KTH Royal Institute of Technology

School of Information and Communication Technology

Degree project in Distributed Computing

Topology-Aware Placement of Stream Processing Components on Geographically Distributed Virtualized Environments

Author:	Ken Danniswara
Supervisors:	Ahmad Al-Shishtawy, SICS, Sweden
	Hooman Peiro Sajjad, KTH, Sweden
Examiner:	Vladimir Vlassov, KTH, Sweden

Abstract

Distributed Stream Processing Systems are typically deployed within a single data center in order to achieve high performance and low-latency computation. The data streams analyzed by such systems are expected to be available in the same data center. Either the data streams are generated within the data center (e.g., logs, transactions, user clicks) or they are aggregated by external systems from various sources and buffered into the data center for processing (e.g., IoT, sensor data, traffic information).

The data center approach for stream processing analytics fits the requirements of the majority of the applications that exists today. However, for latency sensitive applications, such as real-time decision-making, which relies on analyzing geographically distributed data streams, a data center approach might not be sufficient. Aggregating data streams incurs high overheads in terms of latency and bandwidth consumption in addition to the overhead of sending the analysis outcomes back to where an action needs to be taken.

In this thesis, we propose a new stream processing architecture for efficiently analyzing geographically distributed data streams. Our approach utilizes emerging distributed virtualized environments, such as Mobile Edge Computing, to extend stream processing systems outside the data center in order to push critical parts of the analysis closer to the data sources. This will enable real-time applications to respond faster to geographically distributed events. We create the implementation as a plug-in extension for Apache Storm stream processing framework.

Acknowledgment

I am deeply thankful to my supervisors, Ahmad Al-Shishtawy and Hooman Peiro Sajjad for the chance of working together and for their continuous support and encouragement on this master thesis work. Working with them give me a great experience and many pleasures.

I would also like to give gratitude to European Master of Distributed Computing (EMDC) coordinators for giving me the opportunity to experience their two years master programme. All my EMDC classmates: Sana, Igor, Bilal, João, Fotios, Daniel, Sri, Gayana, Gureya, Bogdan, and Seckin.

My final gratitude is for my parents and my sister that always supporting me from afar.

Stockholm, 30 September 2015

Ken Danniswara

Contents

List of Figures

List of Tables

Listings

Chapter 1 Introduction

1.1 Motivation & Problem Definition

Distributed stream processing system (DSPS or Stream Processing) [15] has become one of the research trends in Big Data concept alongside batch processing. With batch processing approach, we are able to do computations from very large amount of data. Examples include querying from a database, massive image processing, or data conversion. Based on the nature of static datasets, batch processing appears to be an ideal technique, both in terms of data distribution, task scheduling, and distributed batch processing frameworks. But, this traditional concept of store-first, process-second architectures are unable to keep up with large volume of arriving data in a very short period. To process each data individually on-the-go and calculate the result in real-time, stream processing is the most suitable solution.

Looking through at the use cases of stream processing, we divide it into two types based on the system response time. Even though the use of stream processing is generally focused on fast or low-latency processing, some use cases like Twitter trending topic analytics are not considered latency critical application. Couple seconds or even a minute of latency is still acceptable for the user. However, in global market exchanges or electronic trading, a process latency of one second is most of the time unacceptable. In this thesis, we are going to focus on the latency-critical application where the results are expected to appear in the range of milliseconds.

Other way to categorize the different application of stream processing is the location of the data sources. Currently, a common use of stream processing application is to receive the stream from databases or message brokers, often parallelized for scalability. The raw data sources from many locations are collected into an intermediate pooling system before it is processed. We called this as centralized source location.

While it is convenient to process data from single location, the growth of the data source emitter: Mobile phones, Internet-Of-Things (IoT) devices, or sensors that are spread in different location creates another problem if we want to do real-time processing. Data source that is located far from where the stream processing computation takes place will suffer from the high-latency communication. By using the new *EdgeCloud* concept, it is possible to collect and perform stream processing directly on each source location. As now the sources are not previously gathered before processed, we generalized those scattered resources as Geo-distributed data sources.

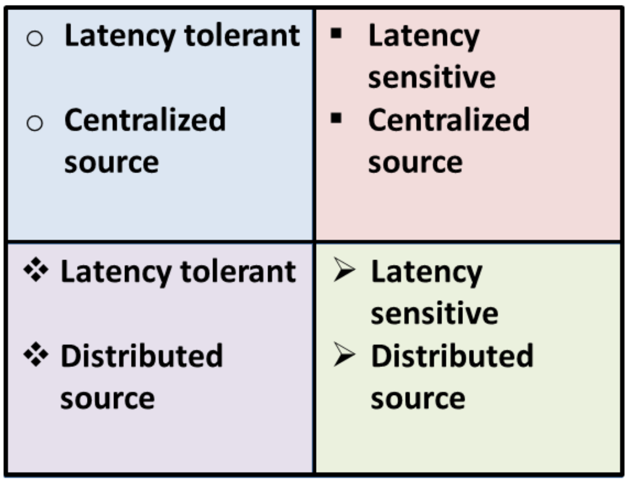


Figure 1.1: Four domains of Stream processing

From two categorization above, Figure 1.1 shows our visualization of different approach to build a stream processing application. In this thesis, we focused on latency-sensitive / real-time application where we distributed the data processing based on the data source locations. According to our observation, research in this area is still relatively new.

1.2 Approach

We started by looking at the new concept of Edge Cloud model. Edge Cloud consists of multiple small data-center / clouds that has a very good prospect to be able to process the distributed data sources in a more efficient way. Then we analyze one of the stream processing frameworks, namely Apache Storm, focusing on its performance when deployed distributively in this model. The focus on this part is to identify the possible bottleneck that could reduce the performance. The results are used as a cornerstone for our proposal to create a better deployment of Storm components (Storm scheduler) for Geo-distributed Edge Clouds. The implementation for the

Apache Storm addition is created by using Storm plug-in API. With this addition, we are expecting a higher performances and better response time for latency-sensitive application compared to using the default deployment.

1.3 Contribution

We have created a new type of Apache Storm scheduler and stream distribution protocol for a deployment in multiple data-centers or clouds. This addition promotes the locality for Geo-distributed sources where each data will be processed in the closest location from where it generated which can significantly reduce the effect of high-latency connection in the backbone network.

The result is presented as an Apache Storm plug-in. There is no modification on the default Storm release (version 0.9.3) even though there is a need to add third-party information to make the scheduler able to run as expected. In the future, we hope that this project will be integrated to the main Storm deployment branch to implement more complex scheduling system.

With this research, we are also contributing to open-source stream processing communities, especially Apache Storm community, to create a proof-of-concept of deploying a single Storm instances on a multiple heterogeneous cloud deployment.

1.4 Structure of the Thesis

Chapter gives the necessary information and components used in our work: stream processing, Apache Storm, Edge Clouds, and CORE Network Emulator. Chapter explains the motivation and idea to deploy Apache Storm in a multi-cloud environment. We did some experiments with two different network environments to observe the performances and find the possible bottlenecks.

Chapter discusses the possibility and considerations of running a real-time application in multiple data-center or cloud model. As a result, in this chapter we propose a new scheduler and stream Grouping that will work in this cloud model. Chapter explains the implementation of our algorithm and discusses some features that are not implemented because of time restriction. Chapter evaluates the performance of our proposed scheduler and stream Grouping compared with the Storm default implementation.

Chapter concludes the thesis report by discussing our proposed Scheduler and stream Grouping, and considerations as well as directions for future work.

Chapter 2
2 Background

2.1 Stream Processing

In-memory stream processing has become one of the trends in Big Data concept alongside batch processing. The disadvantage of batch processing is it cannot provide low latency responses needed when the data is continuously arriving to the system. To process each data individually and get the result in real-time, the stream processing is the more suitable solution.

In stream processing, the data treated as streams of events or *Tuples.* The stream travels from its point of origin and passes through different processing units without saving the immediate results in permanent location first. In this way, the data is processed as they arrived, passed to the next one, and makes the result possible to be presented in almost real-time.

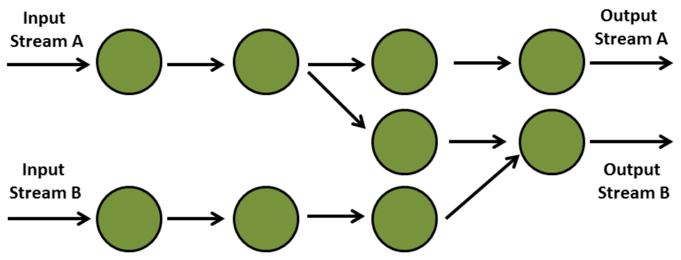


Figure 2.1: Stream processing, each green circle is a processing unit

Stream processing is usually deployed in a single data-center or cloud. This is because placing the components in different location via network connection could create a latencies when sending the Tuples which in turn could reduce system performance.

2.1.1 Apache Storm

Apache Storm is an open-source stream processing project launched in 2012. Storm is created by BackType and then acquired by Twitter in 2011 for their main real-time processing jobs. By 2014, 60 companies have used and/or experimented with Storm [22].

We choose Apache Storm than the other open-source stream processing frameworks because of several considerations. First of all, Apache storm can be seen as a mature project. It started in heavy development from 2011 and still continuing under Apache open-source hood until the current time this Thesis is being worked on 2015 (In the last six months, Storm have undergone several major updates). In the term of technical consideration, Apache Storm is the most suitable from the new environment as they provide a very robust and stateless pure-stream processing to be able to be deployed in multi-cloud environment. There is also a possibility to run mini-batch streaming or stateful process with Trident, an extension that runs on top of Storm. As the core system is still the same, our modification on the lower-part of Storm will still be used without breaking the current instance.

Processing stream in Apache Storm is based on user-defined flow graph called Storm Topology. A topology consists of processing elements (PE) and how each tuples will move along between PE. Usually, the process starts from the PE that handles the stream source and tuple creations (Spout), to number of different PEs (Bolts) until the last one that did not emit more streams. The Topology is submitted to a running Storm instances with the default nature of 'Always Run', where it is expected to run indefinitely until it is stopped by user command or system faults.

Apache Storm structure is based on multiple loosely-coupled components managed by a third party coordination server (Apache Zookeeper). Zookeeper is another Apache open-source project for maintaining services needed by distributed application such as naming, configuration information, synchronization, and providing group services [24]. The Storm components are divided based on master-slave architecture. One component will act as a leader that assign and control jobs to the other worker components. Below are the explanations of Apache Storm component terms:

1. **Nimbus** : Nimbus is a leader component in Storm. A process is started by the user deploying Topology in the Nimbus, where it will distribute the assignments to the Workers inside the Supervisor machine. Nimbus find the list of living Supervisor and their location from Zookeeper. Nimbus itself is run as a Java daemon and do not perform any computation process.

2. **Supervisor** : High-level worker component in Storm. Supervisor is run as a Java process and deployed once in each machine, physical or virtual. Each

living Supervisor that is connected to the Zookeeper is able to receive assignments from Nimbus. Supervisor is called high-level worker because it does not do any computation by itself, but rather creates and manages multiple Workers to do the computation. As the Supervisor is a different Java process from the Workers, the Workers can still run normally even when the Supervisor is down without interrupting the processed stream, at least until the connection timeout between Supervisor and Zookeeper is reached.

3. **Worker** : Java process created by the Supervisor in the same machine. Worker receives tasks from Nimbus and then creates the Executor thread to run the tasks.

4. **Executor** : Executor is a thread inside worker running a task. There can be any number of Executor thread inside a single Worker process. By default, each Executor will only have one task, meaning if a worker needs to run 10 Tasks, then there will be 10 Executor threads.

5. **Task** : Task is a real implementation of stream Processing Elements (Bolt and Spout) created in the user Topology

6. **Bolt** : Bolt is a Storm Processing Element that receive an input stream and is able to produce any number of output stream. Bolt can receive stream from another Bolt or Spout. Bolt consists as a logic computation like a Java class that will be able to do any function. In Stream Processing, Bolts usually perform simple tasks like filtering, streaming aggregation / joins, write to databases, connect to another applications, and so on.

7. **Spout** : Spout is a special type of Bolt that became the source of the stream. Spout cannot receive a stream from another Bolt / Spout, but is dedicated to read and create Tuples from outside Storm system like message brokers (Kafka or RabbitMQ), web API (Twitter API), databases (HBase, HDFS, Cassandra), text files (system logs), or any other source.

Visualization of storm components described above can be seen in figure 2.2 & 2.3. Figure 2.2 gives the bird-eye view of the Storm master-worker architecture. Each node can be located in a single machine (Local deployment) or distributed in different machines (Cluster deployment). From this picture, we can see that initially both Nimbus and Supervisors nodes status are managed via Zookeeper. Figure 2.3 gives more detail on the computation machine or Supervisor nodes. Single machine only need one Supervisor process to register themselves in the Storm cluster. Max number of workers that can be created are based on Supervisor configuration and cannot be changed in the runtime. Each worker can only run the tasks from single topology, as seen in figure 2.4. The Executor thread to run the assigned tasks is called inside each worker. By default Storm scheduler, the tasks will be distributed in a round robin way. There are different studies to create more complex scheduler. This part will be discussed more in the next chapter.

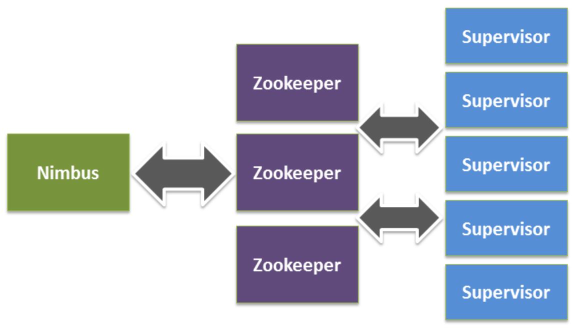


Figure 2.2: Apache Storm Master-Worker Architecture

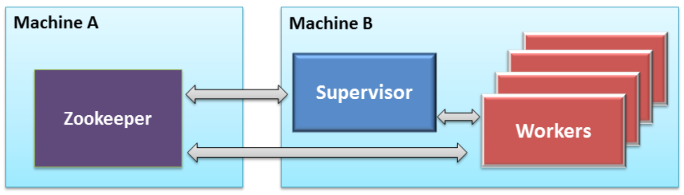


Figure 2.3: Communication between Zookeeper and Supervisor Nodes

Apache Storm advantages that are important to be focused on this thesis work is it's robustness and scalability. Every Nimbus, Supervisor, and Worker components are independent Java Virtual Machines process (JVM) that expected to be able to stop working anytime (fail-fast) without affecting the whole Storm system: Dead Workers will be restarted by their Supervisor in the same machine. Dead Supervisor process won't affect Workers assignments and the stream of the tuple can still keep continuing for a short time. If a Supervisor downtime is exceeded the timeout by the Zookeeper, the whole machines are considered dead and all tasks assignments from the dead supervisor will be reassigned to other machine/Supervisor by Nimbus. In the case of dead Nimbus, The whole Storm process will keep running as long as the Zookeeper is alive. In the Apache Storm guidelines, Nimbus, Zookeeper, and Supervisor Java processes are supposed to be handled and automatically restarted by a 3rd-party control system like Supervisord [8]. The Zookeeper nodes should also run in multiple machines for better fault tolerance and easier consensus solving problem (odd number with minimum of 3).

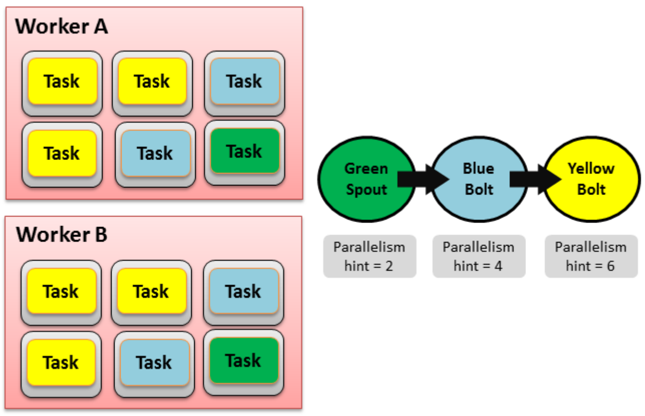


Figure 2.4: Task distribution inside Worker process

Apache Storm loosely-coupled component also create better throughput scalability to handle different amount of input data rate or stream flow. Every Processing Elements or Tasks can be paralleled into different amount and distributed to different Workers. Parallelization level on each Task usually depends on the capability to handle the speed of incoming stream, process, and send the result stream to the next Task. When over-provisioning Tasks are possibly less harmful, under-provisioning tasks can be very bad for the whole Storm performance. Slower processing rate compared with the stream input rate will create a bottleneck in the system and create queue of unprocessed Tuples. This is where increasing parallelization is important to distribute the flow rate of a stream.

To make sure the Task parallelization did not affect the correctness of the result, Storm has seven types of Grouping protocol of how the stream is distributed between two or more Tasks.

- **Shuffle Grouping** : Tuple distributed in round-robin to every Task object receiving the stream. This Grouping guarantees each task to receive same amount of Tuple.
- **Field Grouping** : The stream is divided by the fields specified in the Grouping. Tuple that has same field value will always sent into the same Task. Field grouping can be used for creating stateful computation on a Task as every Tuple arrives will have the same field attribute.
- **LocalorShuffle Grouping** : This Grouping will prioritize sending the Tuple into the next Task that is located in the same Worker process. If there are no aimed Task in the same Worker, it behaves like Shuffle Grouping. LocalOr-Shuffle Grouping is used for a no latency Tuple transmission between Tasks as intra-worker communication is done inside single Java process without using any network protocol.

- **Partial Key Grouping** : Similar with Fields Grouping with better load balance between two or more bolts that are receiving same field value.
- **All Grouping** : Each Tuple in this stream are replicated to all receiving Tasks.
- **Global Grouping** : All Tuple in this stream will be sent to a single Task with lowest ID.
- **Direct Grouping** : Special type of Grouping where the sender Task decides which Task will receive the Tuple. It has different stream implementation where it needs to assign the receivers Task ID.

2.2 Edge Cloud / Cloud on Edge

In the concept of network infrastructure, network edge is a term for part of the network that is close to the end user. For example, network edge can be a telecommunication operator company base stations network where mobile phone directly connected into, or connection between local Internet Service Provider (ISP) routers before it connected to the higher network tier. Network edge have less latency compared with the connection to the rest of Internet as the location is relatively close to the user and less number of network hops [13]. Moreover, bringing part of the computation to the network edge is believed to be able to reduce the network load where the rest of the process is located. This process is called edge computing.

One of the current researches on edge computing is to create a cloud from edge infrastructure. There are three samples of implementing Cloud on Edge or Edge Cloud: on mobile carrier network infrastructures (Section 2.2.1), Telecom base stations (Section 2.2.2), and Community Network (Section 2.2.3). Each implementation have a different purpose and deployment methods (network topology & cloud resources), but with the same concept of enabling application to be put on top of or beside their main utilities.

There are two reasons why it is fundamentally make senses to move the computation to the Edge Cloud: Firstly, the new concept of Internet-of-things (IoT) where IP based networking will be embedded to all type of devices, appliances, consumer electronics, and small sensors. Newest fifth-generation (5G) mobile network also helps enabling the concept by improving the network capabilities even further. However, when all of the devices are connected, the amount of data these systems are generating will keep increasing, which will burden the existing network. Making the computation as close as possible to where the data is generated can significantly reduce the data that moving through the network and decrease the number of network traffic bottlenecks.

Secondly, moving the computation to the edge is more suitable for real-time and latency-critical type of application. Each device will have different performance based on the location or network hops. Distributing this computation to the edge will significantly reduce the latency and better response time. Moreover, if each Edge Cloud server only processes the data from limited area (geographical distribution), the load on each server will be lower than the single centralized cloud.

Edge Cloud can be used as a single cloud instance or to be combined with current centralized cloud infrastructure. With part of the services located on the Edge, we can enhance the cloud experience by segregating the local information based on the location, while the centralized cloud infrastructure is still maintained for global computation or aggregation.

2.2.1 Carrier Cloud

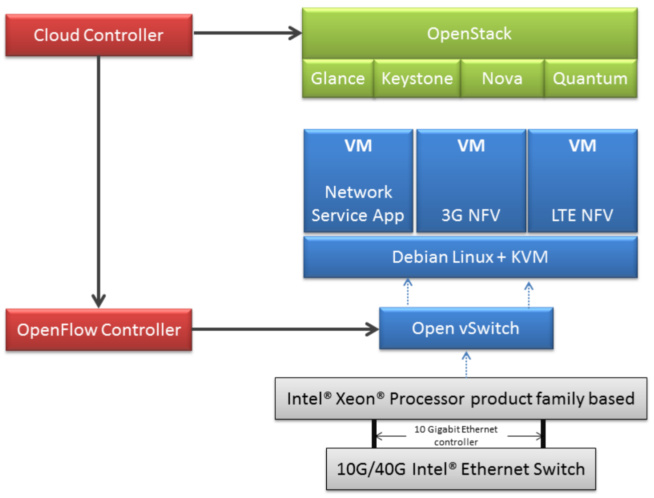


Figure 2.5: Intel carrier cloud system architecture (Simplified from [18])

Carrier Cloud is one of the emerging cloud models located on the network edge. In carrier cloud, mobile telecommunication operator hosts Cloud Computing services on their carrier network infrastructures. Growth of the network and variety of new technologies are the main reason for the companies [2] to change their hardware nodes. Single-function machines / carrier-grade routers and switches are evolving into the general purpose CPU hardware with abstraction of the network function (Network Function Virtualization & Software Defined Network). In figure 2.5, single Ethernet switch with Xeon®based processor provides virtualized network component under Open vSwitch, while OpenStack run in the same machine. With the cloud platform available in the system, lots of improvement and new features can be made. In 2014, Nokia and Intel build a partnership with UK mobile operator EE to upgrade the base station with Intel-based server[7].

2.2.2 Cloud-RAN

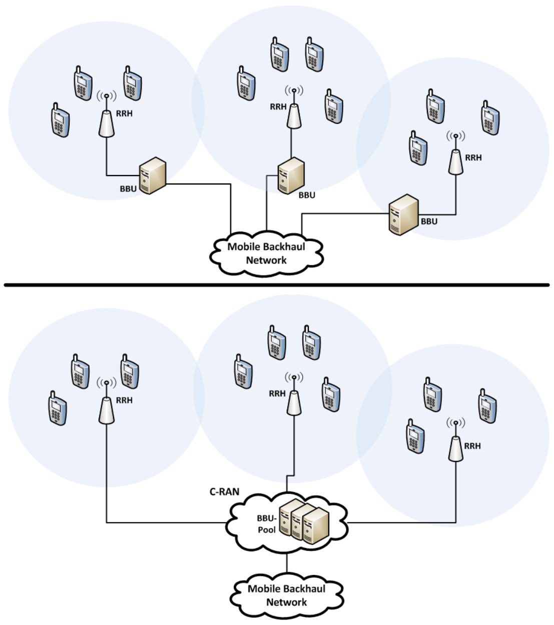


Figure 2.6: Top: Traditional mobile network with BBU-Unit on each location. Bottom: Multiple BBU-unit pooled in a single location

Cloud-RAN (Radio Access Network) is a new model for base stations mobile network. The idea of cloud-RAN is first initiated by IBM with the name of Wireless Network Cloud (WNC) [19]. The concept of Cloud-RAN is to apply cloud-computing technologies on structures behind mobile network architecture. In mobile network architectures, every base station tower is accompanied with two structures: RRH (Remote Radio Head) that processes the DAC (Digital-to-Analog) and ADC (Analog-to-Digital) conversion from/to the tower antenna and BBU (Baseband Unit) or DU (Data Unit) that works more on computation like sampling, mapping, Fourier Transform, and transport protocol. In this thesis we will not discuss about both structures in detail, but we will focus on the network relation between these components.

The differences between traditional and Cloud-RAN mobile network architectures is the modification of Baseband-Unit (BBU), as can be seen in figure 2.6. In traditional mobile network, every base station tower has a dedicated BBU-Unit. This concept have disadvantages on cost and power consumption needed for each base station because number of BBU machine must follow the number of RRH tower. Also, communication between BBU takes more time as the information needs to be sent to the Backhaul network first. In cloud-RAN, multiple BBU for multiple base stations are combined into single BBU pool. This pool will then act as a single cloud system that controlling multiple RRH / base station in a single time. Communication between BBU will then occur less often as the unit located in the same place. Based on load information, over-provisioning or under-utilization can be avoided where increasing or decreasing number of BBU machines also became easier as the administrator can control the centralized and on-demand system.

Table 2.1: Requirements for Cloud Computing and Cloud-RAN applications (Taken from [14])

	IT - Cloud Computing	**Telecom - Cloud RAN**
Client/Base station data rate	Mbps range, bursty, low activity	Gbps range, constant stream
Latency and Jitter	Tens of ms	< 0.5 ms. jitter in ns range
Lifetime of information	Long (Content data)	Extremely Short (data symbols and received samples)
Allowed recovery time	s range (Sometimes hours)	ms range to avoid network outage
Number of clients per centralized location	Thousands, even millions	Tens, maybe hundreds

This Cloud-RAN network is an example of Cloud on the edge. From the previous paragraph, We could imagine a single deployment of BBU pool as one cloud system and the connection between multiple BBU pool is a connection between clouds via backhaul network. Every cloud has information on their own quota and computation power which makes deploying third-party software a possibility. The deployed software can be used for enhancing the main mobile network system. For example, the system could have traffic distribution, Trans-receiver selection, Functional component to physical mapping, and make decisions from previous configurations. Table 2.1 from [14] give us the insights of the requirement needed for application inside the Cloud-RAN system. While number of the clients can be lower than normal cloud computing application because the cloud are distributed in different location, there is a demand for very-low latency and short-life processed data span. We believe that real-time processing like Stream Processing will be suitable to run inside Cloud-RAN system.

2.2.3 Community Network Cloud

Community network is a local communication infrastructure in which a community of citizen build, operate, and own open IP-based networks[12]. Community network is mainly used for Internet sharing solution in an area without or have a bad quality connection to commercial telecom operators. In addition, community network can provide different services like web space, e-mail, distributed storage[21], or cloud services[12]. As explained in those papers, the most suitable design to deploy cloud in wireless community network is a set of microclouds. Each microcloud is a cloud resource that is defined by geographical zone and connected to each other by a super-node. Group of microclouds in community network is similar with the concept of edge cloud where each end-users are located in an area with connection to the relatively closest cloud. Each microcloud is able to communicate with each other or connect to the bigger cloud located outside the community network.

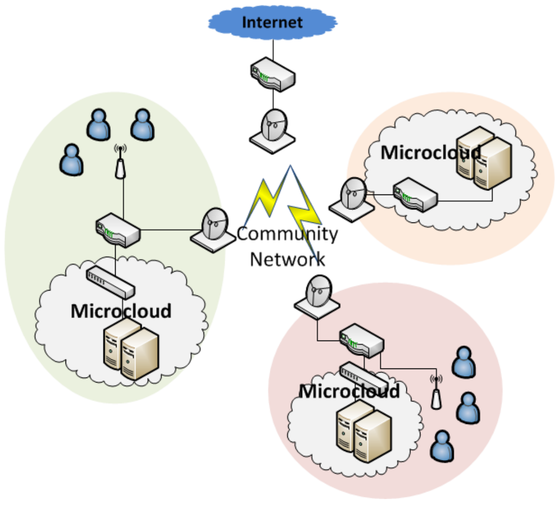


Figure 2.7: Microcloud in Community Network

2.3 Emulation Software: CORE Network Emulator

CORE network emulator is an open source network emulation framework tool developed by Network Technology research group, part of Boeing Research and Technology division, in United States Naval Research Laboratory (NRL)[1]. CORE is a derivative project from Integrated Multiprotocol Network Emulator/Simulator (IMUNES) where the concept of lightweight virtual network instances is presented into FreeBSD 4.11 or 7.0 operating system Kernel[10].

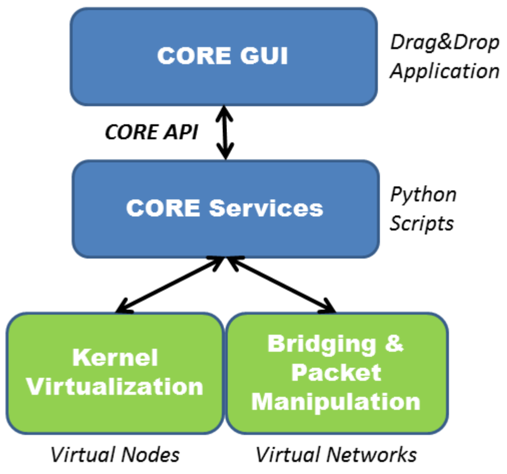
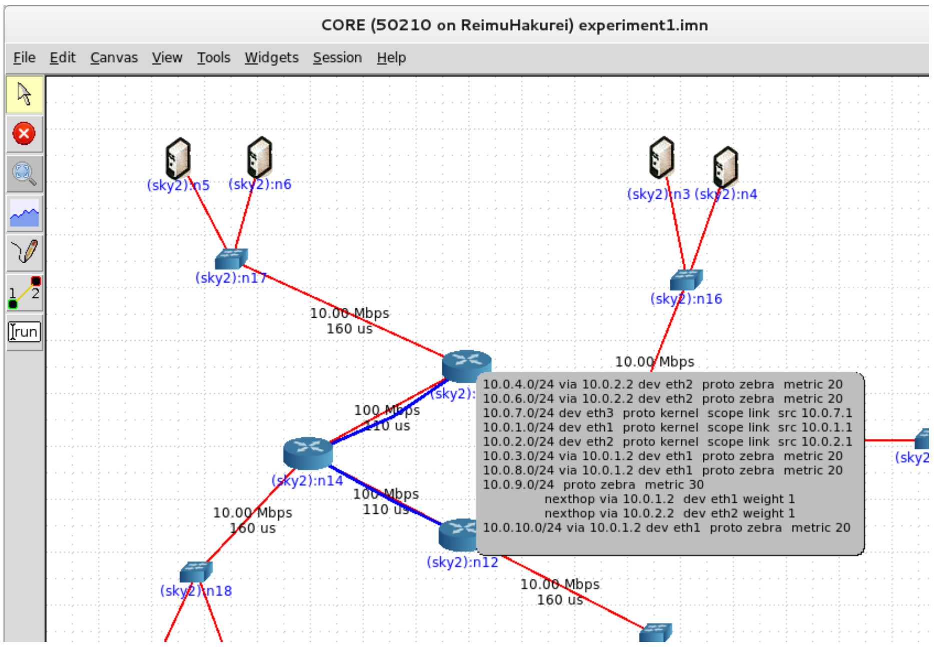


Figure 2.8: Architecture of CORE network emulator

CORE basic architecture can be seen in Figure 2.8. On the top there is a CORE-GUI application where the user can directly interact to create topologies. User can place nodes with different capabilities (Routers, PCs, Servers, Switches, etc) and draw network links between the nodes. User can also control the quality (maximum bandwidth, bit-error rate, latency, and latency jitter) for every network links between two nodes, be it both ways or one direction only.

In execution time, CORE-GUI uses their own CORE-API to give instruction to the CORE services that is connected to via TCP socket-based connection. The GUI itself can be run in different machine with the services. In the run-time, CORE-GUI also has multiple features like customizable widgets to show basic information without opening the interactive shell on each nodes, seen in Figure 2.9. There are also start-up scripts and mobility scripts to send any command to multiple nodes in the run-time.

Under the CORE-GUI, main system of CORE is run as python services. This services is responsible to instantiating Vnodes and virtual network stack in the lower layer. Vnode technology used by CORE is Linux OpenVZ containers. Each Vnode container has private file system and security control. Other features like disk size or memory quota are disabled and shared with the hosts machine. Each Vnode has their own Linux kernel namespaces with clone() system call. New process or application run inside the Vnode will be forked from the Vnode main process and still can be seen as single different process from the host machine. For the virtual network stack, CORE create pairs of *veth* (Virtual Ethernet) in the host machine for every links. Linux Ethernet bridging is then used to connect the *veth* together. This way, host machines network interfaces could also bridge to any *veth*.



```
10.0.4.0/24 via 10.0.2.2 dev eth2  proto zebra  metric 20
10.0.6.0/24 via 10.0.2.2 dev eth2  proto zebra  metric 20
10.0.7.0/24 dev eth3  proto kernel  scope link  src 10.0.7.1
10.0.1.0/24 dev eth1  proto kernel  scope link  src 10.0.1.1
10.0.2.0/24 dev eth2  proto kernel  scope link  src 10.0.2.1
10.0.3.0/24 via 10.0.1.2 dev eth1  proto zebra  metric 20
10.0.8.0/24 via 10.0.1.2 dev eth1  proto zebra  metric 20
10.0.9.0/24  proto zebra  metric 30
        nexthop via 10.0.1.2  dev eth1 weight 1
        nexthop via 10.0.2.2  dev eth2 weight 1
10.0.10.0/24 via 10.0.1.2 dev eth1  proto zebra  metric 20
```

Figure 2.9: Example of CORE-GUI application; Currently showing IPv4 Routes widget from node N11 on runtime

CORE proposes the scalability on the size of topology with the possibility of distributed emulation. From one CORE-GUI controller, it is possible to run experiments where some of the nodes are running in different machines and the links between this two nodes will be seen as a dashed line (Figure 2.10). In the figure, all nodes have a machine hostname as the name prefix (*sky*2 or *sky*4) that shows where the nodes are emulated.

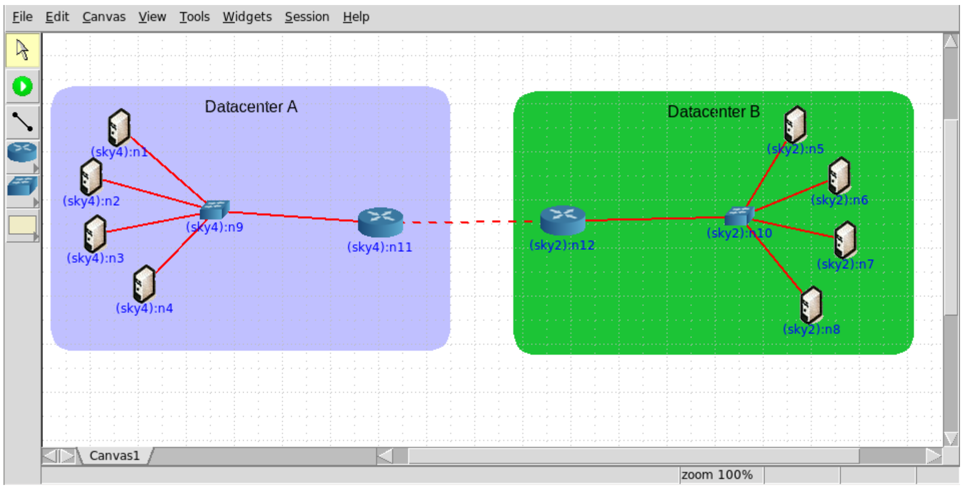


Figure 2.10: CORE Distributed emulation. GREP is used for connection between different emulations

Chapter 3
Apache Storm on multi-cloud environment

Deploying Apache Storm in multiple clouds has its own challenge. Apache Storm systems are usually deployed in one location, i.e. single cloud or data-center. This is because the nodes in a single data-center are connected each other by high bandwidth network connections. This deployment ensures a high performance because a big-data stream processing is desired to be scalable and highly parallelized. Stream processing is also expected to generate high network traffic as the Tuples are processed by multiple computation nodes which can be located on different nodes.

This chapter focuses on exploring Storm capability to be able to run the system in a distributed network environment with multiple data-centers or cloud instances. First we explained the motivation of why we should use a distributed Storm in network-on-the-edge, based on current trend of Geo-distributed sources. We generalized different Edge Cloud model into single model called multi-cloud and then find the best way to deploy Storm components in this model. We also have a hypothesis of what will be the performance bottleneck in this type of Storm. Therefore, we present our experiment to attest the correctness of our hypotheses.

3.1 Multi-cloud environment for Geo-distributed sources

In this era, modern data sources are essentially automatic, distributed, and continuous [16]. While some research focused to gain more information by increasing the data source rates, another factor that creates higher throughput is because the numbers of the object emitting data are also rapidly increasing. User mobile phones, wearable devices, environmental sensors, etc. are enhanced to be smarter, able to connect directly to their data pool via Internet by using an IP-based or mobile network connection. The sources can be located in multiple buildings, cities, regions, countries, or any specified location. This concept creates the term of geographically distributed (Geo-distributed) data sources.

When the data source is Geo-distributed, finding best data-center or cloud location to run latency-critical or real-time application is a challenging problem. Deploying an application in a single location may not satisfy the response time needed on different place. The same reasoning also applies when using a single cloud services that located across high-latency network (Internet) where intangible latency will appear. We are looking for an approach to collect the data and perform the computation locally based on the location of the sources. Data from any location will be computed in the closest cloud, which makes same latency and response time regardless of the location.

Faced with this problem, we are looking at the new concept presented by several research where the physical network devices located closer to the sources can be modified to deploy a cloud application known as Edge Cloud, explained in Chapter 2.2. This concept is in accordance with our need where the cloud location is also distributed by the location of the sources. Figure 3.1 gives the example of distributed Cloud-RAN connected to each other.

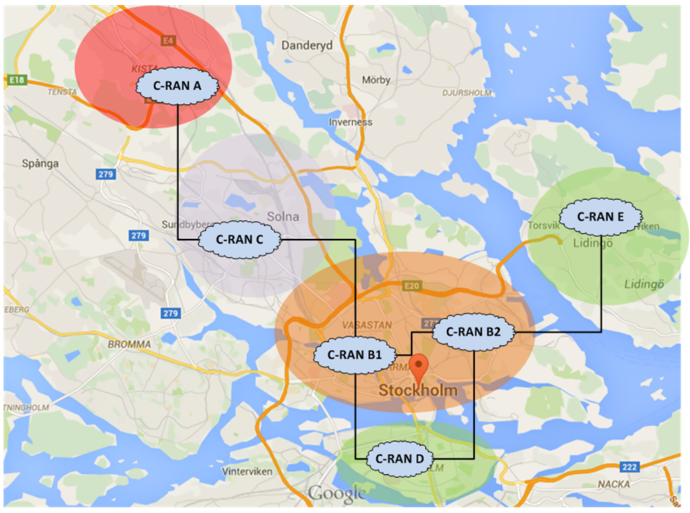


Figure 3.1: Sample of distributed Cloud-RAN between different Stockholm area, some crowded area can have multiple C-RAN instances

Each Edge Cloud model has different network topology and how the clouds are connected to each other. But, aside from the differences we are focused on the concept that Edge Cloud itself is a set of multiple clouds that are scattered in different location. We generalized this concept into **multi-cloud** model. The similarity of different multi-cloud model is the clouds are able to communicate with each other via their own network (ex: private telecoms network) or high-latency network (Internet). This generic approach of multi-cloud model is also assumed to be able to run any application particularly Apache Storm or any stream processing framework. By generalizing different network topology into single model, it will be easier to address the features and problems when deploying Apache Storm in the next section.

3.2 Apache Storm on Multi-cloud

Based on the model of Edge Cloud discussed in previous section, we found two different way to deploy Apache Storm on multi-cloud model. The first way is to deploy different Storm instances on each cloud. In this deployment, all clouds will have its own computation and management components. The second way is to deploy a single Storm instances for all the clouds. In this deployment, the components are distributed into multiple clouds or sites. We will discuss both advantages and disadvantages on each deployment.

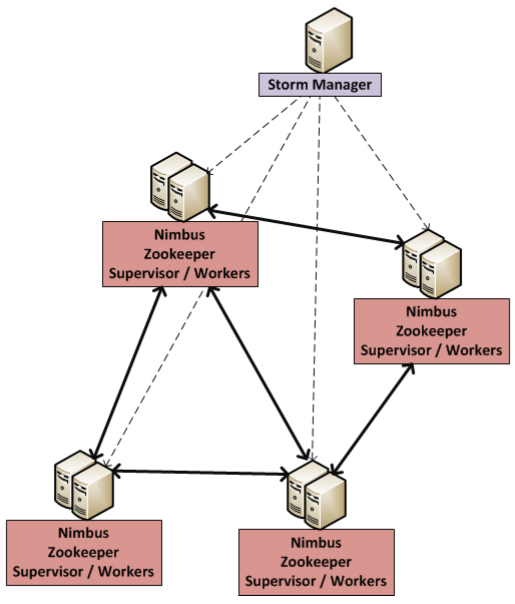


Figure 3.2: Sample deployment of multiple Storm instances in multi-cloud. Third-party server is needed to manage this deployment

3.2.1 Integrated Storm instances

Deploying multiple Storm instances in multi-cloud environment have the advantage of component robustness. In this deployment, the communication between the Workers to Nimbus and Zookeeper occur on each cloud. This design avoids sending the heartbeats between the cloud where the communication can be unreliable. Any fault tolerant or scalability process also handled on each cloud. Another advantage of deploying stream processing in this deployment is the computation result in a cloud will only base on stream that coming to that cloud. This effect of result locality is one of the important consideration later on.

To manage the deployment of multiple Storm instances, we need a Storm manager that is able to control all of Storm components on every cloud. This manager should be able to add / remove Storm components, deploying Storm Topology, monitoring, and handles different clouds (different performance / cloud provider).

Furthermore, when we want to combine the process from multiple Storms, then there is a need to make some adjustments from the basic Storm usage. In the Topology deployment phase, the user or the third-party storm manager needs to send the Storm Topology to multiple Nimbus component on all clouds. Result streams from Every Storm will need to be sent directly to other Storm instances to continue the process, or pooled into an intermediate location. This means we need another message broker system to be able to collect all results. If the computation need to be done in multiple clouds (multiple Storm instances), there is a need to control all of the stream traffics from one cloud to another. Using a third-party system rather than direct communication between the Workers could create a high possibility of system bottleneck.

3.2.2 Centralized single Storm

Another way to deploy Storm instances in multi-cloud model is to only use single storm instance in the whole system. Rather than having management component (Nimbus and Zookeeper) in each cloud, the Storm components are located in different place, as can be seen in Figure 3.3. For example, the Nimbus and Zookeeper are deployed in a cloud that has high bandwidth and good latency to other clouds and the Supervisors are deployed in every other clouds.

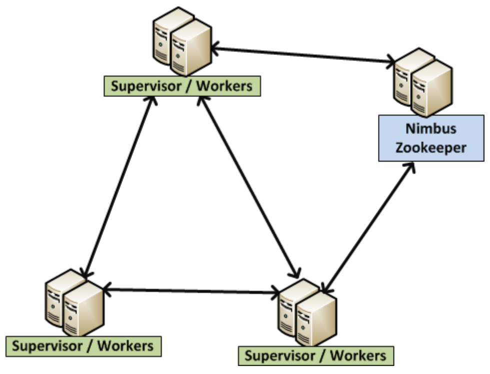


Figure 3.3: Sample deployment of single Storm with distributed components in multi-cloud. Cloud with Zookeeper act as manager for other clouds

This deployment does not need to have any third-party manager like the previous deployment. As long as the Supervisors in all clouds are able to communicate with the Zookeeper in other cloud, the Storm will be working normally like a single data center deployment. Even though the physical machines are located in different sites, communication and data transfer between Worker nodes are controlled by the Storm as long as both Supervisor nodes are also able to communicate. There is no need to have a third-party system to handle the inter-cloud messages. Another advantage with a single Nimbus administrating the whole system, the user are able to create and design more complex topology and utilizing different cloud for different roles.

The major problem in this type of deployment is the unreliable inter-cloud communication. This Storm must be able to handle any possibility of inter-cloud latency or bandwidth limitation that could affect the system availability and ultimately, its performance.

Based on both advantages and disadvantages, we choose the centralized Storm rather than multiple Storm instances. Multiple storm instances are easy to manage, but need lots of modification from the view of the administrator who is managing the instances and changing the Storm user experiences because they need to distribute the query rather than creating a single Storm Topology. On the other side, single Storm management where the components are distributed needs minimum or no changes from the default Storm release. The user or any view from the outside still see the system as a single Storm instances while behind the curtain the components are geographically distributed. However, the study of cloud network heterogeneity and inter-data center connection will affects the components availability and overall performance should be studied further. This will be our focus in the next section.

3.3 Storm deployment in data-center with Heterogeneity network latency

In this section, we are exploring whether the heterogeneity of inter-data center connection inside multi-cloud Storm deployment will affects the performance and availabilities of Storm components or not. There are two evaluation scenarios deployed inside CORE network emulator:

- First, a configuration of 5 different cloud locations (different network subnet) with 2 machines / nodes on each network. All clouds are connected to each other via core or back-haul network. In total there are 10 machines emulated.

- Second, an emulation of 50 community network nodes using the information from Guifi.net community network[3].

3.3.1 System configuration

Every nodes and network links are created inside CORE network emulator in single machine. At first, the writer attempted to run CORE with distributed emulation to get better load balancing for the emulated nodes. When the writer start running the experiment in distributed environment, there is a problem of packet limitation size of GRE tunnel between multiple CORE daemon. After some workaround, the writer chooses to run the experiment in single machine. The machine specification are explained in Table 3.1 below:

Table 3.1: Hardware specification

Component	Specification
Brand	HP ProLiant DL380 server
Processor	2 x Intel Xeon X5660, 24 threads total
RAM	44 GB
Storage	2 TB
Operating System	Red Hat Enterprise Linux (RHEL) 6

Every virtual node has one instance of Apache Storm running, except one node that hosts Zookeeper instance. Each node with Supervisor component will only host a single Worker to make sure the Task are distributed between nodes, not between Workers inside a single machine. Every Worker are set to reserve 512 megabytes of memory. This amount is expected to be able to be provided on each node.

3.3.2 Test case

Test case used on this experiment is based on the idea of yahoo performance test benchmark topology for Storm Netty[4]. The benchmark was initially created for testing the performance of Netty IO framework[6] for inter-Worker Tuple movement, changing the previously used ZeroMQ on Storm version before 0.9.0. This Storm topology focused on the movement of the Tuples inside the network without any process occurred on each Bolt. This is related with this experiment where we wanted to see the effects of latency on different part of the network and ignoring the computation or process happened on each tasks.

The design of the topology can be seen in Figure 3.4. Based on user input, the topology will create 1 type of spout and N level of processing bolts. Both spout and bolts will be then parallelized by the amount of user provided. For example, number of parallelization on both spout and bolts in Figure 3.4 is 3. Processing flow in this topology is started by each Spout creating Tuple with a certain size and rate specified by the user and sent to the Level 1 Bolt (L1_Bolt). Every Bolt on each level will send the received Tuple to the Bolt with a higher level until it arrives on the last level N Bolt (LN_Bolt). As there are no computation happening on each Bolt, this test case is suitable to measuring the latency of the Tuple movements in the network.

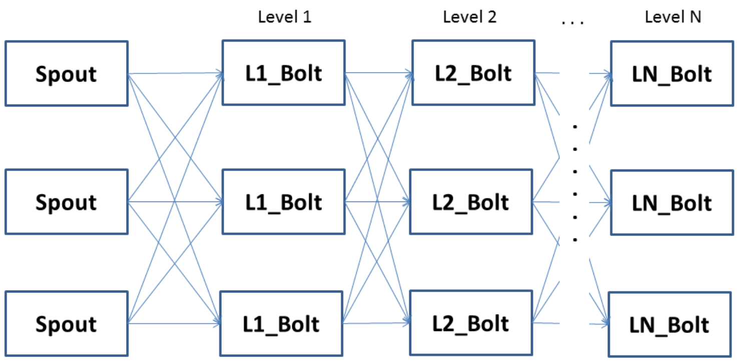


Figure 3.4: Netty performance benchmark Storm topology

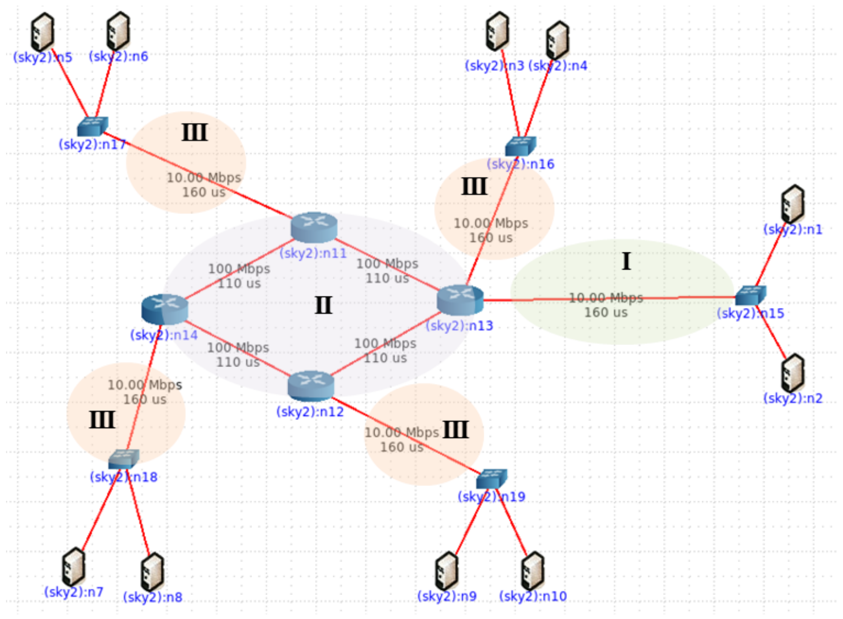


Figure 3.5: Network topology for experiment 1, The three circled areas are the location of the given latency for each case

3.4 Evaluation on multiple data-center / different network subnet

The focus on this experiment is to analyze the heterogeneity of network latency that cloud affect the availability of the Storm components and the system performance. Different amount of network latencies are placed in different location on the network links shown in Figure 3.5. The experiment is divided into four different cases. Each case has different focus on which links are affected by some amount of latency: No latency in the whole system (base case), between the computation nodes to the management nodes (Nimbus and Zookeeper) (I), latency on the central network that is connecting the data-centers (II), and where each data-center has different latencies when connected to the other data-center (III).

3.4.1 Case 1: No latency

This first case is the baseline for the other case where there are no latency applied on all links. This setting can also be seen as data-center deployment where all of the nodes are connected via high-speed connection without legible latency.

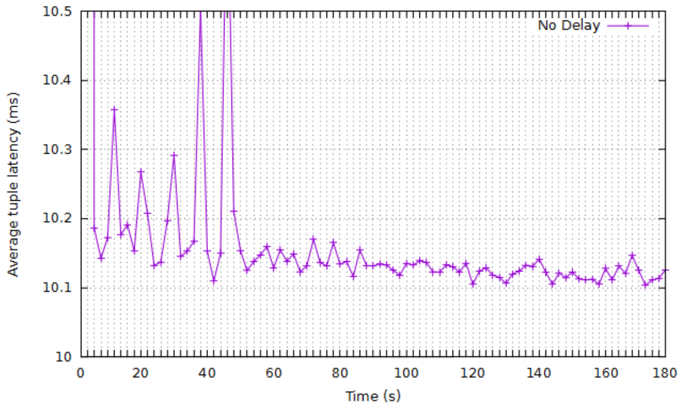


Figure 3.6: Case 1: Average tuple processing latency for every 2 seconds period

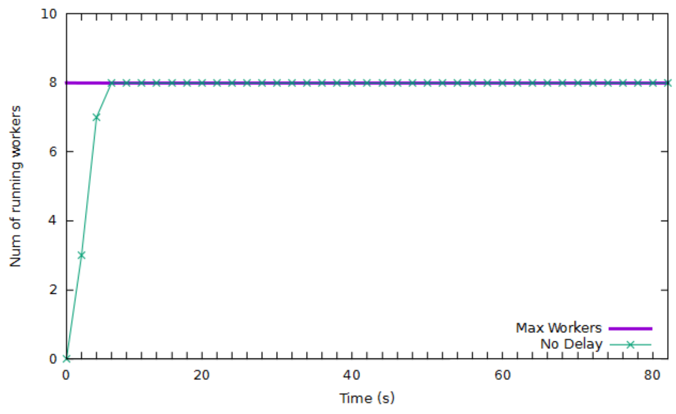


Figure 3.7: Case 1: Number of Workers running the topology

Figure 3.6 explains how long it takes to process Tuple from the time it is generated in the Spout until arriving in the last Bolt. Every step in the x-axis shows the average processing latency of the Tuples on every 2 seconds period. In the first minute, the Tuples still have unstable high peak latency because some Tuples are dropped and restarted while new Workers and Executors are still being spawned on each Supervisors. After the first minute, the average latency stabilize to around 10.15 milliseconds. Figure 3.7 shows the number of Worker processes spawned by Supervisor and registered in the Zookeeper. After 10 seconds all Workers are running and ready to process the stream.

3.4.2 Case 2: Latency on management nodes

In this Case, we are adding some latency in the network link connecting management nodes with the remaining data-centers where the Supervisors is located. There are three amount of latencies tested: 15, 30, and 45 milliseconds. The result is then compared with the base case without latency (Case 1).

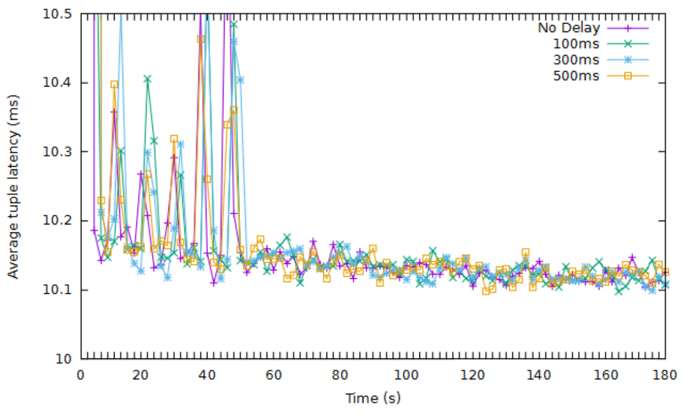


Figure 3.8: Case 2: Average tuple processing latency every 2 seconds period

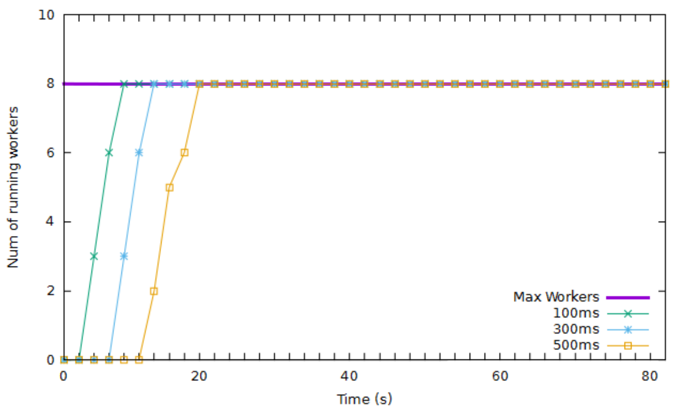


Figure 3.9: Case 2: Number of Workers running the topology

With the modification of latency to Nimbus and Zookeeper with the rest of Supervisors, Figure 3.8 shows that there are no effects on the Tuple process latency. No visible changes happened even after the latency set into 500 ms for each heartbeat from Supervisors and Workers to the Zookeeper. In Figure 3.9 it is apparent that the startup phase of the Workers is expected to have some delay for higher latency because of slower communication with Nimbus. 500ms latency created twice as much startup phase time compared with 0ms latency.

3.4.3 Case 3: Latency on the central network

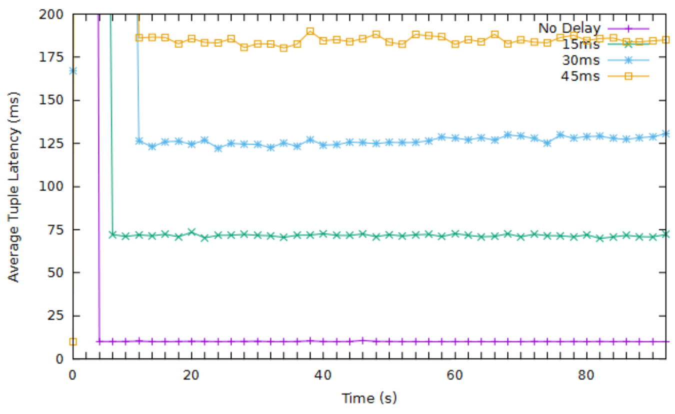


Figure 3.10: Case 3: Average Tuple processing latency every 2 seconds period

Latency in case 3 is located in the network between the routers connecting the clouds. This case is based on the idea of each cloud is connected into high-latency network. There are three different latency run for this case: 15, 30, and 45 milliseconds. Connection between nodes that has higher number of router hops will produce more latency for each packet or Tuple sent.

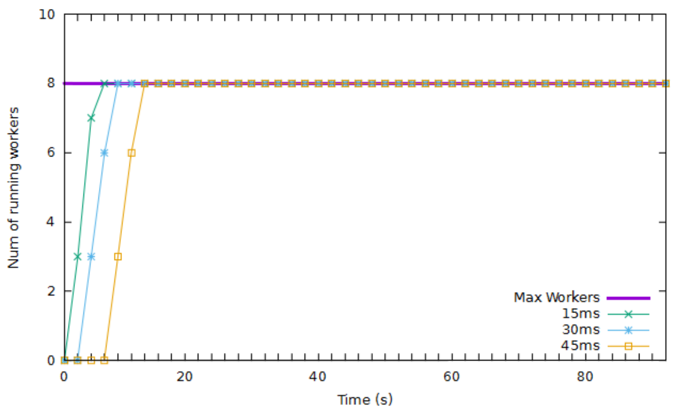


Figure 3.11: Case 3: Number of Workers running the topology

From Figure 3.10, increasing latency in the core network creates significant effect for the average Tuple latency. By the addition of 15 milliseconds latency, the average of Tuple latency suddenly increased by 7 times from 10 into around 75 milliseconds. The Tuple processing time kept increasing until 18 times higher than normal for 45 milliseconds core network latency. This shows that moving Tuple between clouds will affect the performance and should be minimized. Slower starting phase has the same reason with Case 2 where the communication between Workers and Nimbus are taking more time as the latency increased (Figure 3.11).

3.4.4 Case 4: Latency on cloud nodes

Last case in this experiment is to analyze the effect of combining clouds that have high and low latency at the same time. There are 4 types of latency placement: 30 milliseconds latency into 1 data-center (n9 & n10), 2 data-centers (n9,n10,n13,n14), 3 data-centers (n5,n6,n9,n10,n13,n14), and all data-centers.

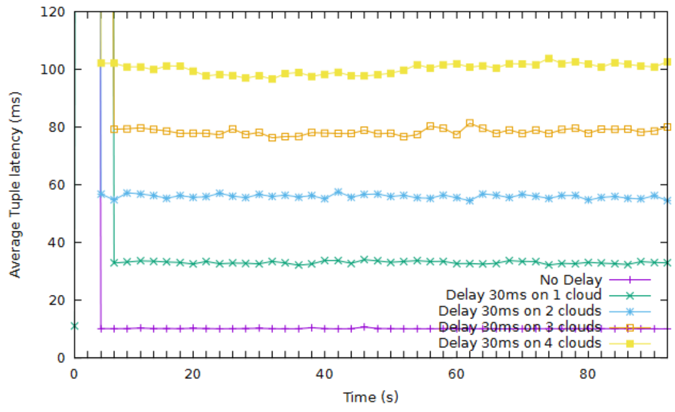


Figure 3.12: Case 4: Average tuple processing latency every 2 seconds period

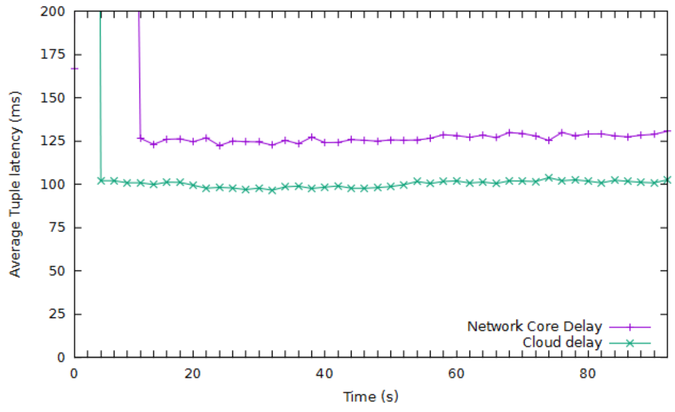


Figure 3.13: Comparison of 30ms latency from Case 3 and all clouds in Case 4

As we can see in Figure 3.12, when each cloud is introduced to a latency one by one, the average processing time increased by around 20 milliseconds. The consistent increment is expected because the Storm default Scheduler is based on round-robin load balancer. While the scheduler will distribute same amount of Tuples on each bolt, the bolt in higher latency cloud will need longer time to process rather than low latency cloud. This will affect the whole system because the normal speed bolt in low latency cloud will they need to wait Tuples from slower bolt on high latency cloud. With higher number of high latency Cloud, the average Tuple computation time will also increases.

3.5 Evaluation on Community Network emulation

In this section we are observing Apache Storm performance on community network cloud. The concept of community network cloud are explained in chapter 2.2.3. This experiment has been published in a research paper with title of "Stream Processing in Community Network Clouds" and presented on August 2015 [17].

Figure 3.14: Community network nodes topology

From the previous discussion, community network cloud can be considered a multi-cloud deployment. The physical devices where the cloud resources are located are distributed and connected via unreliable network. Some resources can be located close (physically or good connection) together and create a high connectivity cluster, as can be seen in Figure 3.14. Each physical device can connect to end-devices like user PCs or wireless sensors to collect the data. This is similar with how the geo-distributed sources are explained before. The challenge to deploy Apache Storm or any stream processing on this type of cloud is its network characteristics. Community networks have a very heterogeneous link quality, especially latency and bandwidth, which makes it important to have different consideration compared to deployment on a data-center.

In this experiment, we are looking at how the different location of the Storm components inside the community network cloud will affect the performance. First, we evaluate how the different placement of Nimbus and Zookeeper affects Storm start-up time to schedule the tasks to the Supervisors. We also observe the stability of the node connection to the Zookeeper while the process is running. Second, we evaluate the behaviour of Worker components based on node connectivity and two types of Storm stream Grouping: Shuffle and LocalorShuffle Grouping. We make an assumption that each node will be able to host at least single Storm instances (Nimbus, Zookeeper or Supervisor).

The sample topology of the community network used in this experiment are collected from small part of Guifi.net[3] on the area of QMP Sants-UPC in Barcelona, Spain. Before emulating the network topology to CORE network emulator, we manually filter nodes that are disconnected, dead, or did not have monitoring information available. In total we are able to collect and run 52 nodes with 112 network links.

Table 3.2: Range of network link quality for community network emulation

	Latency (ms)	Bandwidth (Mbps)
Maximum	84.3	91.6
Minimum	0.31	0.12
Average	3.06	31.9

To emulate network links, we use the nodes and network data collected from the monitoring system for 24 hours. Then we create an estimation of the link quality by calculating the average bandwidth and latency on each link. This information makes the evaluation have different focus from previous section because latency and bandwidth limit are fixed by the collected data. Ranges of the link quality are presented in Table 3.2. While the average latencies of the links are considered good (3 milliseconds), some links suffered with maximum latency of 84 milliseconds. Similar to the condition with bandwidth limitation, some nodes have a very limited bandwidth with less than 1 Megabytes per second. Some part of the network can suffer if traffic from Stream processing is more than the available bandwidth.

Storm Topology used in this evaluation are the identical with the previous section, Yahoo storm-perf-test benchmark (Chapter) with three levels of Bolts. In the first time we tried to run Storm on this network, the system did not work. The problem is a connection timeout error between the Supervisors and the Zookeeper. We expected this problem to happen because the links have a very different quality. To overcome the effect of bad connection, we need to modify Storm configuration file (Storm.yaml) to increase timeout time and reduce heartbeat rate. The modification is presented in Table 3.3. We increased some of the configuration values that controls the fault tolerance by $2,5$ times from default by trial and error. When the values are set to $2,5$ times higher, the majority of the Workers can connect to the Zookeeper and start processing the stream.

3.5.1 Placement of Management components

Management components have two main duty: To deploy Tasks into each Supervisor nodes and maintaining status of all nodes whether they are still alive or not. Even though the bandwidth used in their communication is small, but bad connection could create numerous false-positive node state. We are looking at whether the placement of both Nimbus and Zookeeper are the crucial factor or not to consider in this type of cloud.

Table 3.3: Storm.yaml configuration for the experiment

Parameters	Storm Default value	Modified value
Worker Heartbeat frequency (secs)	1	10
Worker timeout (secs)	30	80
Supervisor heartbeat frequency (secs)	5	20
Supervisor timeout (secs)	60	150
Nimbus task timeout (secs)	30	80
Nimbus monitor frequency (secs)	10	40
Zookeeper session timeout (milisecs)	20000	50000
Zookeeper connection timeout (milisecs)	15000	40000

We categorize the nodes based on their degree of connectivity. This categorization is created to consider two different locations of the nodes. A node with 5 or more direct connection is called SuperNode; whereas a node with less than 5 connections is called EdgeNode. The SuperNodes are able to connect to any other nodes easily because there are lots of connections available. On the other side, EdgeNodes is a remote nodes with few connection and need to rely on other node with limited connectivity. According to our categorization, 52 nodes presented in Figure 3.14 has 22 SuperNodes and 30 EdgeNodes.

From all of the available nodes, we choose 30 nodes to become worker nodes and run Supervisor instance. The Supervisor nodes are selected so the Supervisor instances will be spread evenly in the whole network based on Figure 3.14. Supervisor locations are permanent because we want to focus only on the different placement of Nimbus and Zookeeper for each run. Nimbus and Zookeeper are located in single node different from each other. In total there are 32 nodes running Storm components. Using 32 from 52 available nodes seems to be able to represent the network.

Table 3.4: Management Nodes location on each run

Run ID	Nimbus	Zookeeper
SuperNodes		
Run-1	n47	n50
Run-2	n20	n21
Run-3	n44	n48
EdgeNodes		
Run-1	n2	n3
Run-2	n27	n31
Run-3	n5	n6

For a total of six repetitions, we put the Nimbus and Zookeeper on both type of category (EdgeNodes and SuperNodes) three times each. Runs on the same category are located in different nodes. The location of Nimbus and Zookeeper are shown on Table 3.4.

Figure 3.15 and 3.16 show a detailed view on state of tasks in each run for different placements of the management components. Tasks are considered "running" when the process are created in the worker node and able to receive / process a Tuple. "Max Executors" is the total number of Tasks that should be running. Tasks that located in the node with good connection quality will achieve the "running" state faster than the Tasks that located in bad connectivity nodes. On Figure 3.15, there are a lot of unstable tasks that keep on disconnecting after some time. In the example of Run-2, the system are unable to register all of the Tasks to the zookeeper even after 140 seconds. If we continue following the graph time, in the end Run-2 is able to reach the max Executor line after 214 seconds / 3.5 minutes, while some nodes keep disconnecting all the time. On the other hand, Figure 3.16 shows us a very stable result compared with Figure 3.15. Just by placing the management nodes in high connectivity nodes, we will obtain better stability. Some Tasks take longer time to register themselves because they are located very far from the management nodes.

Figure 3.17 display the average scheduling time of every run. Scheduling time is the time required for all Tasks to achieve "running" state. In the default Storm instance, usually scheduling time is called once when the user deploys Topology for the first time. The scheduler are also called when the system reached the ’Rebalance’ state. In the complex and resource-aware scheduler created by Aniello et al.[11], the Tasks are often need to be moved from one Worker to another. The reason is because the scheduler will find the best Task allocation that fulfills the parameter provided (ex. less traffic or highest Tuple rate per second). As the scheduling time will be recalled more often, it is important to make this scheduling time a consideration when deploying the Tasks in multi-cloud environment.

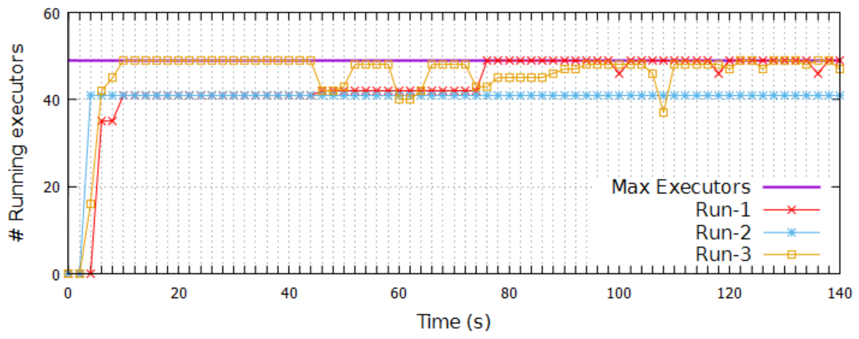


Figure 3.15: Number of tasks running at run-time. Nimbus and Zookeeper located on EdgeNodes

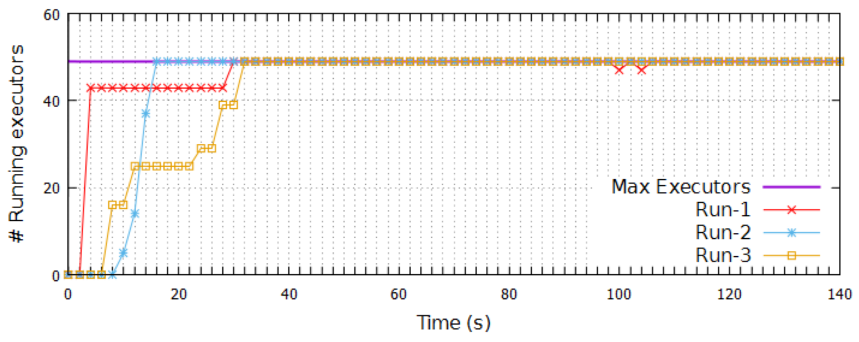


Figure 3.16: Number of tasks running at run-time. Nimbus and Zookeeper located on SuperNodes

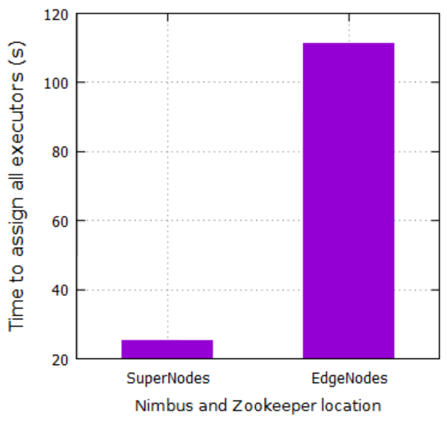


Figure 3.17: Average time until all tasks assigned to workers and acknowledged by the Zookeeper

3.5.2 Worker nodes placement

Inside the community network, sources are distributed on all the nodes. Assuming that we know on which nodes the sources are located, our idea is to allocate Storm Tasks on those nodes. This concept is quite different from deployment on the data-center where raw data are usually pooled into a message broker system such as Kafka or no-SQL databases such as Hbase or Cassandra.

In this second experiment, we choose 29 nodes to serve as Worker nodes and 2 static SuperNodes location for Nimbus and Zookeeper. On each Worker nodes we put single Spout Task to generate the Tuples. In each placement, we also select 10 from 29 Worker nodes to host the Bolt Tasks. In the end, there are 10 nodes with collocated Bolt and Spouts Tasks, and 19 Nodes with only Spout Tasks.

We are measuring the amount of in-out network traffic by placing Bolt Tasks in different location. We capture the network traffic of 52 nodes using Linux *ifstat* utility. This information allows quantifying how Stream Processing will affect the whole network, not only the nodes that host the Storm components. There are two type of Bolt placement: Random Placement and Cluster Placement. In Random Placement we choose 10 from 29 nodes with Spout Task randomly. However in Cluster Placement, we choose 10 SuperNodes that have high connectivity with each other to form a cluster of Bolts.

Figure 3.18 shows the average network traffic for two different placements. Bolts in Cluster Placement generate 30% less traffic compared with Random Placement. This is because the network traffic in Cluster Placement are circulated within the cluster, in contrast Random Placement makes the Tuple need to travel further through higher number of network hops between Bolts, and passed some nodes that did not run any Storm component.

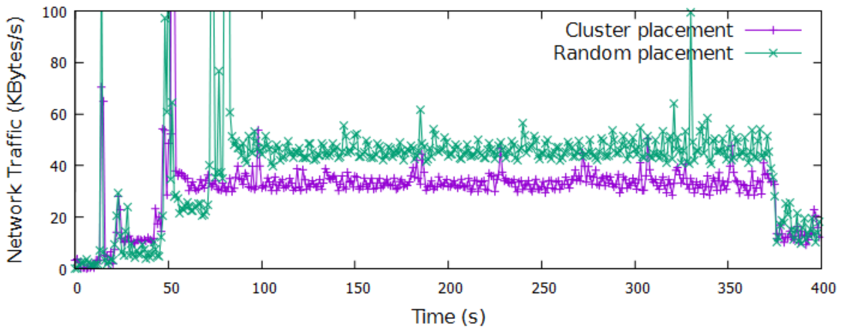


Figure 3.18: Average nodes traffic for different Bolt placement scenario

Second thing to explore is to see the effects of different Storm stream Grouping mechanism to the amount of network traffic generated. We compared Shuffle Grouping and LocalOrShuffle (Local) Grouping because of their similarities (Explained in Section 2.1.1). The advantage of Local Grouping is a Task will send the Tuple to the next processing Bolt that located in the same Worker with the sender. This Grouping are expected to do local communication as much as possible and did not affect the network traffic. The result are shown in Figure 3.19 & Figure 3.20. In both figures, the Shuffle Grouping method has a slightly higher traffic than Local Grouping, but did not create a significant difference. This is because the number of Worker hosting Bolt Task is significantly (around $\frac{1}{3}$) lower than number Worker hosting Spout Task. Therefore, there are only a few Bolts doing the Local Grouping whiles other Spout only Workers will still send the Tuples with Shuffle Grouping.

While Local Grouping is promising to create a locality and reduce inter-Worker traffic in data-center deployment, this Grouping is not very useful for a Geo-distributed deployment with different number of parallelization between the Tasks and heterogeneous network environment. If we try to distribute the Bolts and Spouts as close as possible to location of the data sources, we should consider the balance between the numbers of Bolt and Spout Tasks. If the parallelism factor for bolts is equal or close to the parallelism factor for spouts when using the local-or-shuffle grouping, more data can be processed locally to reduce the network traffic. On the other side, if the parallelism factor of bolts is less than the parallelism factor of spouts, then stream processing consumes less computing resources; however, the positioning of the bolts becomes important to reduce the inter-Worker communication. This problem is the motivation to create a better scheduler and stream Grouping that is able to consider these kind of environment.

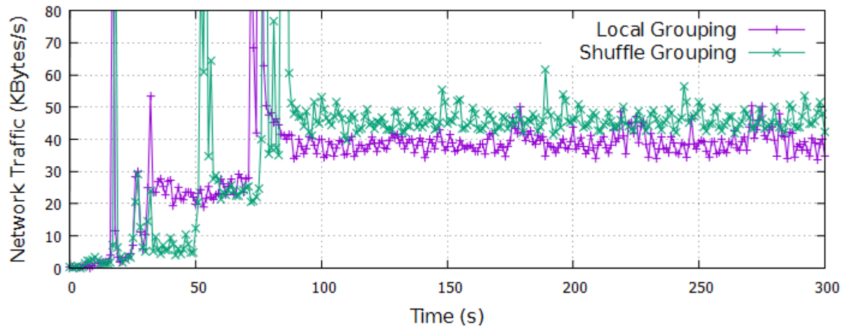


Figure 3.19: Average nodes traffic for Shuffle and Local grouping. Bolt tasks assigned randomly between the available worker nodes

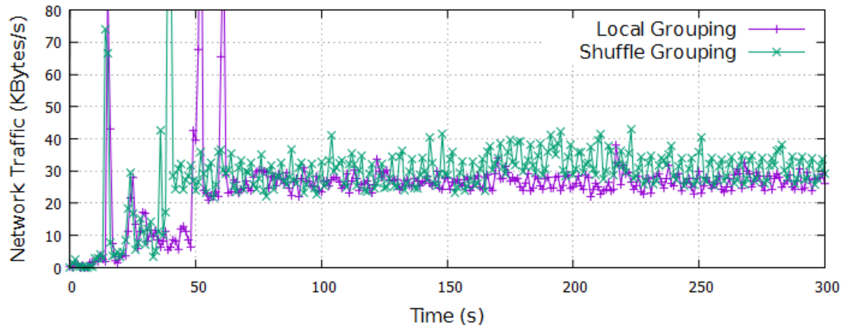


Figure 3.20: Average nodes traffic for Shuffle and Local grouping. Bolt tasks only assigned on Cluster of SuperNodes

3.6 Discussion

The experiments in this chapter is focused on evaluating Apache Storm performance and components availability when deployed in multiple data-center with heterogeneous network quality. We focused on different latency and bandwidth limit on each network links. From the result of both experiments we could get some insights to be put into consideration to create a more efficient Task deployment.

First, the delayed or high latency on the network between management components (Nimbus and Zookeeper) and Supervisors in general only affects the process startup time. When the topology is running, Zookeeper only need to receive small heartbeat messages from the Supervisor nodes and similar with Nimbus for monitoring purposes. So the high latency even until 500ms tested (1 second for round-trip) are not affecting the system. But in case of very high latency and small bandwidth like in the second evaluation, using default Storm configuration will result in many disconnection between Zookeeper and Supervisors. Some workaround in the configuration is able to handle these issues with the drawback of fault tolerance.

The other important result is the effect of Heterogeneous latency between data-centers with Supervisor/Worker being able to affect the Storm throughput. A single area that perform slower than the others (straggler) could reduce the performance of the whole system. This is because the default Storm is focused only on balanced load, both on Tasks deployment (scheduler) and stream distribution. The effect of the straggler can also be amplified if their computation rate is slower than the rate of the incoming stream which creates a queue.

There also a need to make the scheduler to be aware on the submitted topology. Deploying an adjacent Tasks in the same data-center or close to each other could have less inter data-center communication which reduce the network traffic and bandwidth consumption. Furthermore, doing some part of the computation inside a single data-center could also produces faster partial result.

Chapter 4

Geo-Distributed Apache Storm design

This Chapter is focused on creating a better way to deploy Storm components in multi data-center environment. By analysing the results from previous chapter, we seek a different way to deploy the Storm components. We introduce our concept of Tasks scheduling that fulfill the focus of our work; Real-time stream processing where data sources are geographically distributed. We are looking at a Storm Topology that have hierarchical process or multiple result stages. Also, we are introducing a new stream distribution for Storm (Stream Grouping) to support the needs of intra data-center communication and reducing network traffic.

4.1 Real-time Storm Application in multi-cloud deployment

In the experiment discussion section of previous chapter (Section 3.6), we already pointed some problem that happened when the Apache Storm components are distributed in multiple data-centers with different location, especially if there are some latency or bandwidth limitation on network links between the data-centers. To improve performance, there is a need to modify some part of Apache Storm to be able to avoid those problem.

If we use default Storm scheduler designated for single data-center, the major problem occurred will be the uncertainty of computation time for each Tuple. This happened because the scheduler do not have location awareness for every Tasks that actually process each Tuple. A Task could receive a Tuple from the same machine, different machine on the same data-center, or from a different data-center. A real-time or latency sensitive stream processing performance will suffer with this uncertainty. To satisfy the necessity of real-time, every data sources should be processed with the same amount of time regardless of its location. With the assumption of the system knows the location of the data sources, we are trying to approach the problem by deploying the Tasks in the same place or in the closest data-center as possible on where the sources are collected. This creates a multiple partial processing on each data-center location.

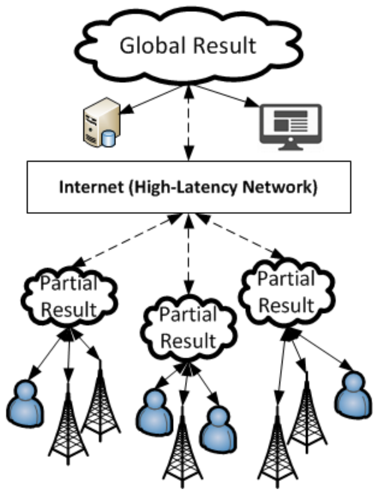


Figure 4.1: Hierarchical computation with multiple result stages.

Processing Geo-distributed data sources result partially in multiple data-centers does not mean we are abandoning the current cloud deployment. The partial result from many processes can be aggregated to the 'central' cloud to achieve result in the global scale. The generalized concept of partial results and global result are shown in Figure 4.1. Usually, in the case of processing Geo-distributed data sources, there is no need to wait for aggregated results from every location. Instead, the users or objects can act or create decision based on partial result from their location. For example, an accident information from car traffic data in different cities will only be needed on the location where it happens. In this case, we only consider the part that needs the real-time result is only the location-based partial result, where the global result is more tolerant to any delay.

The combination of partial and global computation creates a hierarchical partial-global process. For example, the global process can assign the centralized cloud to focus on slower stream processing like batch aggregation, database logging, or publishing interval results while the partial process located close to the data sources are able to process latency-sensitive computation that gives real-time responses or results. This design will provide the heterogeneity combination of both high-performance clouds and Edge Clouds that have less performance.

To implement this hierarchical process, we need to create some modification on how to distribute the Task and controlling the data stream flow as discussed above. In the next section, we explained our new Storm Scheduler, Geo-Scheduler and StreamGrouping that is suitable for the hierarchical process.

4.2 Scheduling and Grouping

4.2.1 Current scheduler and grouping

Storm default scheduler and Grouping is based on the idea of load balancing. The default Scheduler will try to distribute all of the Tasks where in the end each Worker will have the same amount of Task. While in the concept or data stream distribution, the default ShuffleGrouping works by sending Tuples equally to all destination Tasks. This is working well in a single data-center deployment to create a balanced load and network traffic. In multi-cloud deployment, we cannot rely on this load balanced concept because we are expecting the scheduler to have more awareness on each Worker location and deploying the task on the designated location.

A cloud is assumed to be able to run Storm instances. Each cloud can run one or more Supervisor based on the number of machines available (virtual or physical). The idea for the new scheduler is to understand that the number of Workers each cloud could hosts depends on their capability. With this information, the scheduler did not need to see each Worker as an individual object, rather looking at groups of workers on each location. We named this broader viewpoint as a *cloud-level* viewpoint. With cloud-level viewpoint, a scheduler will need to distribute the Tasks into a cloud rather than directly to each Worker. Also, with this viewpoint the scheduler is able to create a distinction between each cloud to deploy a job based on its capability. The ability of categorizing the clouds is important for hierarchical Stream Processing applications.

While the new scheduler is able to distribute the tasks into multiple locations, we also cannot rely on the default Groupings provided by Storm because they do not have any location awareness. If these Groupings are used on multi-cloud deployment, we expect to get plenty of inter-data center communication. To minimize or eliminate inter-data center communication, we need to have a Grouping protocol that understands their location in a cloud and only distribute streams inside a single cloud.

4.2.2 Geo-scheduler

Our proposed scheduler is based on the multi-cloud or Geo-distributed clouds environment described in the previous section. Storm components (Nimbus, Zookeeper, Supervisors) are deployed on multiple clouds instances located on different location. We make the assumption where all of the Supervisor nodes inside a cloud can connect to other nodes in different cloud. This can be achieved for example with public IP or virtual Network (VPN) access to all nodes.

One main concept of the Geo-scheduler is to see the Storm topology from a different view. The current Storm Topology consists of Spouts and Bolts with the directed links between them to represent the stream processing flow. Based on the hierarchical process explained in the previous section, we are trying to add an abstraction layer on the top of default Topology by assigning and categorizing multiple Bolts and Spouts into different group called *TaskGroup*. The current TaskGroup implementation is focused on the topology that is producing two type of results, the partial and global result. Group of Tasks that is required to be located close to the data source or doing a low latency computation are categorized as the LocalTask group. LocalTask is highly parallelized to different number of clouds and is able to maintain the locality of the process on each single cloud even though multiple clouds are running the same Tasks. Usually LocalTask are highly related with producing partial result. On the other hand, other Tasks that are focused on the data aggregation from the result of multiple LocalTask instances are categorized in the GlobalTask group.

The example of assigning and categorizing Storm Topology into TaskGroup can be seen in Figure 4.2. First, a Spout Task must always be located in a LocalTask, which means a LocalTask will at least consist of a single Spout Task. The reason of this is because we want to pair the LocalTask to the clouds where the corresponding data sources are located. Following the concept of Edge cloud, each cloud will acts as a data pool or message broker that is able to collect raw data sources from the surrounding location. We call this type of cloud as *Source cloud*. Source cloud are able to host one or multiple types of data sources. This makes it possible to put different LocalTask on a single Source cloud, as is shown on Figure 4.3. Every LocalTask will be automatically parallelized by any number of registered Source clouds.

To maintain the locality feature of LocalTask, Every Bolt that is located in a LocalTask will only process the Tuple stream from Spout / Bolt from the same cloud. Eventually, some Bolts will be in charge to aggregate or compute all of the local data from the LocalTask instances. These Bolt Tasks need to be registered as GlobalTask group.

While LocalTask will be parallelized and deployed in the Source clouds, placement mechanism for GlobalTask is not provided directly from outside information. The GlobalTask concept is to receive the stream from multiple LocalTask clouds (Edge Clouds) in different location. To get the best performance of GlobalTask, it is the scheduler duty to choose the most suitable cloud. The idea is derived based on paper by [11] and [20], in those paper the focus is to find the most suitable Supervisor node based on network condition and Supervisor capability. The difference in our case is we are looking at the Storm system in the data-center view, not as a single machine/Supervisor level. By specifying different locations that are able to host the GlobalTask, Geo-scheduler is in charge to find the most suitable cloud based based on any given info: Average latency between clouds, bandwidth limitation, computation power, centralized location, etc. In the current version, we made a simple decision system that is based on latency between the clouds. But this decision system part

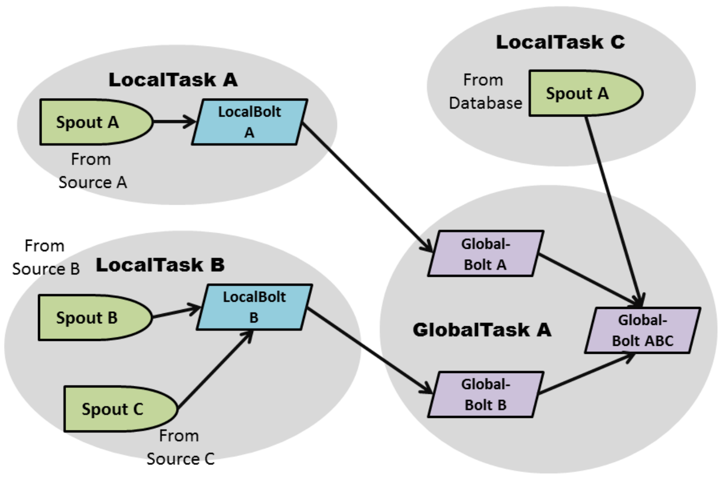


Figure 4.2: TaskGroup categorization for a Storm Topology.

of the scheduler is highly extensible for adding any variable needed to make the decision.

To support the dynamic parallelization of LocalTask to any number of source clouds, we also introduce a new parallelization level on the Storm Topology. In the default Topology, each Task has the parallelization hint value describing how many Tasks will be paralleled between the Workers. We called this worker-level parallelization. To support the TaskGroup concept that we are distributing group of LocalTask, we need to count the parallelization based on how many clouds will receive the LocalTask. This is called the group-level parallelization. In summary, Group-level is the number of clouds that receives the LocalTask, and then each cloud will receive Tasks based on the worker-level parallelization. For example, a LocalTask consists of 1 Spout, 3 Bolt_A (worker-level), and 1 Bolt_B. This means a single source cloud will receive 5 Tasks. Then, this number of Task will need to be multiplied by group-level parallelization. If there are 5 clouds in different location that need to be deployed, there will be 5 Tasks multiplied by 5 Clouds; 25 Tasks distributed in the whole Storm system.

The Geo-scheduler algorithm is shown in Algorithm 1. This scheduler needs three information to be able to run correctly: Tasks that already grouped to the TaskGroup, list of data-centers participated in the scheduling, and location of the Source cloud. In the latest version of the scheduler, it is the user responsibility to distribute the Task into LocalTask or GlobalTask. There can be many improvements to automatize this part by analyzing the user Topology as a graph problem model. The Scheduling process is divided into deploying LocalTask and GlobalTask. For LocalTask, the deployment location is based on the Spout Task and the location of the respective

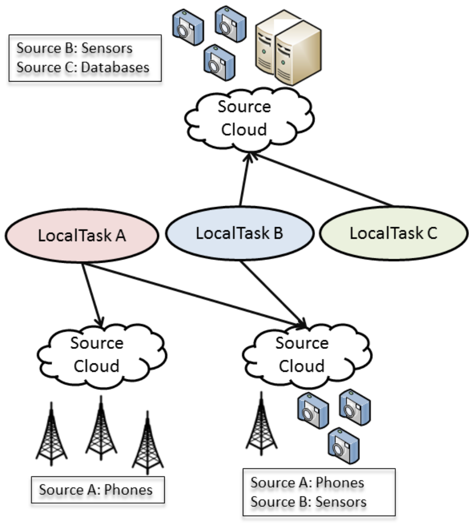


Figure 4.3: Source cloud with different type of data source. LocalTask deployed into Source cloud with corresponding source

Source clouds (line 5). When a Source cloud is discovered, all the Tasks inside that LocalTask will be assigned into that cloud. This is done for any number of Source clouds and create a parallelized deployment of LocalTasks.

To choose the most suitable data-center location for GlobalTask, there is a need to know the location of the deployed LocalTasks first. From the concept of partial-global computation discussed before, the first Bolt in the GlobalTask basically always receives the stream from one or more LocalTasks. By reading the stream information of these Bolts, the scheduler are able to find all of the data-centers location that became the dependencies for the GlobalTask (Line). The results are then saved in a list of dependencies. Based on this list, the scheduler will choose the best data-center to receive the Tuples and continue the processing from all of the LocalTasks. In the sample of Algorithm 2, we tried to decide based on highest network link values between the data-centers. The programming implementation for this part is highly modifiable to add different methods and comparison based on any available information.

4.2.3 ZoneGrouping

After the Tasks are deployed on the designated clouds, the next part that we need to do is make a modification of Storm Grouping system. The modification is needed to reduce the inter-data center communication between the parallelized Tasks. Figure 4.4 shows the nature of default Storm ShuffleGrouping where it does not handle situations where two parallelized tasks are deployed in different location. Without

Algorithm 1 Network-aware Scheduler

```
INPUT:
$\varepsilon = \{e_i\}$ $(i = 1...E)$                                          ▷ Set of Tasks
$\varepsilon_l = \{< e_i, lt >\}$ $(i = 1...E, lt = L_1, L_2, .., L_n)$          ▷ Set of LocalTasks
$\varepsilon_g = \{< e_i, gt >\}$ $(i = 1...E, gt = G_1, G_2, .., G_n)$          ▷ Set of GlobalTasks
$\theta = \{c_i\}$ $(i = 1...C)$                                               ▷ Set of Clouds
$\theta_s = \{e_s, \{c_s\}\}$ $(e_s \in \varepsilon, c_s \in P(\theta))$ ▷ Map of SpoutID with the source cloud location

 1:                                                             ▷ LocalTask placement
 2: for each L (LocalTask) $\iota$ in $\varepsilon_l$ do
 3:     get all Tasks $\{e\}$ from $\iota$
 4:     for each spout $e_{sp}$ in $\{e\}$ do
 5:         get all cloud value $\{c_s\}$ from $\theta_s$, with $e_{sp}$ as the key
 6:         Assign Tasks $\{e\}$ in all $\{c_s\}$
 7:     end for
 8: end for
 9:
10:                                                             ▷ GlobalTask placement
11: for each G (GlobalTask) $\iota$ in $\varepsilon_g$ do
12:     get all Tasks $\{e\}$ from $\iota$
13:     for each Task $e_g$ in $\{e\}$ do
14:         $\{C_d\} \leftarrow$ Get all clouds where Task $e_g$ is dependent to
15:     end for
16:     $c \leftarrow$ FindBestSuitableCloud($C_d$)
17:     Assign all $\{e\}$ in $c$                     ▷ All tasks is located in single cloud
18: end for
```

Algorithm 2 Find Suitable Cloud for global group

```
INPUT:
$V_{c1,c2}$ $(c1, c2 = 1...C)$            ▷ Connection values between cloud $c_{c1}$ and $c_{c2}$
$\theta = \{c_i\}$ $(i = 1...C)$                                   ▷ Set of Clouds

1: procedure FindBestSuitableCloud(CloudDepedencies $C_d$)
2:     $lc = NULL$
3:     $lc_{val} = 0$
4:     for Each cloud $c$ in $\theta$ do
5:         $val \leftarrow$ values between $\{C_d\}$ and $c$
6:         if $lc_{val} \leq val$ then                    ▷ When $c$ is a better cloud
7:             $lc \leftarrow c$
8:             $lc_{val} \leftarrow val$
9:         end if
10:    end for
11:    return $lc$
12: end procedure
```

any modification, each Spout Task will try to balance the load and send the Tuple to the bolt that located in different cloud.

Some possible ideas to achieve this local cloud computation is by using field Grouping or direct Grouping. With field Grouping we can categorize the Tuple to have a field of 'cloud name'. With this additional information we can group the computation of the Tuple based on cloud name, creating a locality of the result. But the inter-data center problem is still occurring because we are unable to set the location of each parallelized Task. For example with Figure 4.4, Tuples that generated in cloud A will always sent to the same one instance of LocalBolt A, rather than sending to both LocalBolt A instances. But the location of the LocalBolt A can be different from the Spout A which creates another inter-data center communication. As for direct Grouping, locality and location of the Task can be controlled. But it is not scalable solution as we need to manually set each Task sender and receiver pairs one-by-one.

Based on this problem, we created our own custom Grouping called ZoneGrouping. ZoneGrouping is the concept of controlling the stream of Tuple to be limited based on the cloud location of the sender Task. Rather than sending a Tuple to all possible recipients on many clouds, with ZoneGrouping the receiver is the only recipient that is located in the same cloud.

The basic step of ZoneGrouping is to map the location of the sender Task with the receiver Tasks. After that, the system will be able to choose the destination Tasks that reside in the same location (same cloud). The step of ZoneGrouping is shown in Algorithm 3. For a proof-of-concept we created two implementation

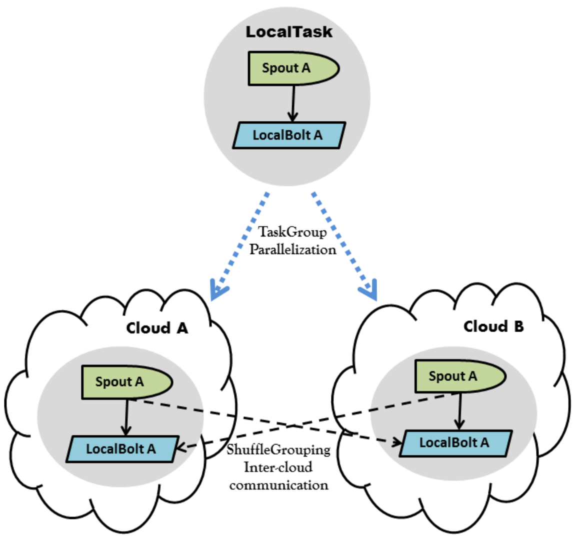


Figure 4.4: Problem with default shuffleGrouping: TaskGroup is parallelized into different cloud. The Spout will keep sending tuples to every Bolt for load balancing

Algorithm 3 ZoneGrouping abstraction

```
for Task $st$ in source tasks do
    Get cloud location $cls$ from $st$
    for Task $dt$ in destination tasks do
        if Cloud location $cld$ from $dt$ equals $cls$ then
            set $dt$ to receive the stream from $t$
        end if
    end for
end for
```

of ZoneGrouping: ZoneShuffleGrouping and ZoneFieldGrouping which works on similar mechanism with shuffle Grouping and field Grouping. The implementation of ZoneShuffleGrouping will be explained in Section 5.2.

Chapter 5
Implementation

This chapter will be focused on the technical implementation on both Geo-scheduler and ZoneGrouping as a plug-in module in Apache Storm. To create all of the modules, we use Storm extension API that is available in Java programming language.

First, we will explain the outer part of the Geo-scheduler. How to use the custom scheduler and modification needed when building Storm Topology while applying the TaskGroup concept. Then we are going deeper to the implementation of Geo-scheduler; TaskGroup Class, LocalTask and source cloud pairings, and choosing cloud for GlobalTask.

Next, we explain how ZoneGrouping could perform a stream distribution inside a single cloud only. There is also description of an information needed by ZoneGrouping that is collected from Geo-scheduler. In the end of the section, we describe how to use ZoneGrouping by using custom Grouping mechanism in Storm Topology.

Finally, we also discuss the distributed system concepts that is important to consider when implementing the module but is not yet implemented or evaluated because of the time limitation or outside the thesis scope.

5.1 Geo-scheduler

Storm scheduler is located on the Nimbus instance. The scheduler will run as long as the Nimbus instance is running, waiting for a Topology to be scheduled. By default, Storm has their own EvenScheduler class that focused on balanced Task distribution between all Workers. To add a custom scheduler, we need to add a JAR file containing the Scheduler class to Storm library folder and specifying its name in Storm configuration (Storm.yaml). An example is shown in Listing 5.1. Storm pluggable features makes the scheduler changing process very convenient without the need to recompile Storm's main source code. However, the Nimbus process needs to be restarted if we want to update to the custom scheduler.

```
...
storm.scheduler: "scheduler.GeoScheduler"
...
```

Listing 5.1: storm.yaml with custom scheduler

5.1.1 TaskGroup in Storm Topology

Assigning Tasks to TaskGroups will need to be done by the user when creating the Storm Topology. Listing 5.2 shows the addition made when creating a Task inside the *TopologyBuilder* instances. To add Spout or Bolt to a TaskGroup, we only need to add group information by using the *addConfiguration* method provided in the class *TopologyBuilder*. *addConfiguration* receives a key-value information. By specifying the key as 'group-name', Geo-scheduler will assign this Task to the TaskGroup that have the same name with the value. It is important to have distinct name for each TaskGroup.

The advantage by only using *addConfiguration* method is that the user can still use the default or other scheduler without any change on the system. Other scheduler will just ignore additional information and run Storm normally without any side effects.

```
TopologyBuilder builder = new TopologyBuilder();
. . .
builder.setSpout("messageSpoutLocal1",
    new SOLSpout(_messageSize, _ackEnabled), 4) /*4 Shows worker-level
     parallelization*/
    .addConfiguration("group-name", "Local1");
. . .
builder.setBolt("messageBoltGlobal1_1A",
    new SOLBolt(), 4)
    .shuffleGrouping("messageBoltLocal1_1")
    .addConfiguration("group-name", "Global1");
```

Listing 5.2: Spout and Bolt declaration in Storm Topology

5.1.2 Geo-scheduler implementation

Creating custom scheduler in Storm is based on implementing *IScheduler* interface on Storm-core library. *IScheduler* has two main method: *Prepare* and *Schedule*. *Prepare* will be called once when Nimbus is started and *Schedule* will be called periodically, or if there are some faults or *Rebalance* event that trigger the system to re-run the scheduler.

In the previous chapter (section 4.2.2) we already explained the concept and types of TaskGroup: LocalTask and GlobalTask. Listing 5.3 shows the implementation of each TaskGroup. Each Task specified by the user will be placed to the respective

TaskGroup with their Worker-level parallelism hint. Spout, as an exception, will only be located in a LocalTask because we want to make the partial process only happened in the LocalTasks. Last variable, *boltDependencies*, is used for GlobalTask to find the cloud location of LocalTasks and find the most suitable location based on its dependence.

In addition, TaskGroup also have the variable *clouds* to save all the cloud names where it will be located. This consideration is different from the view of a cloud that actually hosts a TaskGroup, not the opposite. The reason is because TaskGroup is a fixed component while the cloud can be added or removed at any time in the run-time. It is also because we are doing the deployment loop based on TaskGroup not based on the list of clouds; LocalTask first and then GlobalTask.

```
class TaskGroup {
  public String name;
  public List<String> clouds;
  public Map<String,Integer> boltsWithParInfo;
  public Set<String> boltDependencies;
}

class LocalTask extends TaskGroup {
    public Map<String,Integer> spoutsWithParInfo;
}

class GlobalTask extends TaskGroup {

}
```

Listing 5.3: TaskGroup Class

There are two information that need to be added to make the scheduler work efficiently; specifying the cloud name on each Supervisor and location of source clouds. First information is needed to specify the cloud location of each machines and separate machines from each other. A step-by-step process to implement this is shown on one of the Storm contributor, xumingming[9], shown on Listing 5.4. Supervisors in one cloud must have identical name and unique for each clouds. Preventing the scheduler to mixed up the Supervisors in the process. The second information is to implement the pairing between source clouds and the Spout we explained in the previous chapter. Currently the pairings are saved on a text file that is located in the same place with Nimbus instance. This is shown on Listing 5.5. One of the future works is to add the Supervisor modification to make this information available on the Zookeeper rather than text file. With this change, the scheduler will be able to receive the most up-to-date information when there is a change on the clouds.

```
>Storm.yaml
. . .
supervisor.scheduler.meta:
  name: "SUPERVISOR_NAME"
  cloud-name: "CLOUD_NAME"
```

```
. . .
```

Listing 5.4: Cloud name in each Supervisor

Other than these two informations, in the future the scheduler is planned to be able to retrieve information on a cloud's performance. For example the bandwidth limitation between clouds, computation quota (memory / processor limit) on each cloud, or any other information to create smarter scheduler based on available information.

```
>Scheduler-SpoutCloudsPair.txt
messageSpoutLocal1;CloudEdgeA,CloudEdgeB
messageSpoutLocal2;CloudEdgeB,CloudEdgeC
```

Listing 5.5: Spout and source cloud pairings

Geo-scheduler will start the process by categorizing each Task to their TaskGroup based on the value of "group-name" on each Task. The scheduler also retrieves the Spout and source clouds information. After task assignations are complete, the next step is the TaskGroup deployment to the clouds. LocalTask and GlobalTask have different ways to select the clouds where it will be deployed. As we discussed before, each LocalTask will be deployed on their source clouds. This is done by using a key-value list; the key is the spout name and value is the list of source clouds. When a Spout is deployed in a LocalTask, the list of source clouds with the key similar with the Spout are also registered in the same LocalTask. On the other side, GlobalTask has a harder problem to do the deployment based on the Algorithm 2 in Chapter 4.2.2 for choosing the most suitable cloud. First, every Bolt in a GlobalTask need to check their stream dependencies to the previous LocalTask. A stream or *GlobalStreamId* is the identity of a relationship between two Tasks that is needed when the sender need to send a Tuple to the receiver. This is done by saving sender TaskID of each stream to the list (Listing 5.6).

```
for(GlobalStreamId streamId : inputStreams)
{
    schedulergroup.boltDependencies.add(streamId.get_componentId());
}
```

Listing 5.6: looking for stream dependencies on each Bolt Class

With the Bolt dependencies information, GlobalTask are able to find the location of the clouds that hosts the sender TaskID and create a list of cloud dependency (Listing 5.7).

```
for(String dependentExecutors : globalTask.boltDependencies)
{
    if(executorCloudMap.getValues(dependentExecutors) == null)
      continue;
  else
```

```
      cloudDependencies.addAll((List<String>)

    executorCloudMap.getValues(dependentExecutors));
}
```

Listing 5.7: Creating a list of cloud dependency to this GlobalTask

Finally GlobalTask will be able to call a function inside the *CloudLocator* class (Listing 5.8) while specifying all of the available clouds and the cloud dependency list. The function will return the name of a single cloud where the GlobalTask will be deployed into. We made two sample implementation to locate the best cloud based on network latency between clouds: Average and MinMax[5].

```java
public String getCloudBasedOnLatency(CloudLocator.Type type,
Set<String> cloudNameList, Set<String> cloudDependencies)
{
    String choosenCloud = null;
    . . .
    //do computation here
    . . .
  return choosenCloud;
}
```

Listing 5.8: Sample of CloudLocator class function to choose the best cloud

The final part of the scheduler is to assign the Tasks inside the TaskGroup to the Worker instances inside the clouds. The idea for this part is actually similar with the basic concept of Task deployment in Storm to multiple Workers. The process begins by taking a TaskGroup and all clouds assigned to it, one by one. The scheduler is then able to collect the Tasks from the TaskGroup and the Workers from the selected cloud. From this view, we are actually looking at a single data-center deployment: Multiple Tasks into a cluster of Workers. The Tasks are deployed into the Workers with round-robin method, starting from the Spouts followed by the Bolts. This process is then carried out for each TaskGroup and their assigned clouds.

5.2 ZoneGrouping

Our new ZoneGrouping is created by implementing *CustomStreamGrouping* abstract class. *CustomStreamGrouping* has two main method: *prepare* and *chooseTasks*. *prepare* method will be called once when the scheduler is finished deploying the Task to the Worker and *chooseTasks* method is called each time a task wants to send a Tuple.

Implementing the correct stream Grouping to do local cloud distribution is a bit tricky. Each *StreamGrouping* object are bound to the relation between two Tasks in a Storm Topology, but there are no information about location of the real objects in the system. Listing 5.9 shows the *prepare* method example of *CustomStreamGrouping*. *source* in line 4 gives all IDs on the sender task and *targetTasks* in line 2 gives the IDs of the receiver task. To find the cloud location of the tasks, we are relying on our custom scheduler to provide this information. The *prepare* method starts by reading the result of the scheduler and create a list of every tasks location on the cloud, including all of the parallelization. For example from Listing 5.10, ID 28-31 is the parallelization of "SpoutA" Task that is distributed into CloudEdgeA and CloudEdgeB.

```
1 public void prepare(WorkerTopologyContext context,
2     GlobalStreamId stream, List<Integer> targetTasks) {
3   . . .
4   for(Integer source :
5       context.getComponentTasks(stream.get_componentId())) {...}
6   . . .
7 }
```

Listing 5.9: "prepare" method in ZoneGrouping

```
CloudEdgeA;29,28,20,19,17
CloudMidA;14,15,10,11,8,9,12,13,5,6,7,4
CloudEdgeC;32,35,26,23,22
CloudEdgeB;31,30,21,16,18,34,33,27,24,25
```

Listing 5.10: Result from custom scheduler

With all of these available information, we are able to make a sender-receiver pairings. But the second problem is, there should be no complex, time wasting computation in *chooseTasks* method, because this method will be called for each processed Tuple. Slow process will make a lot of queue and reduce system performance. That is why we create a matching table called *taskResultList* in the prepare method. When the *chooseTasks* method is called (listing 5.11), we can just do a simple lookup based on ID of the sender (*senderTaskId*) as the key and get the value of the entire receiver IDs.

```
public List<Integer> chooseTasks(int senderTaskId, List<Object> values
    )
{
    //Matching the sender ID with all target ID
    //located in the same cloud
    List<Integer> result = taskResultList.get(senderTaskId);
    . . .
    return result;
}
```

Listing 5.11: chooseTasks method in ZoneShuffleGrouping

5.2.1 ZoneGrouping in Storm Topology

ZoneGrouping must be used for every Task pair inside the same LocalGroup, and provided manually by the user. This disadvantages occurred because the Scheduling system is a different component with stream Grouping. Scheduler is called once when the topology is deployed or if there are any re-balancing event, but Grouping is used in the runtime of a Topology. Our future plan is to integrate both custom Scheduler and ZoneGrouping to the main Storm deployment branch. In this case, the system can be triggered automatically to use the ZoneShuffleGrouping rather than default shuffleGrouping when we want to use the local-global TaskGroup in multi-cloud environment.

In the current version of the ZoneGrouping, the users can put the Grouping in the similar way when pairing the Tasks in the Storm Topology. The code example is shown in Listing 5.12. Top part gives the example of calling default Storm ShuffleGroouping and the bottom part shows how to set the ZoneShuffleGrouping.

```
//Default shuffleGrouping
builder.setBolt("messageBoltGlobal1_1A", new SOLBolt(), 4)
.shuffleGrouping("messageBoltLocal1_1")
.addConfiguration("group-name", "Global1");

//Custom ZoneShuffleGrouping
builder.setBolt("messageBoltLocal1_1", new SOLBolt(), 4)
.customGrouping("messageSpoutLocal1", new ZoneShuffleGrouping())
.addConfiguration("group-name", "Local1");
```

Listing 5.12: Example of adding Task with ZoneGrouping

5.3 Guidelines

In the current implementation, user must specify the TaskGroup for every Tasks manually. To assist in understanding the concepts and providing ideas on which TaskGroup each Task should be deployed into, We created some information and consideration guidelines for creating a good deployment. There is no exact rule on the deployment because it depends on how many process that need to be processed in the same location where the data source is located to receive partial results as fast as possible.

As described previously, We are focused on the Topology that has multiple result stages; Partial and global results (Figure 4.1). The Partial computation are expected to process the Tuple separately based on where the data sources are emitted, even when the same type of data sources are located on many locations. At very least, the LocalTask will consists of a single or multiple Spouts. Every Spouts have a direct association with the input stream, so the location of the Spouts must be in the same data-center / cloud with the related data source.

Choosing the TaskGroup for Bolt Tasks are based on the closest result (Partial or Global) needed on the Topology graph. Bolts that used for processing Partial result should be located in the the same LocalTask with the predecessor Spout or Bolt that has been assigned to a TaskGroup. On the other hand, if a Bolt is expected to do an aggregation process from multiple Bolts doing Partial results then this Bolt are considered to be put into GlobalTask.

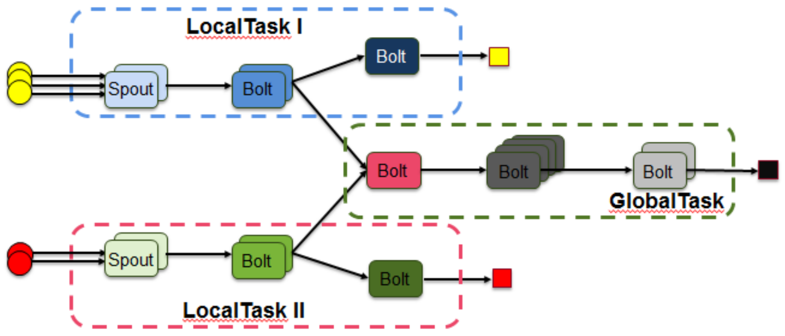


Figure 5.1: Topology example for TaskGroup deployment. LocalTask are deployed on the Tasks between the input until Bolt that emitting Partial result.

The example of the deployment for this type of Topology can be seen in Figure 5.1. A good way to start choosing the location of each Task is to follow the directed graph, work on Spouts first until the furthest Bolt. In the figure, there are two pairs of sources and its Partial result. We divide both spouts into different LocalTasks because each have different type of data source. In another case there is also a possibility of two Spouts will be located in the same LocalTask to be processed together. Bolts working on Partial computation will follow the same LocalTask as the previous Tasks. Every Bolts with blue-based color will be processing yellow type input in the same cloud location and parallelized depending on the distributed sources location. Bolts with green-based color for processing the red type input will be deployed similarly. In the end, the pink Bolt that is in charge on receiving stream multiple LocalTask will need to be assigned to a GlobalTask along with all of the successor Tasks.

5.4 Considerations

In this section, we are going to discuss some part that is important to take a look at when deploying Storm with our new scheduler and Grouping but have not thoroughly tested. Some are tested on the go during the process of checking the correctness of the scheduler and Grouping.

5.4.1 Scalability

The scalability goal of our Geo-scheduler is to focus on geo-distribution of the LocalTask group. Each LocalTask is parallelized to any number of clouds provided. If on the runtime there are new clouds with the related source registered in Storm system, the scheduler will trigger the system to run the scheduler again. While the other TaskGroup is already deployed, the scheduler will put new instance of LocalTask inside this new cloud. There should be no effect of the current system.

For the GlobalTask, in the current development, there is no need to deploy GlobalTask in more than one cloud. This comes with consideration that GlobalTask is located in a cloud with high performance and good intra-cloud scalability so there are no need to deploy GlobalTask in multiple cloud.

5.4.2 Fault tolerance

We are considering two case of fault tolerance in our system. First is for when a cloud loses some part of the computation power. There is a possibility that some physical machines are down or there are failures inside a cloud. The Storm system will see this as reduced number of Supervisor in the runtime. In this case, Storm has their fault tolerance as written in their Apache Storm manual. When a Supervisor is down for too long, all of the assigned Tasks in that Supervisor will be moved to another Supervisor. With our Geo-scheduler, the expected action is to handle this problem by moving the Tasks from the dead Supervisor to another Supervisor that is located in the same cloud.

The second case is when we are looking at a bigger size of faults; when the Storm loses a connection to a whole cloud. This can happen when the network is down or something happened in the whole data-center. This situation is seen in the Geo-scheduler as losing a single object of TaskGroup. When this happened on a cloud hosting LocalTask, the cloud is removed from the list of source cloud, which means there will be no effect on the scheduler or Storm itself. The only visible effect is there is no data or Tuple emitted from that cloud location. If the dead cloud is hosting the GlobalTask group, then the scheduler will need to do a re-scheduling process to choose another cloud to host the GlobalTask.

6 Chapter 6

Evaluation

This chapter focused on evaluation of the correctness of our Geo-scheduler and ZoneGrouping as a proof-of-concept to increase the performance of real-time stream processing with Geo-distributed sources. First, we explained the network topology used to connecting multiple clouds hosting Storm computation nodes (Workers). We are also explaining the Storm topology with TaskGroup used as the test-case. The results are divided into two main parts: Validate our implementation of the scheduler and measure the performances for a specific test-case with hierarchical computation-based Storm Topology.

6.1 Network Topology

In this experiment, we created an emulation of multi-cloud network model, shown in Figure 6.1. Following the concept of Geo-distributed sources, there are three clouds where the sources are located. Information of the data source types and their location are shown in Table 6.1. EdgeCloud A is able to collect type "1" from their surroundings, EdgeCloud C is able to collect type "2", and EdgeCloud B is able to collect both. As we explained before, the information of the source locations are important to deploy the Tasks in the most efficient place. Two remaining clouds (Central cloud A & B) is the current cloud model deployed in a data-center. These clouds did not have any data sources attached to them.

The clouds are located on different areas and connected to each other with some latency in the network. We call the clouds with sources by EdgeCloud because of their distributed location and have smaller computation power than the Central Clouds. The computation power in our case is counted based on the number of physical machines that can host Storm components to become a 'compute node'. In Figure 6.1, each EdgeClouds have 2 physical machines and 4 for each Central Clouds. To get more isolated information of the traffic between the clouds, we reserve the Central Cloud B to only host management components. There will be no Task deployed and any computation in this cloud.

All of machine nodes, routers, and network links are emulated by using CORE network emulator running under inside a physical server located in KTH. The specification of the machine are similar with the experiment in chapter 3.3, shown in table 3.1.

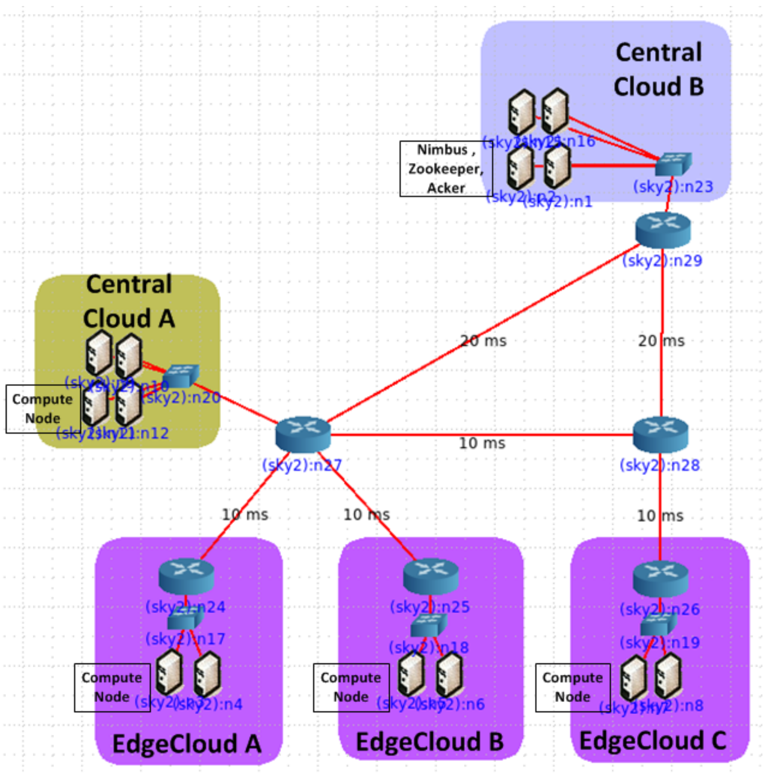


Figure 6.1: Multi-cloud Topology. The topology consist of three Edge Clouds and two centralized clouds

Table 6.1: Different data source and location

Source type	Cloud
Source 1	EdgeCloud A, EdgeCloud B
Source 2	EdgeCloud B, EdgeCloud C

6.2 Storm Topology

Based on the LocalTask and GlobalTask of the TaskGroup, we are focusing on Storm Topology that is embracing the computation with partial-global hierarchy where the sources are geographically distributed on different clouds. The topology are shown in Figure 6.2. The Topology is divided into three types of Task component: Spout, normal Bolt, and result Bolt. Each Spout is set to generate a 500 bytes Tuple with a constant rate of 450 Tuple per second. The normal Bolt is basically not doing any computation and directly pass each incoming Tuple to the next Bolt. One thing to note is that each normal Bolt will dump 40% of the Tuple from the input stream. This is following the concept of filtering or aggregating in Stream Processing where the output stream rates is usually have smaller rates than the input stream. The third Task type is The result Bolt that acts as the sink or destination where the processing time for each Tuple is collected. Based on hierarchical computation, there are two result Bolts: "*LocalBoltResult*" is the sink that is expected to generate partial result from the stream process to the outside system and "*GlobalBoltResult*" that generate the global result, the aggregate result from all partial results on all clouds.

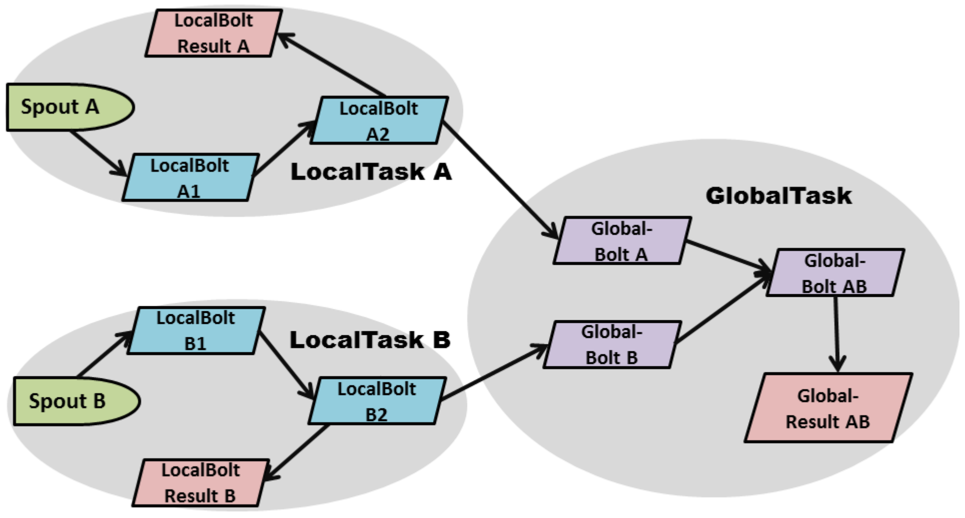


Figure 6.2: Storm Topology used for the validation evaluation. Input: Two sources collected by each respective Spouts. Output: partial and global results.

Each LocalTask is assigned to one of the data source types. Spout in LocalTask A will collect the data from source "1" and Spout in LocalTask B is assigned to source "2". Every LocalTask will do some computation and produces the partial result. The partial results are expected to report the Tuples only from the same source location, so it is important to make sure there are no Tuples from other same LocalTask being counted. The computed Tuple from all LocalTasks are then combined into GlobalTask A. The aggregated results will be emitted from the GlobalBolt Result.

One of the sample use cases that have similar idea with this Topology can be seen on [16]. This paper discussed the idea of a distributed clustering algorithm with both Partial and Global result. Unfortunately, we do not have the time to apply an implementation of this algorithm in this thesis work. But it will be a very interesting future work to see the result precision and effect of the performances.

With our storm modification design, users could create a normal Storm Topology while adding some GroupTask information needed for the scheduler. On the Topology shown on Figure 6.2, the Tasks are divided into 3 TaskGroup. Two LocalTasks are divided based on different Spout and all of the following bolts. On each LocalTask there is also a Bolt that produce results on each cloud. GlobalTask is assigned to combine the results from all LocalTasks, regardless of location and number.

6.3 Implementation validation

6.3.1 Geo-Scheduler

The Storm scheduler works in the startup phase after the Topology is deployed by the user. All of the scheduling process is done inside the Nimbus component in Central Cloud B. The result is then deployed to all other clouds hosting the Worker / compute node. The process starts by creating Task objects from the given Topology as much as its total amount of parallelization (Multi-level parallelization are described in section 4.2.2). As a result, the Tasks are created with unique TaskID number. Based on our previous Topology, the results of each Task and their TaskIDs are shown in Table 6.2. These Tasks will be distributed on the clouds by using different type of scheduler.

We compared both default EvenScheduler and our custom Geo-scheduler. Based on the information of the sources, we want the scheduler to deploy the LocalTask A on the location of Source "1" and LocalTask B on Source "2". The scheduling results are shown by the Table 6.3. Our Geo-scheduler deploy each Bolt and Spout in the expected clouds. Both EdgeCloud A & B receives the same amount of LocalTask A and EdgeCloud B & C receives the same amount of LocalTask B.

The EvenScheduler also deploys the Task as expected. There are no consideration on the location of the Tasks, focusing only on the balanced number of Task between each Worker. The distribution is done in a round-robin way as we can see it from the pattern of the TaskID. Central Cloud A get more Tasks because of their higher number of Workers than the EdgeClouds. This deployment will be inefficient for our

Table 6.2: TaskID information

Task name	Total parallelization	Task IDs
Spout A	4	25-28
Spout B	4	29-32
LocalBolt A1	4	13-16
LocalBolt A2	4	33-36
LocalBolt B1	4	19-22
LocalBolt B2	4	37-40
LocalBolt Result A	2	17-18
LocalBolt Result B	2	23-24
GlobalBolt A	4	1-4
GlobalBolt B	4	5-8
GlobalBolt AB	2	9-10
GlobalBolt Result AB	2	11-12

cases because the Tuple will do a lot of inter-data center movements when streamed from one Bolt to the next Bolt. More detailed evaluation on the inter-data center traffic is explained in next section.

6.3.2 ZoneGrouping

After the task is deployed to the respective cloud, now we are evaluating the correctness of the stream Grouping to be able to process Tuples pooled from the same cloud location (Partial processing). We compared the default Storm shuffleGrouping with our ZoneGrouping by looking at the inter-data center network traffic for about 100 seconds of running Topology. The traffic are collected by using Linux *ifstat* tool. The information collected are the size of the traffic coming inside to the cloud (inbound) and outside from the cloud (outbound). While the network is an emulated one and there are no traffic usage other than Storm, we consider the inbound and outbound traffic as the inter-data center communication where the Tuples are moving between the Bolts that located the other cloud. Communication between different Workers that is located in the same cloud (intra-cloud) are considered stable with high-speed latency-less connection [23] and is not measured in the experiment result.

The outbound traffic are shown in Figure 6.3. From this graph we can see that ZoneGrouping deployments are sending less inter-data center Tuple stream than

Table 6.3: Scheduler result - Location of the assigned TaskIDs

Cloud name	EvenScheduler	Geo-scheduler
EdgeCloud A	4,9,14,19,24,29,34,39,	14,15,17,25,28,33,34
EdgeCloud B	1,8,11,18,21,28,31,38,	13,16,18,19,20,24,26,27,29,32,35,36,38,40
EdgeCloud C	2,5,12,15,22,25,32,35	21,22,23,30,31,37,39
Central Cloud A	3,6,7,10,13,16,17,20,23,26,27,30,33,36,37,40	1,2,3,4,5,6,7,8,9,10,11,12

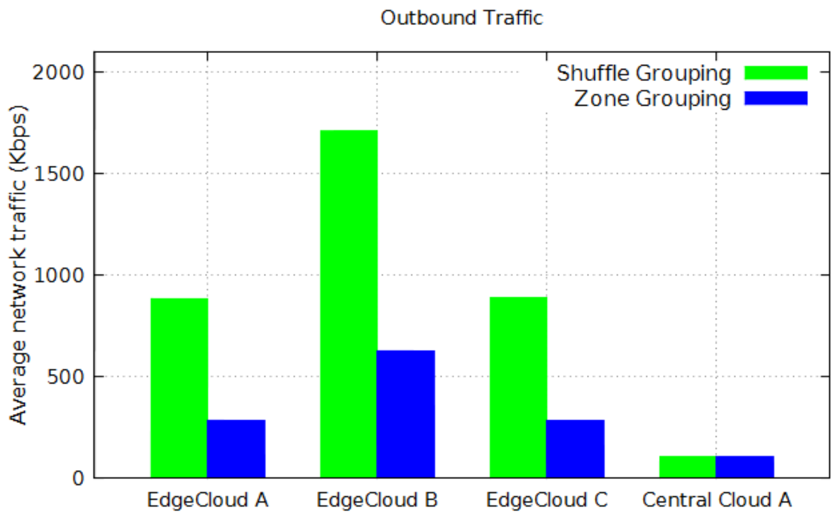


Figure 6.3: Outbound traffic rates from each clouds

EvenGrouping. Another important achievement is the inbound traffic shown in Figure 6.4. On default shuffleGrouping, EdgeClouds are receiving traffic (Tuples) from another cloud where it should not happen. We want each LocalTask to only process data from their own location. In this case, our implementation of ZoneGrouping works as intended. There are no sighted communications between EdgeClouds. The network traffics seen are from only Central Cloud A task that hosts the GlobalTask and receives Tuples from all EdgeClouds.

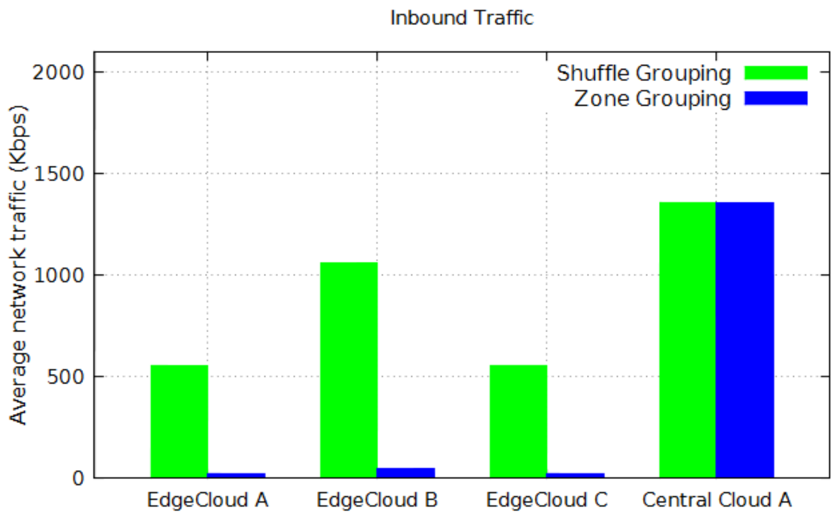


Figure 6.4: Inbound traffic rates from each clouds

We have shown the correctness on both of our implementation on Geo-scheduler and ZoneGrouping. Next we describe the performance evaluation compared to default Storm configuration.

6.4 Performance evaluation

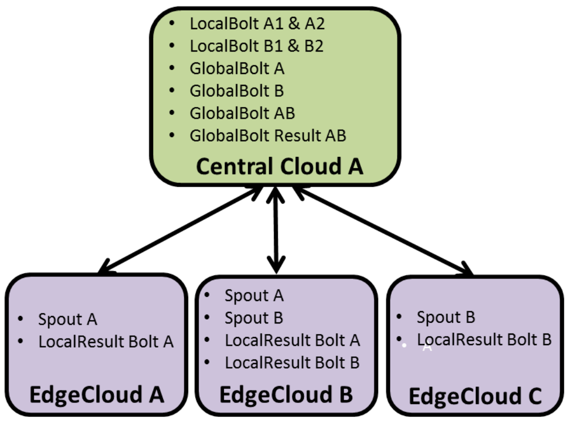


Figure 6.5: Task deployment for Centralized Scheduler

The main purpose of the performance evaluation is to analyse how the different Task deployment on multi-cloud environment could affects the whole network traffic, bandwidth usage, and system response time for real-time or latency-sensitive stream processing application.

We made a comparison between three types of Apache Storm scheduler. First we are using the default Storm scheduler (EvenScheduler). This scheduler do a round-robin deployment for all Tasks and try to balance amount of Tasks between all Workers regardless of the location. For the second type of scheduler, we look back at the concept where all of the computation are done in a single cloud / data-center. This is how the current Stream processing works where the data itself is still geographically distributed. We call it the Centralized deployment. The sample of how the Tasks are deployed in this scheduler is shown in the Figure 6.5. While the data sources are geographically distributed, the Spouts and result Bolts are deployed on the EdgeClouds while other processing Bolts are located in the Central Cloud. Finally, the third scheduler to compare is our implementation of Geo-Scheduler and ZoneGrouping combination.

To be able to see the effects of the scalability when doing stream processing with Geo-distributed data sources, we expand the multi-cloud deployment shown previously in Figure 6.1 to bigger network size with more EdgeClouds. Rather than only using

3 EdgeClouds, in total we are using 9 EdgeClouds and 2 Central clouds. This new cloud deployment is shown in Figure 6.6. In these EdgeClouds, we deployed five different quantities of data sources; 4, 7, 10, 13, and 16 data sources. On every data source sizes, we set the specific amount on each Task parallelization to make sure the system run with the same computation load and same amount of Tuples.

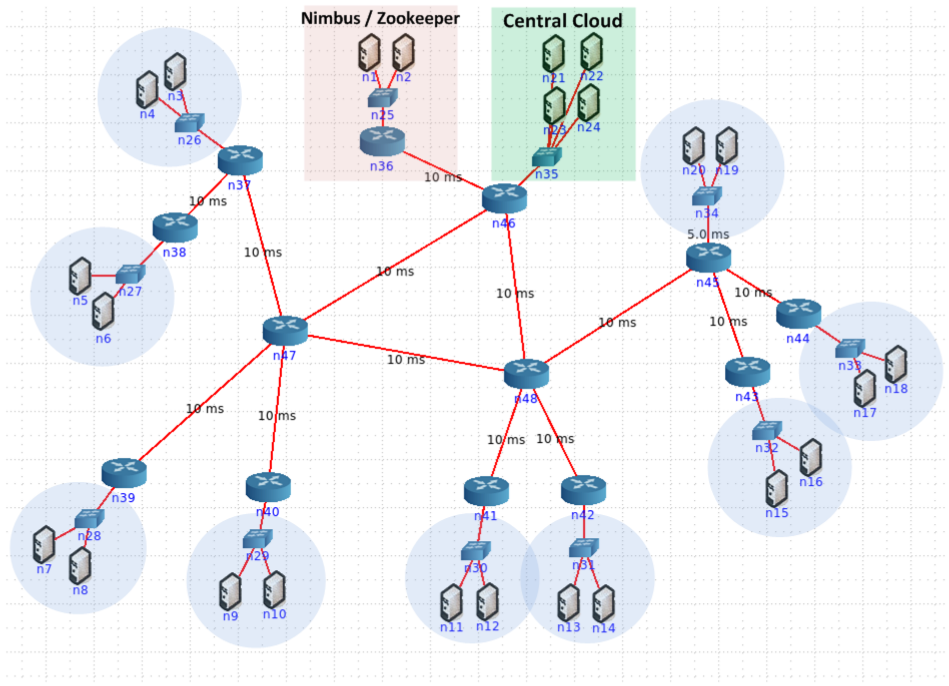


Figure 6.6: Network topology with 9 EdgeClouds

6.4.1 Network traffic

The first metric to measure is the amount of network traffic in the whole system when running the Storm Topology. A high amount of traffic means more bandwidth used by Storm and makes the system more vulnerable to network bottleneck. High traffic will also means the Topology are taking a lot of bandwidth that could interfere with other application that relies on the network connection. The result are shown in Figure 6.7. With higher number of data sources deployed, we were expecting that more Tuples to be processed in the system. In all cases, the EvenScheduler deployment always use the most bandwidth. This is because the location of the Task is not controlled which makes too many inter-data center communication. Centralized scheduler has less traffic than EvenScheduler because the whole computation is located in one place. The only traffic used is when sending the stream of data sources from all EdgeClouds to the central cloud and giving back the partial results that need to be received by LocalResult Bolts on each EdgeCloud. By using ZoneGrouping deployment, the partial result computation is done on every EdgeClouds, so the only traffic needed is to send the stream from localTasks to the GlobalTask. With this scheduler, we reduced the network traffic by around 3.5 times when compared with EvenScheduler for 16 data sources.

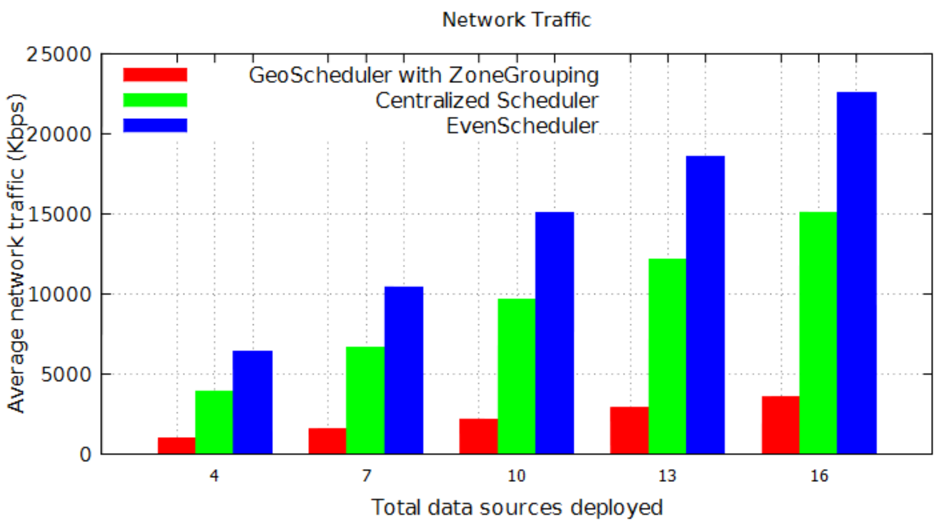


Figure 6.7: Average network traffic in the system with different scheduler

Another insight that we can deduce from the graph is the increment of the traffic when we add more data sources. By adding three data sources each (six for both types), the ZoneGrouping only adds to the network traffic by around 1500 Kbps compared with EvenScheduler that adds around 8500 Kbps. The importance of these results are the scalability of the traffic generated by stream processing can be escalated as the real scenario of a geo-distributed deployment is expected to have have tens or maybe hundreds of EdgeClouds with different type of data sources and more heterogeneous network connection. Running stream processing in these deployments will be an interesting subject to explore further.

6.4.2 Latency-sensitive application

More performance metrics that are important to evaluate is the advantages of having hierarchical Storm Topology with the real-time application. Real-time application requires a very low latency, which is one of the focuses on using LocalTasks. By processing data in the same place as much as possible, Figure 6.8 shows the average Tuple processing time when emitted from the Spout until it reaches the result Bolt (LocalBolt-Result) on the same location to emit the partial results.

With Geo-Scheduler the LocalTasks are located directly with the data sources so there are no network latency affecting the Tuple process time to produce partial result. The Centralized deployment has an extra latency because they are doing the partial process in the Central Cloud and then need to send back the partial result to each EdgeClouds. In return, centralized deployment gives a very good processing time for global result as everything are processed in single location. The small reduction on the partial result that happened when adding the number of data sources from 4 to 16 are caused by taking the average latency between EdgeClouds to the Central Cloud.

The EvenScheduler surprisingly have almost similar Tuple processing time with Centralized scheduler for partial result. However, it has a very high processing time for getting the Global result compared to both Geo-Scheduler and Centralized scheduler. In overall, Geo-Scheduler achieve the fastest processing time for Storm Topology with hierarchical process that has both partial and global result. This is because the Geo-Scheduler tries to minimize the inter-data center network communication as much as possible.

From all of the results, we found out that modification on Storm scheduler and stream Grouping method are able to improve the stream processing performance for a Topology with hierarchical process where each clouds need a fast result from their own local information and an aggregated global result from all of the participating clouds.

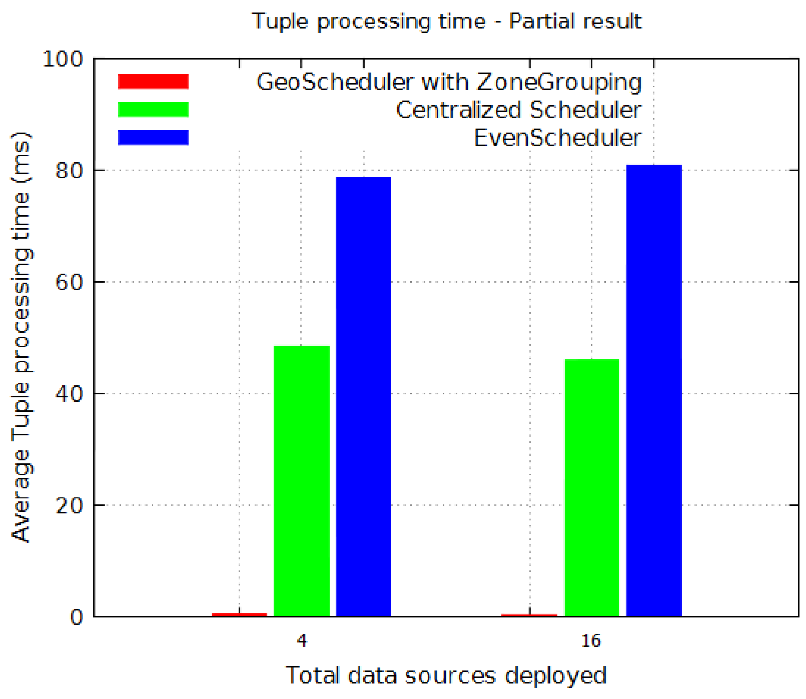


Figure 6.8: Average Tuple processing time to receive partial result

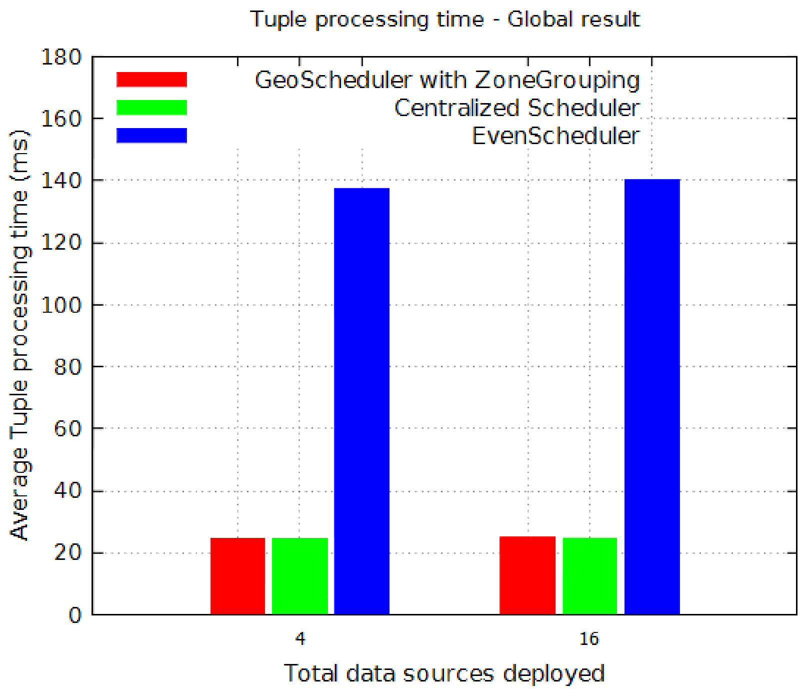


Figure 6.9: Average Tuple processing time to receive global result

7 Chapter 7 Conclusion

7.1 Discussion

To conclude this thesis report, we sum up our work and contributions in this section. First, we are looking at the current trends of stream processing deployment. While stream processing is able to do high speed low-latency computation, the system is usually deployed in a single cloud or data center. However, if the data sources are distributed in different location, it is hard to find a single location that able to achieve the satisfaction level for all users. For example, when processing latency-sensitive application from numerous amount of smart phones or Internet-of-Things devices, actors that are located far from the computation server or have slow connection will experience higher response time compared with others that have good connection.

This thesis focused on the deployment of stream processing that is able to maintain the response time regardless the location of the sources. To be able to do this, we observed the ongoing research of Edge Clouds: micro clouds or cloudlets that is located in the edge of the network, close to the end-user location. Edge Cloud emerges from the virtualization of telecoms network components or Software Defined Networking (SDN) which enables any application to be deployed on the top of their main functionality. By utilizing its computation power, we came up with the idea to deploy stream processing in multiple clouds that are located closer to the sources.

The first part of the thesis work is to investigate the possibility to deploy stream processing in multiple clouds on different location. We choose Apache Storm as it is a very suitable framework for the experiments and looked on different ways to deploy the system: Multiple Storm on each sites or single Storm with centralized management. Then, we did an analysis to measure the performance against heterogeneity of network latency that could happen on a connection between clouds. The methodology was based on emulation of multiple clusters of servers / nodes on different location with variety of network conditions. From the investigation, we clearly saw some problems if we just use a standard single data center deployments on multiple clouds. The analysis of these experiments yielded the following conclusions:

- Controlled placement of Storm components on multi-cloud environments is needed as it affectsthe processing time and the amount of network traffic produced, lowering the stream processing performance as a consequence.
- In stream processing, a node with good computation power could perform very poorly if it has poor/imperfect connectivity with other nodes.
- The same straggler node from the previous point could affect the whole system performance because the default scheduler is only focused on the traffic load balancing when deploying the components.
- In a poor or highly unstable network, there is a need to carefully position the Storm management components, Nimbus and Zookeeper, to make sure there are no false-positive nodes in the runtime. Usually the chosen nodes are the ones that have good connectivity with all of the Worker nodes.

From these insights we create the idea to group the computation Tasks inside Storm Topology that has hierarchical results or output streams; partial and global computation. The part that receives the input stream and produces partial or local location-based result is expected to run in real-time or have very low latency responses. Partial computation holds the locality of their information based on the location of where the data is collected, which is why all of the tasks in this part will be deployed based on the distributed Edge Clouds concept. The result streams from partial or location-based computation are sent to the global computation part. This part is focusing on the combination or aggregation for many number partial computation. The scheduler will decide the cloud / data-center location to put the global computation based on the user specified criteria. This criteria is important to make sure the global computation is able to achieve the best performance.

We implemented these concepts by creating a plug-in with available Storm API in Java programming language. The latest Geo-scheduler implementation can be seen in https://github.com/Telolets/StormOnEdge. By implementing the modification using Apache Storm plug-in, users that want to utilize hierarchical process do not need to recompile or modify their default Storm deployment. To use the plug-in, there are two components that need to be deployed: Geo-Scheduler which is a Storm scheduler to do the specific partial-global Task deployment between the nodes and ZoneGrouping that needs to be applied to control the locality of data stream between partial computation. To validate the correctness between the concept and the implementation, we did an evaluation on both components. We also test its performance test to compar it with the default Storm Task deployment.

From both results, we found that modification on Storm scheduler and stream Grouping method are able to improve the stream processing performance for both parts of partial and global computation. Compared with the default Storm scheduler, in our experiment the network traffic are reduced by around 85% for all data sources. Another improvement made is the time needed to do real-time processing for partial computation. In the case of no computation inside the Bolts, we are able to reduce the latency by 80 milliseconds. We expect to have higher difference when we run a more realistic use case with computation on each Bolts, dynamic data rates, and more unpredictable network between the data-centers.

7.2 Future Work

There are many improvements and extensions that can be made from this thesis work. As the concept of the Edge Cloud itself is still in the research phase, we cannot do a real benchmark on these deployments. Rather, we focused on network latency between clouds for real-time use cases. This thesis is done as a proof-of-concept of stream processing deployment on a multiple clouds, rather than doing all of the computation in a single place.

One of the problems that will occur in the real implementation is the dynamic workload on each cloud. When the system is deployed using heterogeneous clouds from different provider, the computation power can be different on each cloud. The data rates received by each Local process can also be different on each area. This creates inefficiencies because of unbalanced workload. One of the ideas is to create a dynamic scheduler that able to monitor the load on each cloud on run-time. We think this will be a significant addition and will be very useful on real deployments.

The current implementation is also subject for the enhancements. Both Geo-scheduler and ZoneGrouping are basically one object for the hierarchical process, but separate in implementation because of the limitation of the Storm plug-in API. It will be for the best if both are integrated into a single system where the LocalTasks by default will be using the ZoneGrouping rather that specifying it in the Topology. More sophisticated GlobalTask location placement can also be made by using the state-of-the-art graph algorithms to find a best node from multiple nodes. Another idea is to create more than two-level hierarchical process for more complex stream processing Topology.

In the last performance evaluation on Chapter 6, each of the three types of deployment are able to process all Tuples at the same time. This is because there are no other factors that is able to affect the processing rates other than the network latency. In a real Storm deployment, we expect a more complicated topology with processing time on each bolt that creates a latency and possibility of Tuple queues. The performance heterogeneity of each cloud can also affect the overall system performance. These factors are important to be included in the system performance consideration.

Bibliography

[1] Common Open Research Emulator (CORE). http://www.nrl.navy.mil/itd/ncs/products/core. Accessed: 2015-04-01.

[2] Deutsche telekom becoming a software defined operator. http://telecoms.com/191662/deutsche-telekom-becoming-a-software-defined-operator/. Accessed: 2015-03-15.

[3] Guifi.net - Telecommunication network open, Free and Neutral. https://guifi.net/. Accessed: 2015-06-01.

[4] Making storm fly with netty. http://yahooeng.tumblr.com/post/64758709722/making-storm-fly-with-netty. Accessed: 2015-05-11.

[5] Minimax principle. Encyclopedia of Mathematics. http://www.encyclopediaofmath.org/index.phptitle=Minimax_principle&oldid=34361. Accessed: 2015-06-01.

[6] Netty.io. http://netty.io/. Accessed: 2015-05-11.

[7] Nokia and Intel innovation centre. http://www.theinquirer.net/inquirer/news/2363266/nokia-and-intel-launch-innovation-centre-to-spur-mobile-broadband-app-development. Accessed: 2014-02-12.

[8] Supervisor: A process control system. http://supervisord.org/. Accessed: 2015-05-11.

[9] Twitter storm: How to develop a pluggable scheduler. https://xumingming.sinaapp.com/885/twitter-storm-how-to-develop-a-pluggable-scheduler/. Accessed: 2015-03-15.

[10] Jeff Ahrenholz, Claudiu Danilov, Thomas R Henderson, and Jae H Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7. IEEE, 2008.

[11] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[12] Roger Baig, Jim Dowling, Pau Escrich, Felix Freitag, Roc Meseguer, Agusti Moll, Leandro Navarro, Ermanno Pietrosemoli, Roger Pueyo, Vladimir Vlassov, et al. Deploying clouds in the guifi community network. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 1020–1025. IEEE, 2015.

[13] Hyunseok Chang, Adiseshu Hari, Sarit Mukherjee, and TV Lakshman. Bringing the cloud to the edge. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 346–351. IEEE, 2014.

[14] Aleksandra Checko, Henrik L Christiansen, Ying Yan, Lara Scolari, Georgios Kardaras, Michael S Berger, and Lars Dittmann. Cloud ran for mobile networksa technology overview. *Communications Surveys & Tutorials, IEEE*, 17(1):405–426, 2014.

[15] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[16] Graham Cormode, S Muthukrishnan, and Wei Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1036–1045. IEEE, 2007.

[17] Ken Danniswara, Hooman Peiro Sajjad, Ahmad Al-Shishtawy, and Vladimir Vlassov. Stream processing in community network clouds (*To be published*). In *Community Networks and Bottom-up-Broadband, 2015. CnBuB 2015. The 4th International Workshop on Community Networks and Bottom-up-Broadband.* IEEE, 2015.

[18] Intel. Carrier cloud telecoms – exploring the challenges of deploying virtualisation and sdn in telecoms networks. August 2015.

[19] Yonghua Lin, Ling Shao, Zhenbo Zhu, Qing Wang, and Ravie K Sabhikhi. Wireless network cloud: Architecture and system requirements. *IBM Journal of Research and Development*, 54(1):4–1, 2010.

[20] Marek Rychly, Petr Koda, and P Smrz. Scheduling decisions in stream processing on heterogeneous clusters. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on*, pages 614–619. IEEE, 2014.

[21] Mennan Selimi, Felix Freitag, R Pueyo Centelles, and Agustí Moll. Distributed storage and service discovery for heterogeneous community network clouds. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 204–212. IEEE, 2014.

[22] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm @ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[23] Radu Tudoran, Olivier Nano, Ivo Santos, Alexandru Costan, Hakan Soncu, Luc Bougé, and Gabriel Antoniu. Jetstream: enabling high performance event streaming across cloud data-centers. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 23–34. ACM, 2014.

[24] Apache ZooKeeper. What is zookeeper. http://zookeeper.apache.org/. Accessed: 2015-05-11.

Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, 30 September 2015

. . . .
Ken Danniswara