingesting new elements every 100 milliseconds
DataStream<String> localLines = env.readFileStream("/path/to/file",
                                                   100,

WatchType.PROCESS_ONLY_APPENDED);


// read data stream from custom source function
DataStream<<Tuple2<Long, String> stream = env.addSource(new MySourceFunction());


// read from my kafka topic
DataStream<String> messageStream = env.addSource(
                                          new FlinkKafkaConsumer082<>(
                                                parameterTool.getRequired("topic"),
                                                new SimpleStringSchema(),
                                                parameterTool.getProperties())
                                          );
```

Το “FlinkKafkaConsumer082”, είναι ένας consumer ο οποίος διαβάζει από ένα kafka topic. Για να μπορέσουμε να τον ενσωματώσουμε στον κώδικά μας πρέπει να προσθέσουμε στο αντίστοιχο pom.xml του maven project μας τον ακόλουθο κώδικα.

```
<dependency>
        <groupId>org.apache.flink</groupId>
         <artifactId>flink-connector-kafka-0.8_2.10</artifactId>
```

```
        <version>1.1-SNAPSHOT</version>
</dependency>
```

Εκτός από αυτό, πριν καλέσουμε τον FlinkKafkaConsumer082, πρέπει να καλέσουμε και μια Property function όπου μέσα να περάσουμε όλες τις ρυθμίσεις που χρειάζονται για να συνδέσουμε τον kafka με το flink πρόγραμμά μας. Κάτι τέτοιο μπορεί να γίνει με τον ακόλουθο κώδικα.

```
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test-consumer-group");
props.put("enable.auto.commit", "true");
props.put("session.timeout.ms", "30000");
```

Εκτός από αυτά, μπορεί αν γίνει μία σύντομη αναφορά για τις εξόδου του συστήματος. Εάν θέλουμε τα αποτελέσματα ενός stream να γράφονται σε ένα αρχείο .txt, τότε αυτό το κάνουμε ως ακολούθως.

```
stream.writeAsText("/path/to/file");
```

Εάν θέλουμε να γράψουμε σε ένα CSV αρχείο τότε:

```
stream.writeAsCsv("/path/to/file")
```

Εάν θέλουμε να περάσουμε τα αποτελέσματα σε ένα socket τότε:

```
stream.writeToSocket(host, port, SerializationSchema);
```

Παράλληλα, μπορεί να θέλουμε να γραψουμε αποτελέσματα στον Kafka. Για να το πετύχουμε αυτό, χρησιμοποιούμε τον FlinkKafkaProducer, όπου για να τον ενσωματώσομε στον κώδικα πρέπει να κάνουμε κάτι αντίστοιχο με αυτό που κάναμε προηγουμένως με τα Properties. Όταν το κάνουμε αυτό, μπορούμε να τον καλέσουμε ως εξής:

```
DataStream<String> WriteIntoKafkaStream = ...
WriteIntoKafkaStream.addSink (
                        new FlinkKafkaProducer<String> (
                                                "localhost:9092",
                                                "myTopic",
                                                new SimpleStringSchema)
                                        );
```

3.6.3 Windows

Καθώς έχουμε μιλήσει προηγουμένως για τα παράθυρα τα οποίο ομαδοποιούν δεδομένα με διάφορους τρόπους, είναι ώρα τώρα να δούμε και προγραμματιστικά πώς αυτά υλοποιούνται.

Ας υποθέσουμε ότι έρχονται συνεχώς τιμές (κάθε δύο δευτερόλεπτα) από τους αισθητήρες στο πάρκο μας και εμείς θέλουμε να βρίσκουμε την μέγιστη τιμή για κάθε ένα λεπτό. Αυτό μπορεί να γίνει με τον εξής τρόπο.

```
// (sensorID, value)
DataStream<Tuple2<Integer, Double>> sensors = ...

sensors
        .keyBy(0)                                   // group by sensorID field
        .timeWindow(Time.minutes(1), Time.second(2))
        .max("value");
```

Εκτός όμως από αυτό το παράθυρο, παρακάτω δείχνουμε πώς μπορούμε να υλοποιήσουμε και τα υπόλοιπα παράθυρα που αναφέραμε πιο πάνω.

Το Tumbling time παράθυρο:

```
.timeWindow(Time.minutes(1))
```

Το Sliding time παράθυρο:

```
.timeWindow(Time.minutes(1),Time.seconds(8))
```

Το Tumbling count παράθυρο:

```
.countWindow(1000)
```

Το Sliding count παράθυρο:

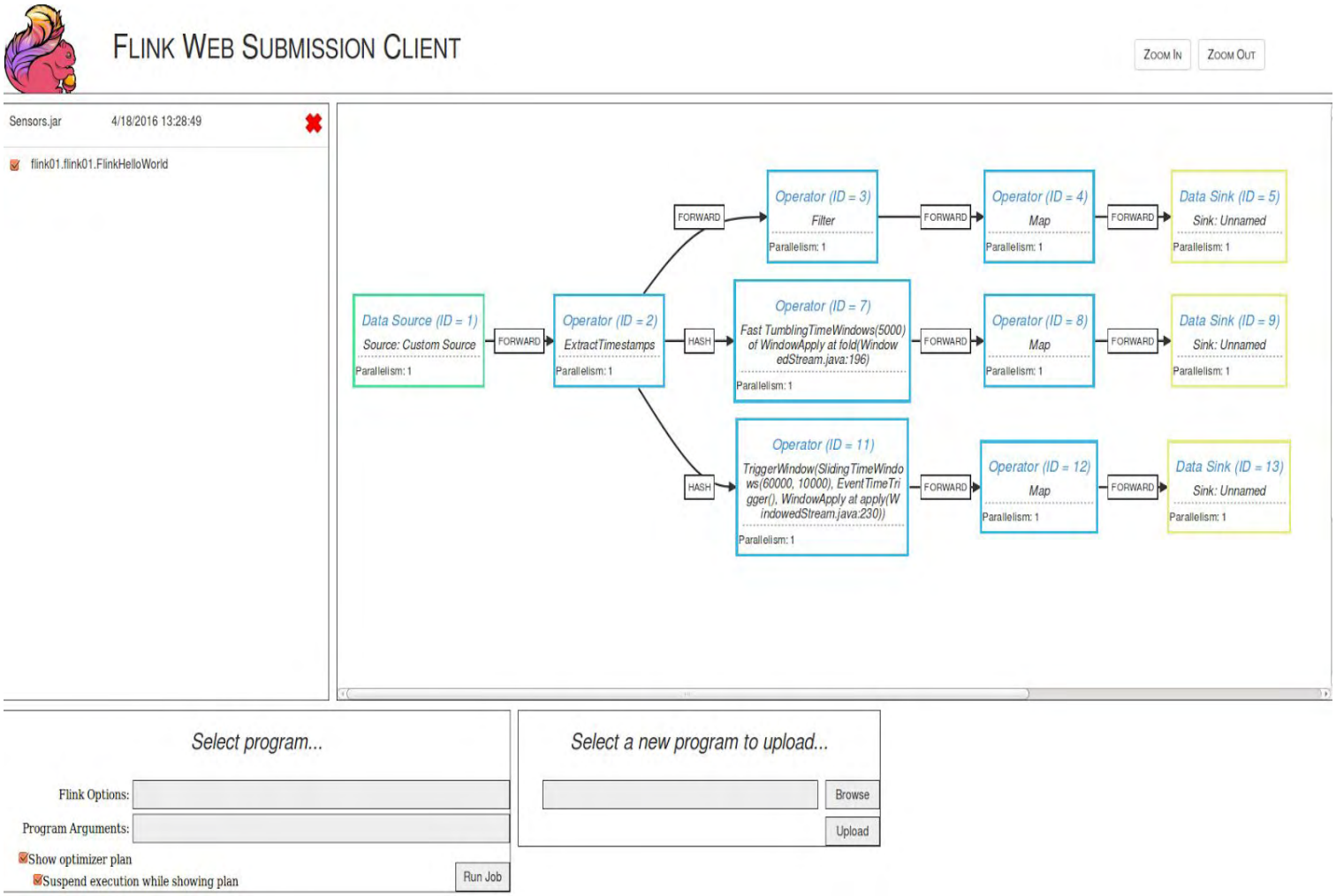
```
.countWindow(200, 20)
```

Ενδιαφέρον παρουσιάζει το γεγονός να φτιάχνουμε δικές μας συναρτήσεις για να κρατάμε στατιστικά, όπως για παράδειγμα εάν θέλουμε να κρατάμε τον μέσο της τιμής ενός αισθητήρα για κάθε λεπτό. Για να γίνει αυτό, στηριζόμαστε σε ένα timeWindow και μέσω της "*.apply()*" περνάμε τα δεδομένα όπως φαίνεται στον ακόλουθο κώδικα.

```
DataStream<SensorReading> readings = ...
readings
        .keyBy(sensorID())
        .timeWindow(Time.minutes(1), Time.seconds(2))
        .apply(new WindowFunction<SensorReading, Statistic, String, TimeWindow>()    {
                @Override
                 public void apply(String id,
                                        TimeWindow window,
                                        Iterable<SensorReading> values,
                                        Collector<Statistic> out)
                {
                                int counter = 0;
                                double agg = 0.0;
                        for (SensorReading r : values) {
                                agg += r.nextValue();
                                 counter++;
                                }

                        out.collect(new Statistic(id, window.getStart(), agg / counter));
            }
        })
        .print();
```

Τέλος, αξίζει να γίνει μια σύντομη αναφορά στο "Flink Web Submission Client", το οποίο είναι ένα εργαλείο για να μας παρέχει μια οπτική εικόνα του σκελετού του προγράμματός μας. Ένα χαρακτηριστικό στιγμιότυπο ακολουθεί παρακάτω.


FLINK WEB SUBMISSION CLIENT
ZOOM IN
ZOOM OUT
Sensors.jar 4/18/2016 13:28:49
flink01.flink01.FlinkHelloWorld
Data Source (ID = 1)
Source: Custom Source
Parallelism: 1
FORWARD
Operator (ID = 2)
ExtractTimestamps
Parallelism: 1
FORWARD
Operator (ID = 3)
Filter
Parallelism: 1
FORWARD
Operator (ID = 4)
Map
Parallelism: 1
FORWARD
Data Sink (ID = 5)
Sink: Unnamed
Parallelism: 1
HASH
Operator (ID = 7)
Fast TumblingTimeWindows(5000) of WindowApply at fold(WindowedStream.java:196)
Parallelism: 1
FORWARD
Operator (ID = 8)
Map
Parallelism: 1
FORWARD
Data Sink (ID = 9)
Sink: Unnamed
Parallelism: 1
HASH
Operator (ID = 11)
TriggerWindow(SlidingTimeWindows(60000, 10000), EventTimeTrigger(), WindowApply at apply(WindowedStream.java:230))
Parallelism: 1
FORWARD
Operator (ID = 12)
Map
Parallelism: 1
FORWARD
Data Sink (ID = 13)
Sink: Unnamed
Parallelism: 1
Select program...
Flink Options:
Program Arguments:
Show optimizer plan
Suspend execution while showing plan
Run Job
Select a new program to upload...
Browse
Upload

ΚΕΦΑΛΑΙΟ 4

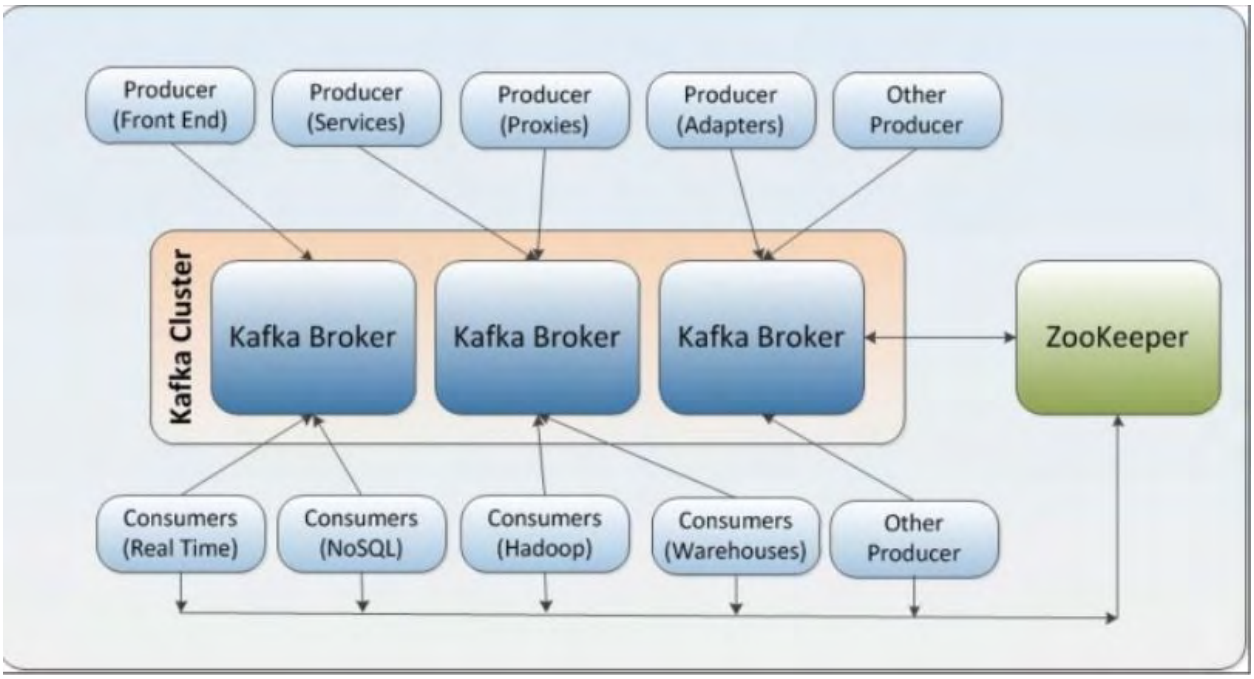
Apache Kafka

4.1 Αρχιτεκτονική

Το Apache Kafka είναι ένα σύστημα high-throughput κατανεμημένο το οποίο χρησιμοποιείται από μεγάλες εταιρείες όπως είναι η LinkedIn, Twitter, Foursquare κ.ά.. Θα επεκταθούμε στην αρχιτεκτονική του διότι στηρίζεται άμεσα με την αρχιτεκτονικήτου Samza. Για την ακρίβεια το Samza έχει χτιστεί πάνω στο Apache Kafka Project από την LinkedIn. Το Kafka στηρίζεται στην αρχιτεκτονική του publish-subscribe και έχει τα ακόλουθα χαρακτηριστικά:

- Persistent messaging, με την έννοια ότι το σύστημα είναι αξιόπιστο και δεν χάνει μηνύματα.
- High throughput, δεδομένου του ότι το Kafka μπορεί να χειρίζεται εκατοντάδες Mbs δεδομένων το δευτερόλεπτο και ότι μπορεί να γράφει σε πολλούς clients ταυτόχρονα.
- Distributed, αφού εν γένει έχει κατασκευαστεί με μία cluster-centric δομή η οποία μπορεί να επεκτείνεται δυναμικά ανάλογα με τις ανάγκες του συστήματος.
- Real time, αφού οι consumers οι οποίοι εγγράφονται στο σύστημα αυτό πρέπει να βλέπουν αμέσως τα μηνύματα τα οποία παράγονται. Αυτό είναι ένα κρίσιμο ζήτημα για συστήματα όπως τα Complex Event Processing (CEP).

Εδώ πρέπει να τονίσουμε ότι το Kafka είναι αλληλένδετο με το zookeeper το οποίο τρέχει μαζί του για να κρατάει τις "συντεταγμένες" του συστήματος. Μια χαρακτηριστική εικόνα που περιγράφει το σύστημά μας είναι η ακόλουθη.

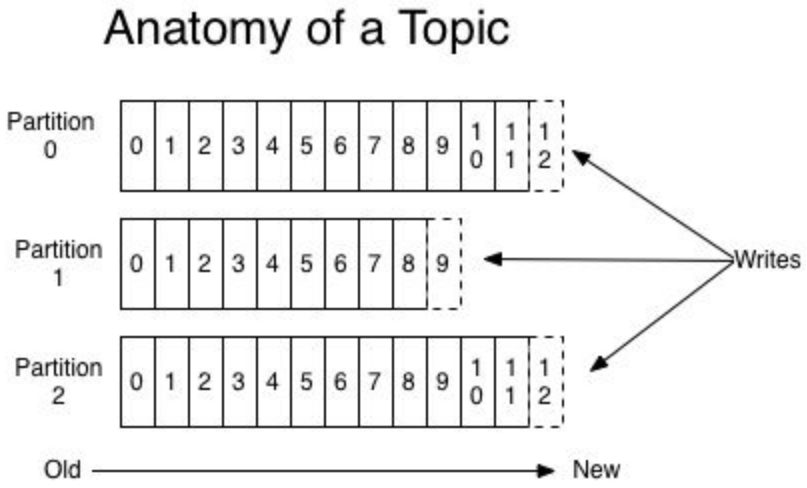


Το Kafka μπορούμε να το χρησιμοποιήσουμε με διάφορους τρόπους. Τέτοιοι είναι το Log Aggregation, το Commit Log, το Messaging, αλλά και το Stream processing. Εμάς μας ενδιαφέρει στην παρούσα διπλωματική το Stream processing για 'υτό και θα επεκτεθούμε σε αυτό.

Όσον αφορά το stream processing, το Kafka συλλέγει δεδομένα από διαφορετικά topics, τα στέλνει σε μια μηχανή επεξεργασίας όπως είναι το Flink ή το Samza για να αναλυθούν τα ακατέργαστα δεδομένα και έπειτα τα παίρνει πάλι ο Kafka και τα εναπωθέτει στα σωστά topics.

Ένα topic αποτελείται από πολλά partitions, τα οποία μέσα τους έχουν προφανώς τα δεδομένα. Ουσιαστικά ένα topic είναι μια κατηγορία των δεδομένων τα οποία περιέχει. Τα partitions είναι υποσύνολα δεδομένων τα οποία είναι ουσιαστικά ο

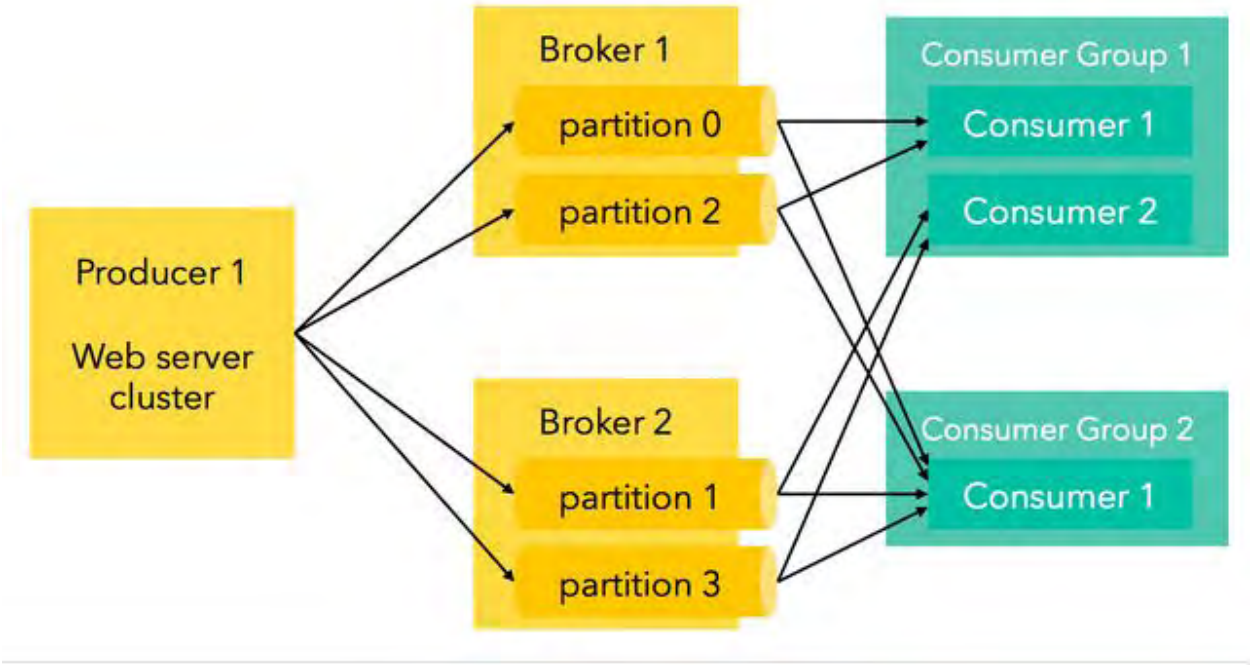
βαθμός παραλληλισμού. Χαρακτηριστική είναι η παρακάτω εικόνα η οποία δείχνει την ανατομία ενός topic.



Οι υπηρεσίες οι οποίες τα γεμίζουν αυτά με δεδομένα ονομάζονται producers και οι υπηρεσίες οι οποίες διαβάζουν τα δεδομένα αυτά ονομάζονται consumers. Η επικοινωνία των consumers με τους producers πραγματοποιείται κάτω από το TCP πρωτόκολλο. Το Kafka τοποθετεί σε κάθε μήνυμα και ένα μοναδικό id το οποίο καλείται message-offset το οποίο αναπαριστά την μοναδικότητα και αυξάνει την χρονοσφραγίδα του μηνύματος αυτού.

Ένα άλλο χαρακτηριστικό που αξίζει να σημεωθεί είναι ότι μπορούμε να ενεργοποιήσουμε πολλαπλούς consumers οι οποίοι θα βρίσκονται μέσα σε ένα consumer group. Το σύστημα τότε υποχρεούται να μας δώσει την εγγύηση ότι το μύνημα που θέλουμε θα επεξεργαστεί τουλαχιστον μια φορά (at least once).

Στην παρακάτω εικόνα έχουμε έναν producer (ένα cluster από web servers), ο οποίος στέλνει τα μηνύματα μέσα σε ένα topic με 4 partitions. Αυτά τα 4 partitions τα διαχειρίζονται 2 brokers. Αυτά τα 4 partitions τα βλέπουν και οι 4 consumers οι οποίοι τα καταναλώνουν κιολας.



Πολύ σημαντικό θέμα στον Kafka είναι αυτό της εγγύησης που προσφέρει στα μηνύματα. Μας δίνει λοιπόν 3 βασικές εγγυήσεις.

- Τα μηνύματα που παράγονται από τους producers, σε ένα topic, θα παραδοθούν στους consumers με την σειρά με την οποία δημιουργήθηκαν.
- Μας δίνει την εγγύηση ότι το μήνυμα που παράγεται θα παραδοθεί τουλάχιστον μια φορά σε έναν consumer. (at least once)
- Για ένα topic με συντελεστή αντιγραφής N (replication factor), θα υπάρχει η ανοχή για N-1 αποτυχίες-βλάβες των servers χωρίς να χάσουμε ούτε ένα μήνυμα.

4.2 Υλοποίηση

Εφόσων το Samza είναι βασισμένο πάνω στην αρχιτεκτονική του Kafka (on top of Kafka), θα παρουσιάσουμε ένα βασικό πραγματικό προγραμματιστικό μοντέλο.

Πρώτα πρέπει να κατεβάσουμε την τελευταία έκδοση (0.9.0.1).

tar -xzf kafka_2.11-0.9.0.0.tgz

```
ioakim@noah-lt:~/Downloads$ tar -xzf kafka_2.11-0.9.0.1.tgz
```

Μόλις τελειώσει αυτή η διαδικασία πρέπει να μπούμε στον φάκελο και να ξεκινήσουμε τον zookeper server μας ο οποίος βρίσκεται στον φάκελο “bin”. Ο zookeeper server πρέπει να ξεκινήσει πρώτος για να φτιάξει στις συντεταγμένεστου συστήματος. Τον ξεκινάμε έτσι:

bin/zookeeper-server-start.sh config/zookeeper.properties

Το **config/zookeeper.properties είναι** το configuration file το οποίο του το δίνουμε στο startup. Αν θέλουμε να τρέχει σε ένα cluster το διαμορφώνουμε αναλόγως.

Αφού ξεκινήσει θα πρέπει να δούμε μια εικόνα σαν την ακόλουθη.

```
/kafka_2.11-0.9.0.1/bin/../libs/aopalliance-repackaged-2.4.0-b31.jar (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:java.library.path=/usr/java/packages/lib/amd64:/usr/lib/x86_64-linux-gnu/jni:/lib/x86_64-linux-gnu:/usr
rg.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:java.io.tmpdir=/tmp (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:java.compiler=<NA> (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:os.name=Linux (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:os.arch=amd64 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:os.version=4.2.0-34-generic (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:user.name=ioakim (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:user.home=/home/ioakim (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,589] INFO Server environment:user.dir=/home/ioakim/Downloads/kafka/kafka_2.11-0.9.0.1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,601] INFO tickTime set to 3000 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,601] INFO minSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,601] INFO maxSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-04-01 11:40:12,638] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxnFactory)
```

Αμέσως μετά, πρέπει να ξεκινήσουμε τον kafka server μας. Το κάνουμε αυτό με την ακόλουθη εντολή:

bin/kafka-server-start.sh config/server.properties

Όταν την τρέξουμε αυτήν την εντολή θα ξεκινήσει και ο kafka, όπως φαίνεται και στο παρακάτω terminal.

```
2016-04-01 11:44:56,574] INFO Logs loading complete. (kafka.log.LogManager)
2016-04-01 11:44:56,624] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogManager)
2016-04-01 11:44:56,626] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogManager)
2016-04-01 11:44:56,632] WARN No meta.properties file under dir /tmp/kafka-logs/meta.properties (kafka.server.BrokerMetadataCheckpoint)
2016-04-01 11:44:56,737] INFO Awaiting socket connections on 0.0.0.0:9092. (kafka.network.Acceptor)
2016-04-01 11:44:56,741] INFO [Socket Server on Broker 0], Started 1 acceptor threads (kafka.network.SocketServer)
2016-04-01 11:44:56,807] INFO [ExpirationReaper-0], Starting  (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
2016-04-01 11:44:56,808] INFO [ExpirationReaper-0], Starting  (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
2016-04-01 11:44:56,941] INFO Creating /controller (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
2016-04-01 11:44:56,986] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
2016-04-01 11:44:56,987] INFO 0 successfully elected as leader (kafka.server.ZookeeperLeaderElector)
2016-04-01 11:44:57,177] INFO [GroupCoordinator 0]: Starting up. (kafka.coordinator.GroupCoordinator)
2016-04-01 11:44:57,180] INFO [ExpirationReaper-0], Starting  (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
2016-04-01 11:44:57,181] INFO [GroupCoordinator 0]: Startup complete. (kafka.coordinator.GroupCoordinator)
2016-04-01 11:44:57,183] INFO [ExpirationReaper-0], Starting  (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
2016-04-01 11:44:57,193] INFO [Group Metadata Manager on Broker 0]: Removed 0 expired offsets in 15 milliseconds. (kafka.coordinator.GroupMetadataManager)
2016-04-01 11:44:57,213] INFO New leader is 0 (kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
2016-04-01 11:44:57,266] INFO [ThrottledRequestReaper-Produce], Starting  (kafka.server.ClientQuotaManager$ThrottledRequestReaper)
2016-04-01 11:44:57,267] INFO [ThrottledRequestReaper-Fetch], Starting  (kafka.server.ClientQuotaManager$ThrottledRequestReaper)
2016-04-01 11:44:57,280] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader$)
2016-04-01 11:44:57,298] INFO Creating /brokers/ids/0 (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
2016-04-01 11:44:57,326] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
2016-04-01 11:44:57,348] INFO Registered broker 0 at path /brokers/ids/0 with addresses: PLAINTEXT -> EndPoint(noah-lt,9092,PLAINTEXT) (kafka.utils.ZkUtils)
2016-04-01 11:44:57,380] INFO Kafka version : 0.9.0.1 (org.apache.kafka.common.utils.AppInfoParser)
2016-04-01 11:44:57,380] INFO Kafka commitId : 23c69d62a0cabf06 (org.apache.kafka.common.utils.AppInfoParser)
2016-04-01 11:44:57,381] INFO [Kafka Server 0], started (kafka.server.KafkaServer
```

Το αρχείο που δίνουμε ως input “**server.properties”** είναι πολύ σημαντικό όταν τρέχουμε τον Kafka στο cloud. Για να κάνουμε ένα configuration στο σύστημά μας και να τον τρέχουμε κατανεμημένα πρέπει να κάνουμε το εξής:

vim server.properties

και θα μας εμφανιστεί το αρχείο.

```
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=0

############################# Socket Server Settings #############################

listeners=PLAINTEXT://:9092

# The port the socket server listens on
#port=9092

# Hostname the broker will bind to. If not set, the server will bind to all interfaces
#host.name=localhost

# Hostname the broker will advertise to producers and consumers. If not set, it uses the
# value for "host.name" if configured.  Otherwise, it will use the value returned from
# java.net.InetAddress.getCanonicalHostName().
#advertised.host.name=<hostname routable by clients>

# The port to publish to ZooKeeper for clients to use. If this is not set,
# it will publish the same port that the broker binds to.
#advertised.port=<port accessible by clients>
```

Στο αρχείο αυτό θα πρέπει να ορίσουμε μοναδικά broker.id για κάθε κόμβο και να ορίσουμε σωστά τις τιμές host.name, advertised.host.name και advertised.host.port.

Στην συγκεκριμένη περίπτωση θα το αφήσουμε όπως έχει για να τρέξουμε ένα απλό παράδειγμα localhost.

Αμέσως μετά από αυτά, θα δημιουργήσουμε το πρώτο μας topic με την ακόλουθη εντολή.

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

Για να δούμε ποια topics υπάρχουν στο σύστημά μας το κάνουμε με την εντολή:

bin/kafka-topics.sh --list --zookeeper localhost:2181

Τέλος, το πακέτο το οποίο κατεβάσαμε προσφέρει έτοιμο producer και consumer, οι οποίοι δεν κάνουν τίποτα άλλο από το στέλνουν και να εμφανίζουν ένα μήνυμα.

Ξεκινάμε τον παραγωγό μας:

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

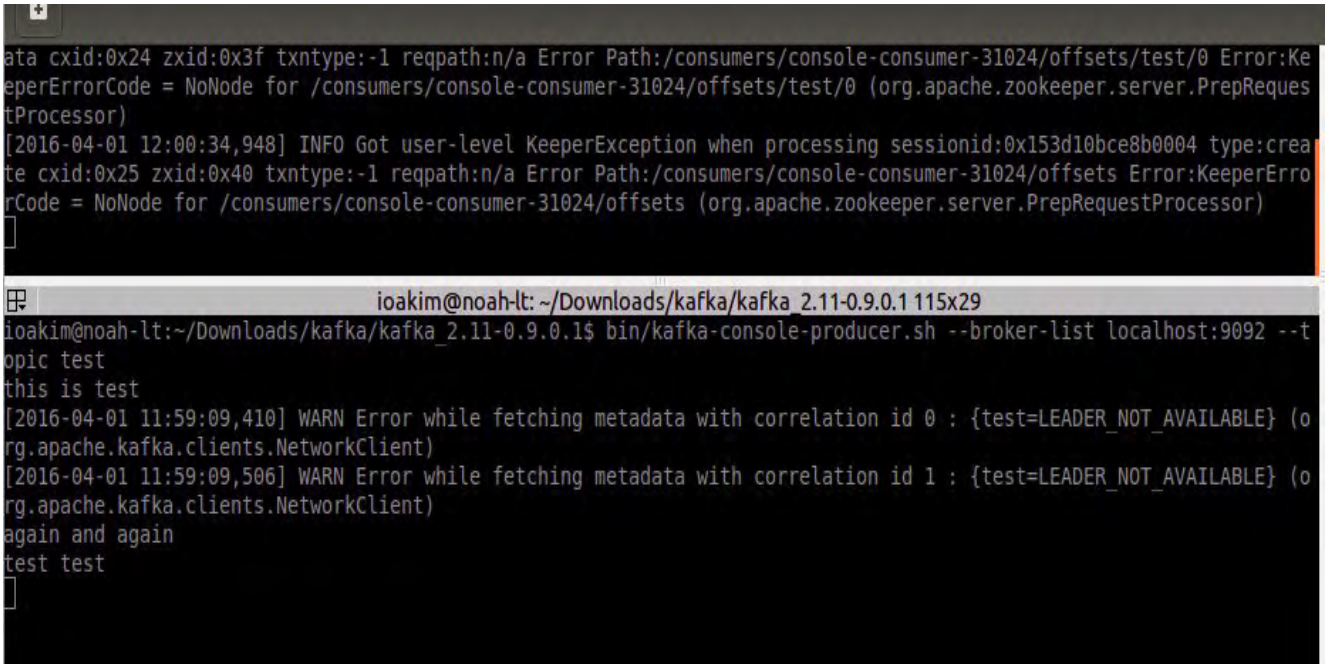
Στέλνουμε κάποια μηνύματα χωρίς να έχουμε ξεκινήσει κανέναν consumer.

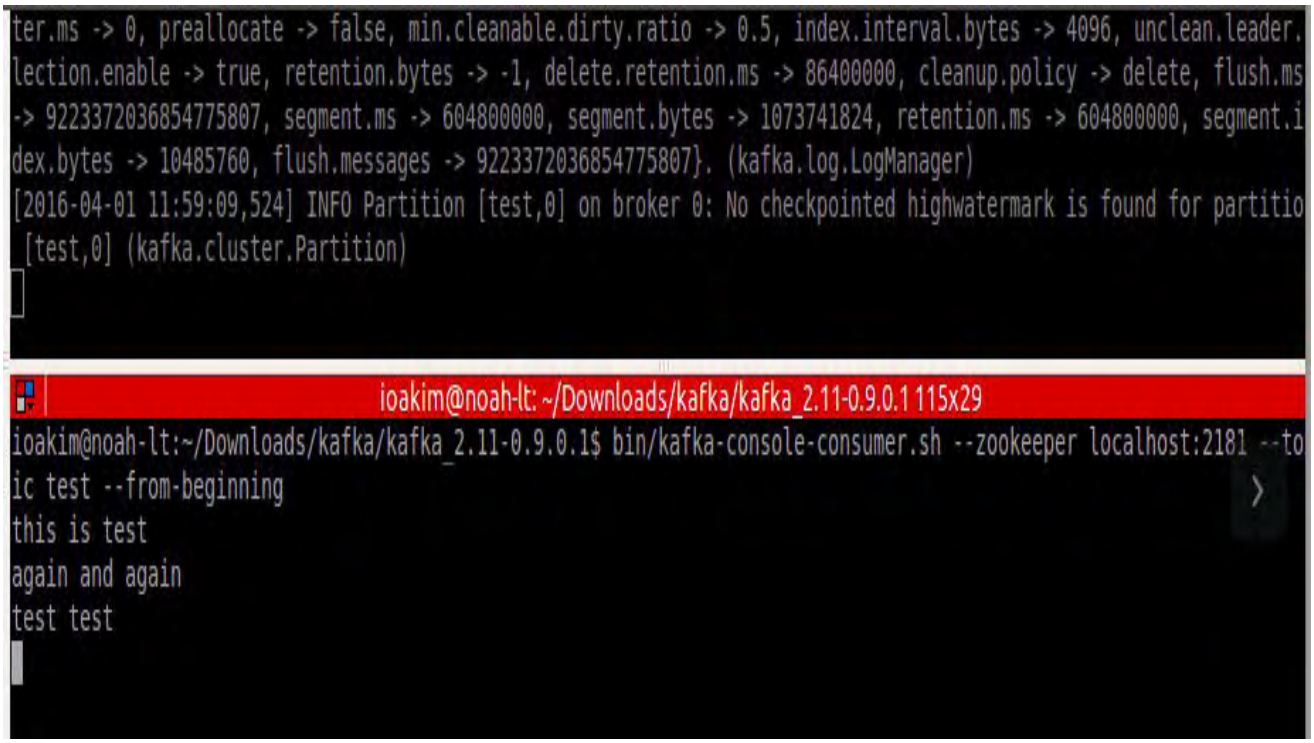
Έπειτα ξεκινάμε έναν consumer με την εντολή:

bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

και παρατηρούμε ότι με το που ξεκινήσει διαβάζει τα μηνύματα τα οποία έχουν σταλεί στο topic πριν δημιουργηθεί. Εδώ φαίνεται η εγγύηση ότι δεν χάνονται μηνύματα στο σύστημα.

Το terminal μας λοιπόν μοιάζει κάπως έτσι.

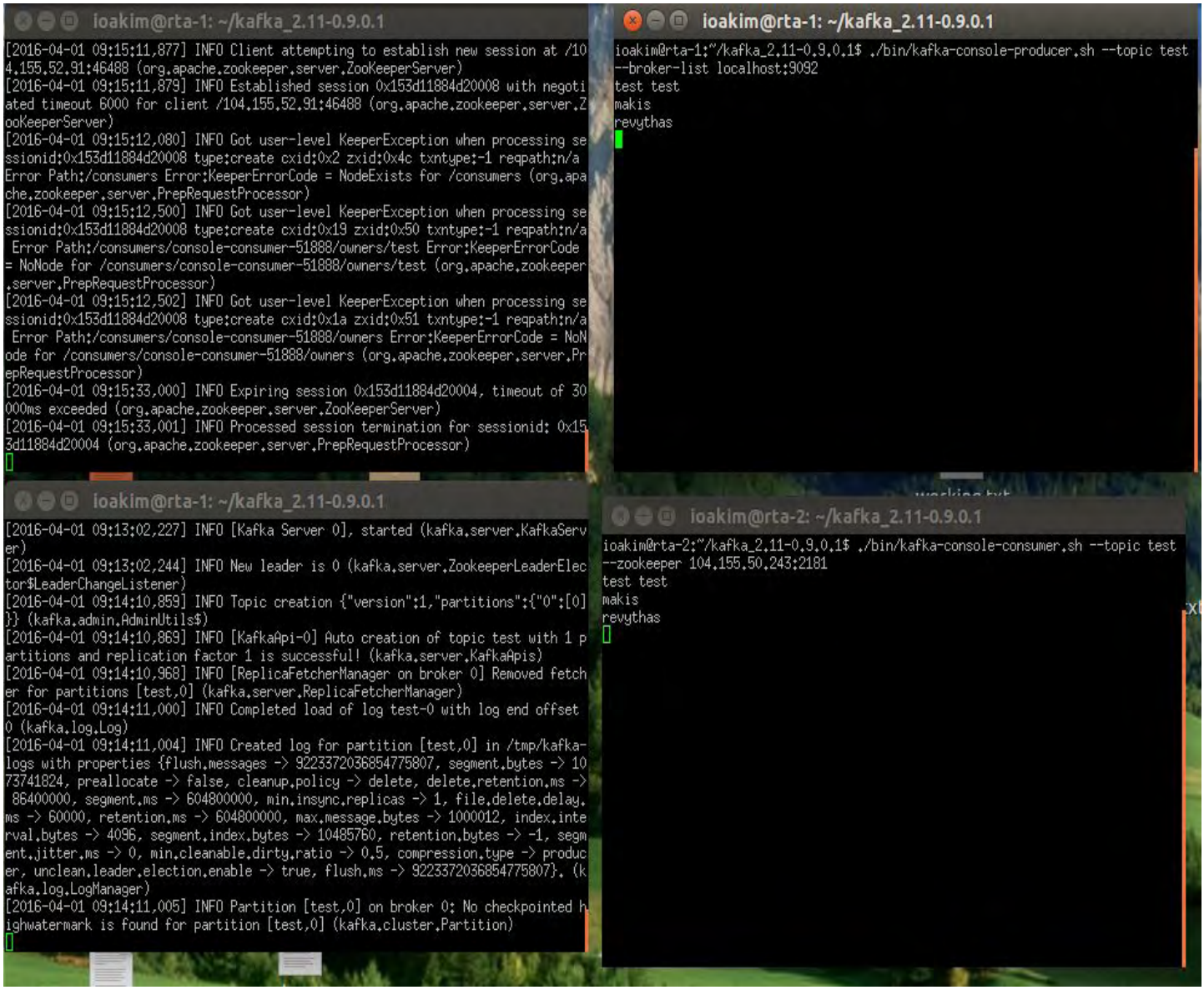

ata cxid:0x24 zxid:0x3f txntype:-1 reqpath:n/a Error Path:/consumers/console-consumer-31024/offsets/test/0 Error:Ke
eperErrorCode = NoNode for /consumers/console-consumer-31024/offsets/test/0 (org.apache.zookeeper.server.PrepReques
tProcessor)
[2016-04-01 12:00:34,948] INFO Got user-level KeeperException when processing sessionid:0x153d10bce8b0004 type:crea
te cxid:0x25 zxid:0x40 txntype:-1 reqpath:n/a Error Path:/consumers/console-consumer-31024/offsets Error:KeeperErro
rCode = NoNode for /consumers/console-consumer-31024/offsets (org.apache.zookeeper.server.PrepRequestProcessor)
ioakim@noah-lt: ~/Downloads/kafka/kafka_2.11-0.9.0.1 115x29
ioakim@noah-lt:~/Downloads/kafka/kafka_2.11-0.9.0.1$ bin/kafka-console-producer.sh --broker-list localhost:9092 --t
opic test
this is test
[2016-04-01 11:59:09,410] WARN Error while fetching metadata with correlation id 0 : {test=LEADER_NOT_AVAILABLE} (o
rg.apache.kafka.clients.NetworkClient)
[2016-04-01 11:59:09,506] WARN Error while fetching metadata with correlation id 1 : {test=LEADER_NOT_AVAILABLE} (o
rg.apache.kafka.clients.NetworkClient)
again and again
test test


ter.ms -> 0, preallocate -> false, min.cleanable.dirty.ratio -> 0.5, index.interval.bytes -> 4096, unclean.leader.
lection.enable -> true, retention.bytes -> -1, delete.retention.ms -> 86400000, cleanup.policy -> delete, flush.ms
-> 9223372036854775807, segment.ms -> 604800000, segment.bytes -> 1073741824, retention.ms -> 604800000, segment.i
dex.bytes -> 10485760, flush.messages -> 9223372036854775807}. (kafka.log.LogManager)
[2016-04-01 11:59:09,524] INFO Partition [test,0] on broker 0: No checkpointed highwatermark is found for partitio
[test,0] (kafka.cluster.Partition)
ioakim@noah-lt: ~/Downloads/kafka/kafka_2.11-0.9.0.1 115x29
ioakim@noah-lt:~/Downloads/kafka/kafka_2.11-0.9.0.1$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --to
ic test --from-beginning
this is test
again and again
test test

Το δεύτερο και πιο ενδιαφέρον μέρος του πειράματός μας με τον Kafka είναι να έχουμε 2 VMs, στις διευθύνσεις:

104.155.50.243 και 104.155.52.91

και να στέλνουμε μηνύματα από το ένα VM στο άλλο. Για να γίνει αυτό, αρκεί να ξεκινήσουμε στο 104.155.50.243 τον zookeeper και kafka server και έναν producer. Στο 104.155.52.91 VM θα έχουμε έναν consumer ο οποίος καταναλώνει τα μηνύματα όπως φαίνεται στην παρακάτω εικόνα.



4.3 Υλοποίηση ενός Kafka Consumer

Ακολουθεί ο κώδικας ενός λειτουργικού kafka consumer ο οποίος διαβάζει από ένας topic, και το προβάλει στο τερματικό.

```
import java.util.Arrays;
import java.util.Properties;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

public class MyConsumer {
        private final String topic;

            public MyConsumer(String zookeeper, String groupId, String topic) {
               this.topic = topic;
            }

            public void testConsumer()    {
               Properties props = new Properties();
               props.put("bootstrap.servers", "104.155.50.243:2181");
               props.put("group.id", "test");
               props.put("enable.auto.commit", "true");
               props.put("auto.commit.interval.ms", "1000");
               props.put("session.timeout.ms", "30000");
               props.put("key.deserializer",
                                  "org.apache.kafka.common.serialization.StringDeserializer");
               props.put("value.deserializer",
                                  "org.apache.kafka.common.serialization.StringDeserializer");
               KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
               consumer.subscribe(Arrays.asList(topic));

               while (true) {
                  ConsumerRecords<String, String> records = consumer.poll(100);
                  for (ConsumerRecord<String, String> record : records)
                     System.out.printf("offset = %d, key = %s, value = %s", record.offset(),
                                                                  record.key(), record.value());
               }

            }

            public static void main(String[] args) {
               String topic = "test";
               MyConsumer simpleHLConsumer = new MyConsumer("104.155.50.243:2181",
```

```
                                                    test-consumer-group", topic);
            simpleHLConsumer.testConsumer();

        }
}
```

4.4 Υλοποίηση ενός Kafka Producer

Ακολουθεί κώδικας, ενός kafka producer ο οποίος διαβάζει ένα αρχείο json και το αποθηκεύει σε ένα topic.

```
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Properties;
import java.util.Scanner;

public class SimpleProducer {
   private static Producer<Integer, String> producer;
   private final Properties properties = new Properties();
   private static Scanner scanner;

   public SimpleProducer() {
      properties.put("metadata.broker.list", "104.155.52.91:9092");
      properties.put("serializer.class", "kafka.serializer.StringEncoder");
      properties.put("request.required.acks", "1");
      producer = new Producer<>(new ProducerConfig(properties));
   }

   public static void main(String[] args) throws FileNotFoundException {
      new SimpleProducer();
      String topic = "test";
      String msg = null;
      File text = new File("/home/ioakim/Downloads/data-single.json");

      //Creating Scanner instnace to read File in Java
      Scanner scnr = new Scanner(text);
      StringBuilder  stringBuilder = new StringBuilder();

      while(scnr.hasNextLine()){
         String line = scnr.nextLine();
```

```
        //   stringBuilder.append( line );
           KeyedMessage<Integer, String> data = new KeyedMessage<>(topic,
                                                                                    line);

           producer.send(data);

        }
        producer.close();
           }
}
```

ΚΕΦΑΛΑΙΟ 5

Apache Samza

5.1 Γενικά για το Samza

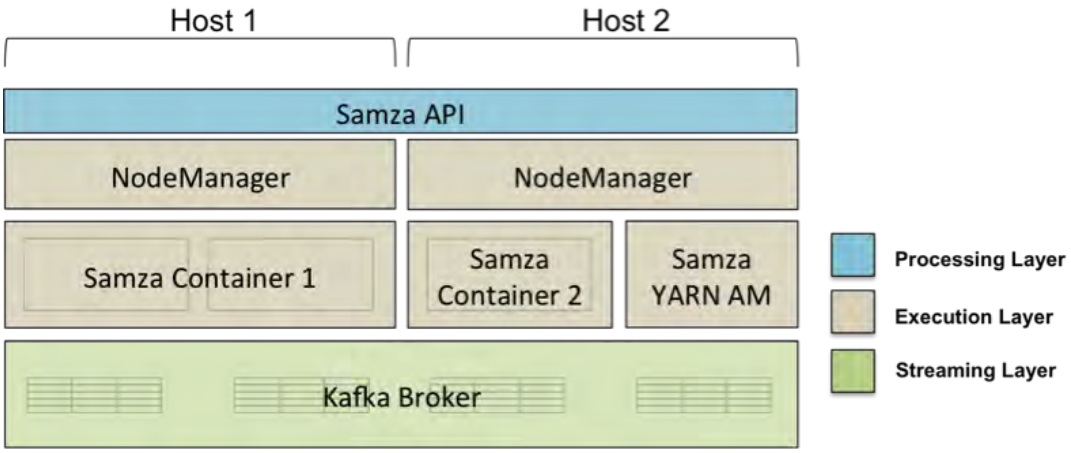
Όπως αναφέραμε και προηγουμένως το Samza στηρίζεται αρχιτεκτονικά στον Kafka. Στόχος του Samza είναι να παρέχει ένα ελαφρύ Framework για συνεχή επεξεργασία δεδομένων. Το Samza είναι ένα πολύ χρήσιμο εργαλείο για να κάνουμε update μια βάση δεδομένων, να υπολογίζουμε μετρητές ή γενικότερα για να επεξεργαζόμαστε μηνύματα. Ένα χαρακτηριστικό παράδειγμα ενός Samza Job του LinkedIn είναι η επεξεργασία πάνω από 1.000.000 μηνύματα το δευτερόλεπτο. Κάποια βασικά χαρακτηριστικά του Samza είναι ότι έχει πολύ απλό API, είναι ανεκτικό σε βλάβες και μας δίνει εγγυήσεις ότι το μήνυμα που θα επεξεργαστεί θα αποθηκευτεί σε ένα topic.

5.2 Αρχιτεκτονική του Samza

Η αρχιτεκτονική του Samza αποτελείται από 3 βασικά μέρη:

- Streaming layer, το οποίο είναι υπεύθυνο για να παρέχει διαρκή και τμηματοποιημένα steams (Kafka)
- excecution layer, το οποίο είναι υπεύθυνο για την χρονοδρομολόγιση των tasks στις συστοιχίες του συστήματος (YARN)
- processing layer, το οποίο είναι υπεύθυνο για το input stream (Samza API)

Η παραπάνω περιγραφή της αρχιτεκτονικής παρουσιάζεται σχηματικά στο ακόλουθο σχήμα:



Ο SamzaContainer, είναι υπεύθυνος για την διαχείριση των StreamTasks. Ουσιαστικά είναι αυτός ο οποίος ελέγχει τα checkpoints, τις μετρικές και γενικά διαχειρίζεται την εκτέλεση του task. Ένας SamzaContainer τρέχει ως ανεξάρτητη εικονική μηχανή (VM) JAVA. Παρ' όλα αυτά, ένα Samza Job μπορεί να βρίσκεται σε διαφορετικούς SamzaContainers, άρα και ταυτόχρονα σε διαφορετικά μηχανήματα.

Από την άλλη πλευρά, υπάρχει ο NodeManager σε υψηλότερο επίπεδο, ο οποίος βρίσκεται μέσα στην YARN αρχιτεκτονική, και είναι αυτός ο οποίος ξεκινάει την εκτέλεση των διεργασιών. Σε ακόμα πιο υψηλό επίπεδο, υπάρχει ο ResourceManager ο οποίος μιλάει με όλους τους NodeManagers, και τους λέει ποιά διεργασία πρέπει να ξεκινήσουν να τρέξουν.

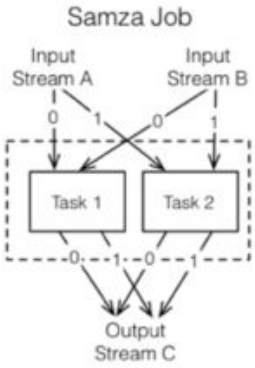
5.2.1 Streams and Jobs

Ένα από τα πιο σημαντικά μέρη του συστήματος είναι τα Streams και τα Jobs. Αυτά τα 2 components, είναι τα 2 βασικά μέρη για να χτίσουμε ένα Samza application.

Ένα stream αποτελείται από αμετάβλητες ακολουθίες παρόμοιων μηνυμάτων. Για να πετύχουμε scalability στο σύστημά μας πρέπει να σπάσουμε τα streams που έχουμε ως είσοδο σε ανάλογα partitions.

Ένα Job, είναι ο κώδικας ο οποίος θα καταναλώσει ή θα παράξει ένα stream. Για να πετύχουμε scalability στο throughput, τα jobs πρέπει να σπάσουν σε μικρότερα units τα οποία ονομάζονται tasks. Το κάθε task καταναλώνει δεδομένα από ένα ή περισσότερα partitions για κάθε input stream.

Στην παρακάτω εικόνα φαίνεται η σχηματική απεικόνιση των όσων περιγράψαμε.



5.3 Βασικό παράδειγμα κατανόησης

Για να καταλάβουμε την ακριβής λειτουργία θα δώσουμε ένα πραγματικό παράδειγμα. Ας υποθέσουμε ότι θέλουμε να δούμε πόσο ποιοτικά δεδομένα στέλνει ένας αισθητήρας από ένα φωτοβολταϊκό πάρκο. Για να το κάνουμε αυτό πρέπει να μετρήσουμε από τις τιμές που μας φέρνει ο αισθητήρας κάθε στιγμή πόσες από αυτές ανταποκρίνονται στις πιθανές πραγματικές και πόσες είναι σφάλμα μέτρησης.

Για να πραγματοποιηθεί αυτό έχουμε 2 Jobs. Το ένα δέχεται τα inputs από τον αισθητήρα, ελέγχει με κάποια κριτήρια εάν είναι αποδεκτές τιμές και τα τοποθετεί στο ανάλογο topic. Από την άλλη, το δεύτερο Job, διβάζει από το topic το οποίο έχει τις αποδεκτές τιμές και για κάθε νέα τιμή που γράφεται στο topic αυτό αυξάνει έναν μετρητή. Από αυτόν τον μετρητή ξέρουμε κάθε στιγμή την ευστοχία του αισθητήρα μας.

Αυτή η διαδικασία έχει ομοιότητες με την λειτουργία Map/Reduce του Hadoop. Η κύρια διαφορά όμως με το Hadoop είναι ότι στο Hadoop πρέπει να έχουμε καθορισμένη είσοδο (fix inputs), σε αντίθεση με το Samza που μπορεί να έχει ακαθόριστες εισόδους.

5.4 Προγραμματιστική υλοποίηση

Για να υλοποιήσουμε μια Samza εφαρμογή πρέπει καταρχάς να υλοποιήσουμε ένα Job. Ένα Samza Job αποτελείται από τον βασικό κώδικα που δίνει την λειτουργικότητα (actual code) το οποίο είναι ένα task και από ένα configuration αρχείο. Αυτά τα 2 πρέπει να γίνουν μαζί deploy στον server για να τρέξουμε την εφαρμογή μας.

Για να δημιουργήσουμε ένα task, πρέπει να υλοποιήσουμε το StreamTask interface.

```
package com.samza;

public class MyTask implements StreamTask {

        private static final SystemStream OUTPUT_STREAM       =
                                                        new SystemStream("kafka","sensorOut");
        public void process (IncomingMessageEnvelope envelope,
                                MessageCollector collector,
                                TaskCoordinator coordinator)
        throws Exception {
                // Do something useful
        }
}
```

Αυτό είναι ένα έγκυρο task το οποίο όμως δεν έχει κάποια λειτουργικότητα. Μπορούμε να προσθέσουμε μια λειτουργικότητα και μαζί με το ακόλουθο configuration file να το κάνουμε deploy και να τρέξει.

```
# Εδώ ορίζουμε την κλάση την οποία θα φορτώσει για να τρέξει το πρόγραμμα
task.class=com.samza.MyTask
```

```
# Εδώ ορίζουμε ένα σύστημα kafka στο οποίο θα είναι η πηγή του συστήματος
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory

# Εδώ ορίζουε ότι το MyTask θα λαμβάνει το input από το σύστημα kafka και πιο
# συγκεκριμένα από το topic MyTaskTopic
task.inputs=kafka.MyTaskTopic

#Εδώ βάζουμε προτεραιότητα στην είσοδο. Μπορούμε να έχουμε #παραπάνω από
# μία εισόδους. Η σύνταξη γίνεται ως εξής:
 systems.<system>.streams.<stream>.samza.priority=<number>
systems.kafka.streams.myRealTimeTopic.samza.priority=1
systems.kafka.streams.myBatchTopic.samza.priority=2


# Εδώ ορίζουμε έναν serializer/deserializer ο οποίος λέγεται #"json" και κάνει
parse
# JSON μηνύματα
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory

# Χρησιμοποιεί "json" σειριοποίηση για μηνύματα στο “MyTaskTopic” topic
systems.kafka.streams.MyTaskTopic.samza.msg.serde=json
```

Τώρα θα υλοποιήσουμε μια βασική λειτουργικότητα του MyTask.

```
public class MyTask implements StreamTask {

        private static final SystemStream OUTPUT_STREAM      =

                                                    new SystemStream("kafka","sensorOut");

        public void process (IncomingMessageEnvelope envelope,
                        MessageCollector collector,
                        TaskCoordinator coordinator) throws Exception {
```

```
                try {
                    @SuppressWarnings("unchecked")
                    Map<String,Object> json = (Map<String,Object>)envelope.getMessage();
                    String value=null;
                    value = (String)json.get("sensosValue");
                    value = value.concat("__Sensor_working");
                    HashMap<String,Object> val = new HashMap<String,Object>();
                    val.put("output:", value);
                    collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM,val));
                } catch(Exception e) {
                        System.err.println(e);
                }
        }
}
```

Το συγκεκριμένο Task κάνει μια βασική λειτουργία. Διαβάζει από ένα stream, το οποίο μπορεί να είναι το **MyTaskTopic** του kafka το οποίο έχει οριστεί σαν input στο configuration file (.properties), την τιμή ενός αισθητήρα και κάνει μια συμβολική πράξη. Αφού κάνει την ανάλογη πράξη το στέλνει αμέσως στο **sensorOut** topic του kafka. Αξίζει να σημειωθεί ότι στο topic το οποίο διαβάζει το task μας βρίσκονται αρχεία της μορφής .json.

5.5 Metrics

Όταν τρέχουμε μία streaming διαδικασία, είναι σημαντικό να έχουμε καλές μετρικές για το Job που τρέχουμε. Το Samza μας προσφέρει τέτοιες βιβλιοθήκες για να μπορούμε να κάνουμε κάποια βασικά πράγματα εύκολα και γρήγορα, όπως είναι το throughput των μηνυμάτων. Εκτός από τις έτοιμες μετρικές όμως, μπορούμε να δημιουργήσουμε και δικές μας εύκολα.

Οι μετρικές στο πρόγραμμά μας μπορούν να γίνουν report με διαφόρους τρόπους. Μπορούμε αν τις εκθέσουμε μέσω του JMX της Oracle, το οποίο είναι και ο default τρόπος. Για να το κάνουμε αυτό πρέπει να προσθέσουμε στο configuration file το εξής:

 # Ορίζουμε έναν metrics reporter τον οποίο τον ονομάζουμε "jmx", ο οποίος
 # δημοσιεύει στο JMX
metrics.reporter.jmx.class=org.apache.samza.metrics.reporter.JmxReporterFactory
metrics.reporters=jmx

Ένας κλασσικός τρόπος για να εκθέτουμε τις μετρικές μας είναι να κάνουμε publish τις μετρικές μας περιοδικά σε ένα kafka topic, από το οποίο θα τις κάνουμε consume μέσω ενός samza job, όπου μέω αυτύ θα τις στέλνουμε σε ένα γραφικό σύστηα όπως είναι το Graphite.

Για να πραγματοποιηθεί αυτή η κλασσική διαδικασία πρέπει πρώτα να βάλουμε τα σωστά properties στο αντίστοιχο configuration file. Αυτά είναι τα ακόλουθα:

 # Ορίζουμε έναν metrics reporter τον οποίο τον ονομάζουμε "snapshot" ο οποίος
 # είναι υπεύθυνος για να δημοσιεύει περιοδικά (κάθε 60 sec.) τις μετρικές.

metrics.reporter=snapshot
metrics.reporter.snapshot.class=org.apache.samza.metrics.reporter.MetricsSnapshotReporterFactory

 # Λέμε στον "snapshot" reporter να δημοσιεύει σε ένα topic "metrics" στο σύστημα
 # "kafka"
metrics.reporter.snapshot.stream=kafka.metrics

 # Κωδικοποιεί τα δεδομένα του metrics reporter ως JSON.
serializers.registry.metrics.class=org.apache.samza.serializers.MetricsSnapshotSerdeFactory
systems.kafka.streams.metrics.samza.msg.serde=metrics

Εάν τώρα θέλαμε να φτιάξουμε το δικό μας metric σύστημα, θα μπορουσαμε να το κάνουμε αυτό μεσω του MetricsRegistry. Το task που θα δημιουργούσαμε θα έπρεπε να κάνει implement το InitialTask ώστε να μπορεί να πάρει μετρικές από το TaskContext. Ένα τέτοιο βασικό παράδειγμα θα ήταν το ακόλουθο:

```
public class MyJavaStreamTask implements StreamTask, InitableTask {

    private Counter messageCount;
    public void init(Config config, TaskContext context) {
        this.messageCount = context
                .getMetricsRegistry()
                .newCounter(getClass().getName(),
                        "message-count");

    }

    public void process(IncomingMessageEnvelope envelope,
                        MessageCollector collector,
                        TaskCoordinator coordinator) {
        messageCount.inc();
```

```
        }
}
```

Αυτό το απλό παράδειγμα μας δείχνει πως υπολογίζουμε έναν μετρητή ο οποίος αριθμεί τα μηνύματα που επεξεργαζόμαστε σε αυτό το Task. Το Samza υποστηρίζει μέχρι στιγμής counters, gauges και timers.

5.6 Checkpointing

Όπως έχουμε αναφέρει προηγουμένως το Samza πρέπει να είναι ανεκτικό σε βλάβες. Για να το πετύχουμε αυτό τοποθετούμε στο properties file την ρύθμιση για να κάνει το σύστημα περιοδικά checkpoints.

```
# Ορίζουμε checkpoints στον kafka
task.checkpoint.factory=org.apache.samza.checkpoint.kafka.KafkaCheckpointManagerFactory
task.checkpoint.system=kafka

# Από default, το checkpoint  γράφει κάθε 60 δευτερόλεπτα. Με αυτήν την ρύθμιση εμείς μπορούμε να γράφουμε με ότι τιμή θέλουμε (msec).
task.commit.ms=60000
```

5.7 Windowing

Συνήθως στο stream processing τα jobs που έχουμε δημιουργήσει, τόσο σε Samza αλλά και σε άλλες πλατφόρμες όπως το Flink, πρέπει να υπολογίζουν περιοδικά κάτι συγκεκριμένο. Κάτι τέτοιο γίνεται πολύ εύκολα σε συνδυασμό με τα παράθυρα στο Samza. Για να το εξηγήσουμε καλύτερα αυτό, θα επιστρέψουμε στο πρώτο μας παράδειγμα με τους αισθητήρες από το φωτοβολταϊκό πάρκο. Ας υποθέσουμε λοιπόν, ότι έχουμε ένα job, το οποίο υπολογίζει πόσες τιμές του αισθητήρα από το φωτοβολταϊκό πάρκο είναι εκτός ορίων. Μια δεύτερη υπόθεση είναι ότι ο αισθητήρας στέλνει δεδομένα κάθε δέκα δευτερόλεπτα. Εμείς θέλουμε να ξέρουμε την αστοχία μέτρησης που έχει ο αισθητήρας κάθε δέκα λεπτά. Για να το πετύχουμε αυτό, έχουμε έναν μετρητή τον οποίο τον αυξάνουμε κάθε φορά που έχουμε μία αστοχία. Μια φορά στα δέκα λεπτά, το οποίο είναι και το μέγεθος του παραθύρου μας, στέλνουμε το αποτέλεσμα σε ένα topic και μηδενίζουμε τον μετρητή μας. Ακολουθεί ένα χαρακτηριστικό παράδειγμα.

```
public class SensorCounterTask implements StreamTask, WindowableTask {
  public static final SystemStream OUTPUT_STREAM = new SystemStream("kafka", "events");
  private int failCounter = 0;

  public void process(IncomingMessageEnvelope envelope,
                      MessageCollector collector,
                      TaskCoordinator coordinator) {

        if (the_value_is_out_of_bounds())
        {
              failCounter++;
        }
  }

        public void window(MessageCollector collector,
                           TaskCoordinator coordinator) {
              collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM,
```

```
failCounter));
                failCounter = 0;
        }
}
```

Παράλληλα, όπως σε κάθε task που γράφουμε, πρέπει να προσθέσουμε και τα αντίστοιχα properties στο configuration file.
Για το παράθυρο, το μόνο που χρειάζεται είναι να προσθέσουμε το εξής:

```
# Καλεί το Παράθυρο κάθε 10 λεπτα
task.window.ms=600000
```

ΚΕΦΑΛΙΟ 6

6.1 Συμπεράσματα

Σε αυτό το σημείο, έχοντας δουλέψει και με τα δύο frameworks, είμαστε σε θέση να συμπεράνουμε ποιο από τα δύο υστερεί ή υπερέχει σε κάτι.

Μιλώντας πάντα για scalable συστήματα, στο οποίο πολύ πιθανόν να έχουμε κάποιο message queue protocol, θα βόλευε πολύ να χρησιμοποιηήσουμε το Samza, και αυτό διότι ενσωματώνει το Kafka, το οποίο είναι ένα τέτοιο. Παράλληλα, μας δίνει την δυνατότητα να δουλέψουμε και με άλλα συστήματα όπως το RabbitMq.

Ένα άλλο βασικό ερώτημα θα ήταν, ποιο από τα δύο συστήματα είναι πιο γρήγορο στο deploy, ή αλλιώς σε ποιο από τα δύο έχουμε πιο γρήγορη παραγωγή τελικού κώδικα. Σε αυτό και πάλι υπερτερεί το Samza, καθώς, γράφουμε πολύ εύκολο κώδικα, χωρίς ιδιαίτερες γνώσεις του framework, παρά μόνο φτιάχνοντας configuration files και Java native κώδικα. Από την άλλη, το Flink, χρησιμοποιεί δικές του μεθόδους και συναρτήσεις, οι οποίες απαιτούν την πλήρη κατανόηση πρώτα για να μπορέσουμε να γράψουμε πρόγραμμα με ουσιαστικό αποτέλεσμα. Όσον αφορά το deploy, το Samza είναι πάλι πιο γρήγορο, καθώς μένει να "σηκώσουμε" τον server μας, και μετά γράφοντας κάθε φορά ένα αρχείο Java με το αντίστοιχο config file του, το κάνουμε host και τρέχει αμέσως. Από την άλλη, στο Flink κάθε φορά που δημιουργουμε ενα αρχείο Java πρεπει να κανουμε πάλι deploy ολο το project μαζι και να το σηκωσουμε παλι. Το Samza λοιπον, ειναι σε θέση αν "βλέπει" τα ανεξάρτητα αρχεια Java και να τα ενσωματώνει αυτόματα στο project.

Τέλος, πρέπει να αναφερθούμε στο ίσως πιο σημαντικό κομμάτι, αυτό της απόδοσης. Για να το συμπεράνουμε θα παρουσιάσουμε μια πειραματική μελέτη, με πραγματικά νούμερα όπως φαινεται παρακάτω.

Σημειώνεται ότι οι δοκιμές έγιναν σε Amazon Cloud, και σε VM με τις παρακάτω προδιαγραφές:

Πυρήνες : 4
RAM : 16 GB
Δίσκος : 60 GB SSD
Δίκτυο : VDSL

Η περιγραφή του πειράματος είναι απλή. Ακολουθεί το μοντέλο producer-consumer, και έχει ως εξής:

Έχουμε έναν producer, ο οποίος παράγει μηνύματα τύπου “Number: x”, όπου x είναι ένας αριθμός απο 1 μέχρι N. Το μύνημα αυτό το στέλνει σε ένα topic.
Από την άλλη πλευρά, έχουμε τον consumer ο οποίος διαβάζει το μύνημα από το topic αυτό, το επεξεργάζεται και τοποθετέι το συμπέρασμά του σε ένα άλλο topic.

Ας τρέξουμε το σύστημα για την Samza αρχιτεκτονική πρώτα.

6.2 Samza Producer

Ο κώδικας του producer παρουσιάζεται παρακάτω, με το N, το οποίο είναι το όριο μηνύματων που παράγει να είναι την πρώτη φορά 3.000, 2.000.000 και τέλος 100.000.000.

```
package SimpleProducer.SimpleProducer;

import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Properties;
import java.util.Scanner;
import java.util.concurrent.TimeUnit;
import org.apache.flink.api.common.JobExecutionResult;



/**

 * Created by ioakim on 16/3/2016.

 */



public class SimpleProducer {
        private static Producer<Integer, String> producer;
        private final Properties properties = new Properties();
        private static Scanner scanner;

        public SimpleProducer() {
                properties.put("metadata.broker.list", "localhost:9092");
                properties.put("serializer.class", "kafka.serializer.StringEncoder");
                properties.put("request.required.acks", "1");
                producer = new Producer<>(new ProducerConfig(properties));
        }

        public static void main(String[] args) throws FileNotFoundException {
                long startTime = System.currentTimeMillis();
                new SimpleProducer();
```

```
                String topic = "data";
                int k = 0;

                while (k <  ....N....) {
                        String counter = String.valueOf(k);

        // String for Samza
                String output = new StringBuilder().append("{\"NUMBER\":")
                                                   .append("\"").append(counter)
                                                   .append("\"}").toString();



                KeyedMessage<Integer, String> data = new KeyedMessage<>(topic,   output);
                        producer.send(data);
                        k++;
        }

        long endTime   = System.currentTimeMillis();
        long totalTime = endTime - startTime;

        System.out.println(totalTime);

    producer.close();


    }
}
```

6.3 Samza Consumer

Και εδώ παρουσιάζεται ο κώδικας του consumer, δηλαδή του host κωδικα στον samza server.

```
package samza.thesis.samza.task;

import java.util.HashMap;
import java.util.Map;
import org.apache.samza.system.IncomingMessageEnvelope;
import org.apache.samza.system.OutgoingMessageEnvelope;
import org.apache.samza.system.SystemStream;
import org.apache.samza.task.MessageCollector;
import org.apache.samza.task.StreamTask;
import org.apache.samza.task.TaskCoordinator;
import org.codehaus.jettison.json.JSONArray;
import org.codehaus.jettison.json.JSONObject;


/**

 * Created by ioakim on 16/3/2016.

 */


public class MyTask implements StreamTask {
        private static final SystemStream OUTPUT_STREAM_EVEN         = new
                                                SystemStream("kafka","even");
        private static final SystemStream OUTPUT_STREAM_ODD          = new
                                                  SystemStream("kafka","odd");

        public void process(IncomingMessageEnvelope envelope,
                                MessageCollector collector,
                                TaskCoordinator coordinator) throws Exception {


                try {
                        @SuppressWarnings("unchecked")
```

```
                Map<String,Object> json =
                        (Map<String,Object>)envelope.getMessage();

                String number=null;
                number = (String)json.get("NUMBER");

                HashMap<String,Object> val = new
                                                HashMap<String,Object>();

                if ((Integer.parseInt(number) % 2) == 0) {
                        val.put(number, null);
                        collector.send(new
                        OutgoingMessageEnvelope(
                                OUTPUT_STREAM_EVEN,val));

                } else if ((Integer.parseInt(number) % 2) != 0) {
                        val.put(number, null);
                        collector.send(new
                        OutgoingMessageEnvelope(
                                OUTPUT_STREAM_ODD,val));
                }
            } catch(Exception e) {
                System.err.println(e);
            }

        }

}
```

Ουσιαστικά, ο κώδικας του Consumer, κάνει την εξής απλή δουλειά. Για κάθε μήνυμα που φτάνει στο data topic, το οποίο δηλώνεται ως input από το αντίστοιχο config file, το διαβάζει και κάνει μια απλή πράξη. Μετατρέπει το String σε Integer, και μετά το κάνει mod 2. Εφόσων ο αριθμός είναι ζυγός, τότε το γράφει στο "even" topic. Διαφορετικά το γράφει στο "odd" topic. Στην συνέχεια, παραθέτουμε τον αντίστοιχο κώδικα για το Flink consumer-producer, οι οποίοι κάνουν ακριβώς την ίδια δουλειά όπως και στο Samza.

6.4 Flink Producer

Εδώ παραθέτουμε τον producer όπου και πάλι το N θα πάρει τις τιμές για 3.000, 2.000.000 και 100.000.000.

```
package SimpleProducer.SimpleProducer;

import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Properties;
import java.util.Scanner;
import java.util.concurrent.TimeUnit;
import org.apache.flink.api.common.JobExecutionResult;


/**
 * Created by ioakim on 16/3/2016.
 */


public class SimpleProducer {
        private static Producer<Integer, String> producer;
        private final Properties properties = new Properties();
        private static Scanner scanner;

        public SimpleProducer() {
                properties.put("metadata.broker.list", "localhost:9092");
                properties.put("serializer.class", "kafka.serializer.StringEncoder");
                properties.put("request.required.acks", "1");
                producer = new Producer<>(new ProducerConfig(properties));
        }

        public static void main(String[] args) throws FileNotFoundException {
                long startTime = System.currentTimeMillis();
                new SimpleProducer();
```

```
                String topic = "data";
                int k = 0;

                while (k <  ....N....) {
                        String counter = String.valueOf(k);

                        // String for Flink
                        String output = counter

                        KeyedMessage<Integer, String> data =
                                        new KeyedMessage<>(topic, output);
                        producer.send(data);
                        k++;
                }

                long endTime   = System.currentTimeMillis();
                long totalTime = endTime - startTime;

        System.out.println(totalTime);

    producer.close();

    }

}
```

6.5 Flink Consumer

Παράλληλα, ο consumer, έχει την ακόλουθη μορφή:

```
package flinkThesis.flinkThesis;

import org.apache.flink.api.common.JobExecutionResult;
import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.typeutils.TypeExtractor;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer09;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer09;
import org.apache.flink.streaming.util.serialization.DeserializationSchema;
import org.apache.flink.streaming.util.serialization.SerializationSchema;
import org.apache.flink.streaming.util.serialization.SimpleStringSchema;
import java.util.Properties;
import java.util.concurrent.TimeUnit;


/**
 * Created by ioakim on 16/3/2016.
 */


public class MyFlink {

        public static void main(String[] args) throws Exception {

                StreamExecutionEnvironment env =
                        StreamExecutionEnvironment.getExecutionEnvironment();

    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers", "localhost:9092");
```

```
properties.setProperty("zookeeper.connect", "localhost:2181");
properties.setProperty("group.id", "thesis_comparison");

DataStream<String> messageStream = env.addSource(new
                    FlinkKafkaConsumer09<>("data", new
                                        SimpleStringSchema(), properties));



DataStream<String> filtered_even = messageStream.filter(new
                                        FilterFunction<String>() {

            @Override
            public boolean filter(String value) {
                    return (Integer.parseInt(value) % 2) == 0;
            }}
    );

    filtered_even.addSink(new
    FlinkKafkaProducer09<>(properties.getProperty("bootstrap.servers"),
                                                                    "even",
                                                    new MyStringSchema()));

DataStream<String> filtered_odd = messageStream.filter(new
                                                FilterFunction<String>() {

  @Override
  public boolean filter(String value) {
                    return (Integer.parseInt(value) % 2) == 0;
            }}
    );

    filtered_odd.addSink(new
    FlinkKafkaProducer09<>(properties.getProperty("bootstrap.servers"),
                                                                    "odd",
                                                    new MyStringSchema()));

JobExecutionResult result = env.execute("My Flink Job");

System.out.println("******The job took " +
    result.getNetRuntime(TimeUnit.SECONDS) + " seconds to execute");

}
```

```
public static class MyStringSchema implements DeserializationSchema<String>,
                                                SerializationSchema<String> {

        @Override
        public byte[] serialize(String element) {
                return element.toString().getBytes();
        }


        @Override
        public String deserialize(byte[] message) {
                return (new String(message));

        }


        @Override
        public boolean isEndOfStream(String nextElement) {
                return false;

        }


        @Override
        public TypeInformation<String> getProducedType() {
                return TypeExtractor.getForClass(String.class);

        }
    }
}
```

6.6 Τελικές μετρήσεις - χρόνοι

Κάνοντας deploy τους παρακάνω κώδικες, παίρνουμε τις ακόλουθες μετρήσεις:

Για N = 3.000
Samza : 5842 ms
Flink : 7785 ms

Για N = 2.000.000
Samza = 775.379
Flink = 795.287

Για N = 100.000.000
Samza =27.084.624 ή αλλιώς ~ 7.5 ώρες
Flink = 28.762.766 ή αλλιώς ~ 8 ώρες

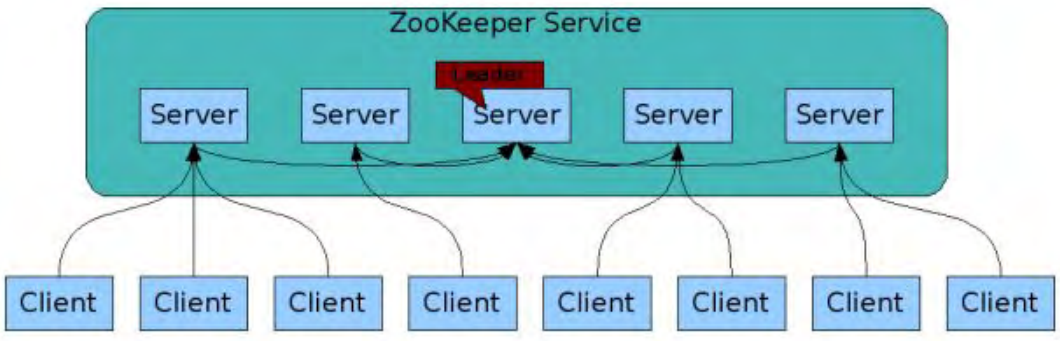
Συνοψίζοντας λοιπόν τις πραγματικές μετρήσεις, μπορεί κανείς να συμπεράνει ότι το Samza σύστημα υπερέχει έναντι του Flink τόσο σε απόδοση, όσο και σε ευκολία στην παραγωγή κώδικα.

ΠΑΡΑΡΤΗΜΑ

A.1 Zookeeper

Είναι ένα πολύ βασικό στοιχείο το οποίο ενσωματώνεται σε πολλά κατανεμημένα συστήματα για τον συγχρονισμό και την οργάνωση των κόμβων. Το Zookeeper τρέχει σε συστοιχίες υπολογιστών και έχει σχεδιαστεί έτσι ώστε να αποθηκεύει τις "συντεταγμένες" του συστήματος, όπως πληροφορίες, ρυθμίσεις κ.ά..

Το σύστημα του Zookeeper λοιπόν, αποτελείται από τους Servers και από τους clients. Οι Servers όλοι μαζί συνθέτουν την υπηρεσία του Zookeeper. Οι Clients απο την άλλη, είναι ουσιαστικά TCP connections οι οποίες συνδέονται όλες σε κάθε ένα Server.



Αναφορές

[1] J. Gantz, and D. Reinsel. (2012). The digital universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. [Online]. Available:http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

[2] Storm vs. Spark Streaming: Side-by-side comparison. [Online]. Available:http://xinhstechblog.blogspot.pt/2014/06/storm-vs-spark-streaming-side-by-side.html. Accessed 20 Dec 2014.

[3] Joel Koshy. (2014) Transactional messaging in Kafka. [Online]. Available:https://cwiki.apache.org/confluence/display/KAFKA/Transactional+Messaging+in+Kafka.

[4] Tao Feng. (2015) Benchmarking Apache Samza: 1.2 million messages per second on a single node. [Online]. Available:http://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node.

[5] N. Marz.(2011) How to beat the CAP theorem. [Online] Available: http://nathanmarz.com/blog/ how-to-beat-the-cap-theorem.html

[6] Jay Kreps. I Heart Logs. O'Reilly Media, September 2014. ISBN 978-1-4919-0932-4.

[7] Jay Kreps. Why local state is a fundamental primitive in stream processing, July 2014. [Online] Available: http://radar.oreilly.com/2014/07/why-local-state-is-a-fundamental-primitivein-stream-processing.html.

[8] Jay Kreps. Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines), April 2014. [Online] Available: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[9] Jay Kreps. (2015) Putting Apache Kafka to use: a practical guide to building a stream data platform (part 2). [Online] http://blog.confluent.io/2015/02/25/stream-data-platform-2/.

[10] Alex Woodie. (2016). Apache Flink Creators Get $6M to Simplify Stream Processing. [Online] Available: http://www.datanami.com/2016/03/30/apache-flink-creators-get-6m-simplify-stream-processing/

[11] D. J. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. The VLDB Journal, 12(2):120–139, Aug. 2003.

[12] K. V. Shvachko, "HDFS Scalability: The limits to growth". April 2010, pp. 6–16

[13] S. Weil, S. Brandt, E. Miller, D. Long, C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," In Proc. of the 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, November 2006.

[14] G. Mackey, S. Sehrish, J. Wang, Improving metadata management for small files in HDFS, In 2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09), New Orleans,Sept, 2009, pp.1-4.

[15] What is Hadoop. http://www-01.ibm.com/software/data/infosphere/hadoop/. Accessed 24 May 2016

[16] Available:http://samza.apache.org/learn/documentation/0.10/api/javadocs/. Accessed 15 Jun 2016

[17]Available:https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/streaming/index.html. Accessed 24 May 2016