KOSTALIA ELISAVET ELLI

STREAM DATA PROCESSING IN THE CLOUD FOR REAL-TIME ANOMALY DETECTION

Technical University of Crete
Department of Electrical and Computer Engineering

STREAM DATA PROCESSING IN THE CLOUD FOR REAL-TIME ANOMALY DETECTION

ELISAVET ELLI KOSTALIA
SUPERVISOR: EVRIPIDES G.M. PETRAKIS

A thesis submitted in partial fulfilment of the requirements
for the diploma of Electrical & Computer Engineer

Chania, June 2019

Abstract

In recent years, data stream processing is becoming extremely popular because of the Big Data era and the increasing number of microservices and IoT devices being used, therefore, the development, deployment and management of distributed services are more important than ever. Data gets stored and analyzed in order to provide predictive and actionable results via frameworks, such as Apache Storm that allows distributed real-time computation of tons of data coming in extremely fast, from various sources. Non-functional requirements often demand a highly-available, high-throughput, fault-tolerant and massively scalable solution. In this context, Apache Kafka is used as a publish-subscribe messaging system that will serve as a broker between various data sources. In our implementation, all services above can reach their highest utility when deployed in containers. This way, the implementation takes advantage of the virtualization features of cloud computing. We ran several experiments based on a simulated (but realistic) use case scenario for detecting unauthorized system accesses (anomalies) in real time. The project runs a distributed cluster of Apache Storm worker nodes (implemented as separate containers) that process incoming data, and uses decision tree classifiers to detect anomalies based on a given dataset. The experimental results demonstrates that Amanda application responds to the increasing resource demands of the application leading to significantly faster response times while more workers are deployed distributed compared to a non-distributed implementation where all service requests are handled by the maximum statically pre-allocated resources.

Acknowledgements

I wish to express my gratitude to my supervisor, Prof. Euripides Petrakis, for his continuous guidance, suggestions and encouragement along the way. Special thanks to Spyros Argyropoulos for his support and his valuable advices. I would also like to thank Prof. Vassilis Samoladas and Prof. Georgios Chalkiadakis for serving on my thesis committee. I would not have been able to complete this without some amazing people in my life: my family, especially my parents, for the support and the motivation and, of course, all my friends, thank you for the memorable moments we lived together during our studies, the never-ending patience and the encouragement. All of you contributed to this, each one on his own way. I Thank you all.

Contents

1. Introduction

1.1 Overview

We present a framework for real time anomaly detection using docker containers to deploy the distinct processes. What we focused on, is creating an application that supports stream data processing, not in a single server environment, but in the Cloud. For the deployment of our application, a flexible implementation which is based on Docker containers. This means that our application is based on lightweight virtualization that can run anywhere distributively, and not necessarily on a single server.

It is a scalable, fault-tolerant application for handling data in real-time and at a massive scale. The proposed application uses existing big data processing frameworks, Apache Kafka, and Apache Storm in conjunction with machine learning techniques and tools. The approach we suggest consists of a system for real-time data processing and analysis of the network-flow data. For our application we used KDD Cup 1999 Data set[1]. This dataset was used in the Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Third International Conference on Knowledge Discovery and Data Mining. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between “bad” connections, called intrusions or attacks, and “good” normal connections. This database contains a standard set of data to be audited, which includes a wide variety of intrusions simulated in a military network environment. Furthermore, the network anomaly patterns were identified and evaluated using machine learning techniques. We present results on anomaly detection with a testing set of network traffic data evaluated with multiple workers of Storm.

Application containerization works with distributed applications, as each container operates independently of others and uses minimal resources from the host. Each microservice communicates with others through application programming interfaces with the container virtualization layer able to scale up microservices to meet rising demand for an application component and distribute the load. This setup also encourages flexibility.

1 KDD Cup 1999 Data http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

1.2 Motivation

The primary motivation towards advocating a distributed approach is the overall traffic of the Internet data that has noted a significant increase the past years. This has led to a direct increase in the amount of network traffic that the individual sub-networks of the Internet are expected to handle. The detection of unexpected behavior in data describing network traffic. A cluster based solution provides scalability and fault-tolerance in addition to providing scope for parallelization of computations that result in a direct increase in system throughput, responsiveness and performance. The security solutions that can be built for the Advanced Metering Infrastructure built usually fall into two categories:

1. Detection: This class of solutions focuses on the detection of potential threats to the system, merely reporting the current state of the system to the person in charge of the network, usually the System or Network Administrator.
2. Prevention: This class of solutions encompasses both the detection of potential threats and real-time preventive action taken in accordance to these threats.

Our application runs under the first category of solutions to detect anomalous activity in the network that is potentially malicious and it takes no preventive action. However, our system is designed in a modular fashion keeping in mind future requirements and can be easily extended to take preventive action, including things like making the application run online or changing firewall configurations on a real time basis to blacklist connections belonging to certain IP addresses that are thought to belong to the attacking entities. The containerized environment is expected to provide the independence of the distinct processes running and the portability of our application. Application scalability is also a significant factor that leads Docker containers to make most of the application.

1.3 Problem Definition

Our problem derives from a number of applications that run and take as an input big loads of data and in most cases, in real time. We were called to face a number of independent issues at various granularities:

- handle incoming real-time data
- detect deviations from “normal” data, intruders
- deploy a multi-node Storm cluster with working topology
- containerize each distinct process

The main task of Amanda application is to provide efficient handling of real-time data taking full advantage of containerization.

1.4 Solution

Our solution to the problem is dedicated to the orchestration of platforms such as Apache Storm and Apache Kafka deployed on Docker. We built a dockerized application we called Amanda that deploys Storm's orchestration of big data processing in real time. Containerization's main feature, scalability, is a factor we need to take full advantage of, in order to achieve a better performance of the application, in terms of the response time of the application. What is expected to prove is the ability to make the application scalable after using a containerized environment to deploy it on.

1.5 Contribution

The utilization of Docker, including the images and volumes, targeting to scalability, reallocation of computing resources of the system and monitoring the load of the real-time input in the system. The algorithm that runs inside the Storm workers is a trained model that came out of the machine learning training.

2. Background

2.1 Cloud

Cloud computing supports the on-demand availability of computer system resources, including data storage and computing power, without the active management of the user. It relies on sharing of resources to achieve coherence and economies of scale. Large clouds, predominant today, often have functions distributed over multiple locations from central servers. Clouds may also be limited to a single organization (enterprise clouds), be available to many organizations (public cloud), or a combination of both (hybrid cloud). The availability of high-capacity networks, low-cost computers and storage devices as well as the widespread adoption of hardware virtualization, service-oriented architecture, and autonomic and utility computing has led to growth in cloud computing[1]. Cloud computing exhibits the following key characteristics:

Device and location independence give users the ability to access systems using a web browser regardless of their location or what device they use. As infrastructure is typically provided by a third-party and accessed via the Internet, users can connect to it from anywhere.

Maintenance of cloud computing applications is easier, because they do not need to be installed on each user's computer and can be accessed from different places.

Agility for organizations may be improved, as cloud computing may increase users' flexibility with re-provisioning, adding, or expanding technological infrastructure resources.

Cost reductions are claimed by cloud providers. A public-cloud delivery model converts capital expenditures to operational expenditure. This purportedly lowers barriers to entry, as infrastructure is typically provided by a third party and need not be purchased for one-time or infrequent intensive computing tasks. Pricing on a utility computing basis is "fine-grained", with usage-based billing options.

Multitenancy enables sharing of resources and costs across a large pool of users thus allowing for: centralization of infrastructure in locations with lower costs, peak-load capacity increases (users need not engineer and pay for the resources and equipment to meet their highest possible load-levels) utilisation and efficiency improvements for systems that are often only 10–20% utilised.

Monitoring the system performance by IT experts from the service provider, and consistent and loosely coupled architectures are constructed using web services as the system interface.

Productivity may be increased when multiple users can work on the same data simultaneously, rather than waiting for it to be saved and emailed.

Reliability improves with the use of multiple redundant sites, which makes well-designed cloud computing suitable for business continuity and damage recovery.

Scalability and elasticity via on-demand provisioning of resources on a self-service basis in near real-time, without users having to engineer for peak loads. This gives the ability to scale up when the usage need increases or down if resources are not being used.

Security can improve due to centralization of data and increased security-focused resources, but concerns can persist about loss of control over certain sensitive data, and the lack of security for stored kernels. The complexity of security is of course greatly increased when data is distributed over a wider area or over a greater number of devices, as well as in multi-tenant systems shared by unrelated users. In addition, user access to security audit logs may be difficult or impossible. Private cloud installations are in part motivated by users' desire to retain control over the infrastructure and avoid losing control of information security.

2.2 Virtualization

Virtualization refers to the creation of a virtual resource such as a server, desktop, operating system, file, storage or network and it has been a part of the IT landscape for decades now. Today it can be applied to a wide range of system layers, including the operating system-level virtualization, hardware-level virtualization and server virtualization. The main goal of virtualization is to manage workloads by radically transforming traditional computing to make it more scalable.

The most common form of virtualization is the operating system-level virtualization. In operating system-level virtualization, it is possible to run multiple operating systems on a single piece of hardware. Virtualization technology involves separating the physical hardware and software by emulating hardware using software. When a different OS is operating on top of the primary OS by means of virtualization, it is referred to as a virtual machine.

A virtual machine is nothing but a data file on a physical computer that can be moved and copied to another computer, just like a normal data file. The computers in the virtual environment use two types of file structures: one defining the hardware and the other defining the hard drive. The virtualization software, or the hypervisor, offers caching technology that can be used to cache changes to the virtual hardware or the virtual hard disk for writing at a later time. This technology enables a user to discard the changes done to the operating system, allowing it to boot from a known state[2].

2.3 Containers

Unlike hypervisor virtualization, where one or more independent machines run virtually on physical hardware via an intermediation layer, containers run in user space on top of an operating system's kernel. As a result, container virtualization is often called operating system-level virtualization. Container technology allows multiple isolated user space instances to run on a single host. Containers have been deployed in a variety of use cases. They are popular for hyperscale deployments of multi-tenant services, for lightweight sandboxing, and, despite concerns about their security, as process isolation environments. Indeed, one of the more common examples of a container is a chroot jail, which creates an isolated directory environment for running processes. Attackers, if they breach the running process in the jail, then find themselves trapped in this environment and unable to further compromise a host[3].

Containers are generally considered a lean technology because they require limited overhead. Unlike traditional virtualization or paravirtualization technologies, they do not require an emulation layer or a hypervisor layer to run and instead use the operating system's normal system call interface. This reduces the overhead required to run containers and can allow a greater density of containers to run on a host. Despite their history containers haven't achieved large-scale adoption. A large part of this can be laid at the feet of their complexity: containers can be complex, hard to set up, and difficult to manage and automate. Docker aims to change that.

Proponents of containerization point to gains in efficiency for memory, CPU and storage compared to traditional virtualization and physical application hosting. Without the overhead required by VMs, it is possible to support many more application containers on the same infrastructure.

Portability is another benefit. As long as the operating system is the same across systems, an application container can run on any system and in any cloud without requiring code changes. There are no guest operating system environment variables or library dependencies to manage. Reproducibility is another advantage to containerizing applications, which is one reason why container adoption often coincides with the use of a DevOps methodology[4].

2.3.1 Docker

Docker is an open-source engine that automates the deployment of applications into containers. Docker adds an application deployment engine on top of a virtualized container execution environment. It is designed to provide a lightweight and fast environment in which to run a code as well as an efficient workflow to get that code from e.g a laptop to a test environment and then into production. Docker relies on a copy-on-write model so that making changes to an application is also incredibly fast: only what the user needs to change gets changed. The user can then create containers running applications. Most Docker containers take less than a second to launch. Removing the overhead of the hypervisor also means containers are highly performant so that more of them can be packed into hosts making the best possible use of compute resources. With Docker, Developers care about their applications running inside containers, and Operations cares about managing the containers. Docker is designed to enhance consistency by ensuring the environment in which developers write code matches the environments into which your applications are deployed.

Docker aims to reduce the cycle time between code being written and code being tested, deployed, and used. It aims to make applications portable, easy to build, and easy to collaborate on. Docker also encourages serviceoriented and microservices architectures. Docker recommends that each container run a single application or process. This promotes a distributed application model where an application or service is represented by a series of inter-connected containers. This makes it easy to distribute, scale, debug and introspect your applications. The core components that compose Docker are the Docker client and server, also called the Docker Engine, the Docker Images, the Registries and the Docker Containers

2.3.1.1 Docker client and server

Docker is a client-server application. The Docker client talks to the Docker server or daemon, which, in turn, does all the work. Sometimes the Docker daemon is referred to as Docker Engine. Docker ships with a command line client binary, docker, as well as a full RESTful API to interact with the daemon. Docker daemon and client can run on the same host or connect a local Docker client to a remote daemon running on another host.

2.3.1.2 Docker Images

Images are the building blocks of the Docker world. Containers are launched from images. Images are the "build" part of Docker's life cycle. They are a layered format, using Union file systems, that are built step-by-step using a series of instructions. Images are considered to be the "source code" for containers. They are highly portable and can be shared, stored, and updated.

2.3.1.3 Registries

Docker stores the images in registries. There are two types of registries: public and private. Docker, Inc., operates the public registry for images, called the Docker Hub. A user creates an account on the Docker Hub and uses it to share and store images. Docker Hub also contains, at last count, over 10,000 images that other people have built and shared. Images can also be stored privately on Docker Hub. These images might include source code or other proprietary information that is stored securely or shared with other members of a team or organization. A user can also run his own private registry. This allows to store images behind a firewall, which may be a requirement for some organizations.

2.3.1.4 Containers

Docker helps a user build and deploy containers inside of which applications and services are packed. As mentioned before, containers are launched from images and can contain one or more running processes. One can think about images as the building or packing aspect of Docker and the containers as the running or execution aspect of Docker. A Docker container consists of an image format, a set of standard operations and an execution environment.

Docker doesn't care about the contents of the container when performing actions. Each container is loaded the same as any other container whether it is a web server, a database, or an application server. Docker also doesn't care where containers are shipped from: they can be built on a laptop, upload to a registry, then download to a physical or virtual server, test, deploy to a cluster of a dozen hosts, and run. It can build local, self-contained test environments or replicate complex application stacks for production or development purposes. The possible use cases are endless.

2.4 Data Procesing Platforms

2.4.1 Spark

Apache Spark alongside with Hadoop and Storm is one of the most popular frameworks for large scale data prosessing under the wing of the Apache Software Foundation. Spark is a general-purpose distributed data processing engine that is suitable for use in a wide range of circumstances. Programming languages supported by Spark include: Java, Python, Scala, and R. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale. Tasks most frequently associated with Spark include batch jobs across large data sets, processing of streaming data from sensors, IoT, or financial systems, and machine learning tasks[6]. Basically Spark is a framework which provides a number of interconnected platforms, systems and standards for Big Data projects.

Spark has proven very popular and is used by many large companies for huge, multi-petabyte data storage and analysis. This has partly been because of its speed. Last year, Spark set a world record by completing a benchmark test involving sorting 100 terabytes of data in 23 minutes - the previous world record of 71 minutes being held by Hadoop[7].

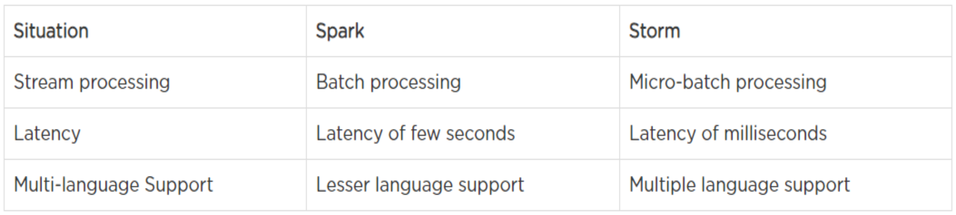
2.4.2 Storm

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate. Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.

Apache storm is an open-source distributed real-time computational system for processing data streams. Similar to what Hadoop used to do for batch processing, Apache Storm does for unbounded streams of data in a reliable manner. Able to process over a million jobs in fraction of a second on a node Integrated with Hadoop to harness higher throughputs Easy to implement and can be integrated with any programming language.

Apart from other projects of Apache such as Hadoop and Spark, Storm is one of the star performers in the field of data analysis. Companies can get benefitted as this technology facilitates multiple applications at once.



Situation	Spark	Storm
Stream processing	Batch processing	Micro-batch processing
Latency	Latency of few seconds	Latency of milliseconds
Multi-language Support	Lesser language support	Multiple language support

There are two types of nodes in Storm cluster:

The master node of Storm runs a daemon called ‘Nimbus’, which is similar to the ‘Job Tracker’ of Hadoop cluster. Nimbus is responsible for distributing codes, assigning tasks to machines and monitoring their performance.

Similar to the master node, the worker node also runs a daemon called ‘Supervisor’ which is able to run one or more worker processes on its node. The Nimbus node is responsible for assigning the work load to the supervisor nodes, and starts and stops the worker processes when required. Every worker process runs a specific set of topology which consists of worker processes working around machines. Since Apache Storm does not have the abilities to manage its cluster state, it depends on Apache Zookeeper for this purpose. Zookeeper facilitates communication between Nimbus and Supervisors with the help of message acknowledgements, processing status, etc.

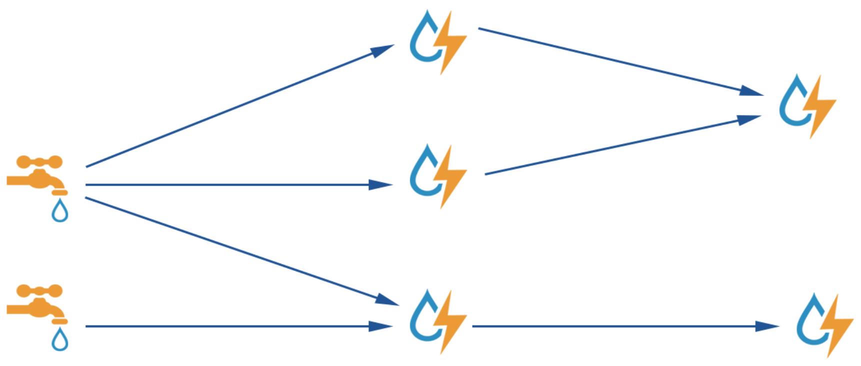
There are basically four components/abstractions which are responsible for performing the tasks:

Storm topology can be described as a network made of spout and bolts. It can be compared to the Map and Reduce jobs of Hadoop. Spouts are the data stream source tasks and Bolts are the accrual processing tasks. Every node in the network consists of processing logic’s and links to demonstrate the ways in which data will pass and the processes will be executed. Each time a topology is submitted to the storm cluster, Nimbus consults the supervisor nodes about the worker nodes. An example of a Storm's topology is presented in figure 2.1.

Stream is one of the basic abstractions of the storm architecture is stream which is an unbounded pipeline of tuples. A tuple can be defined as the fundamental component in the Storm cluster containing a named list of the values or elements.

Spout is the entry point or the source of streams in the topology. It is responsible for getting in touch with the actual data source, receiving data continuously, transforming those data into actual stream of tuples and finally sending them to the bolts to be processed.

Bolt keep the logic required for processing. These are responsible for emitting the streams for processing by other bolts and saving or sending the data for storage. These are capable of running functions, filtering tuples, aggregating and joining streams, linking with database, etc[8].



2.1 Apache Storm's Topology

2.4.3 Flink

Apache Flink is a distributed data processing platform for use in big data applications, primarily involving analysis of data stored in Hadoop clusters. Supporting a combination of in-memory and disk-based processing, Flink handles both batch and stream processing jobs, with data streaming the default implementation and batch jobs running as special-case versions of streaming applications. Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale. Any kind of data is produced as a stream of events. Credit card transactions, sensor measurements, machine logs, or user interactions on a website or mobile application, all of these data are generated as a stream. Data can be processed as unbounded or bounded streams.

Unbounded streams have a start but no defined end. They do not terminate and provide data as it is generated. Unbounded streams must be continuously processed, i.e., events must be promptly handled after they have been ingested. It is not possible to wait for all input data to arrive because the input is unbounded and will not be complete at any point in time. Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness.

Bounded streams have a defined start and end. Bounded streams can be processed by ingesting all data before performing any computations. Ordered ingestion is not required to process bounded streams because a bounded data set can always be sorted. Processing of bounded streams is also known as batch processing.

Apache Flink excels at processing unbounded and bounded data sets. Precise control of time and state enable Flink's runtime to run any kind of application on unbounded streams. Bounded streams are internally processed by algorithms and data structures that are specifically designed for fixed sized data sets, yielding excellent performance.

The core Flink runtime supports a pipelined streaming architecture; it also offers a built-in method to support iterative data processing for machine learning and other analytics applications. Dedicated APIs and libraries are provided for development of machine learning programs, as well as string handling, graph processing and other uses. Another API is focused on Hadoop application integration.

2.5 Machine Learning

"Machine learning is based on algorithms that can learn from data without relying on rules-based programming."

McKinsey & Co.

Machine Learning is an application of artificial intelligence (AI) that gives systems the ability to perform tasks after automatically learning from data without relying on rule-based programming. It focuses on learning and improving from data, and then make determinations or predictions on future data.

Learning methods are categorized as supervised or unsupervised. The first one describes learning algorithms that apply what has been already learnt, to new data, using labeled examples in order to predict future events. Contrary, unsupervised learning is used when the data being used are neither classified nor labeled. Unsupervised machine learning studies how systems can form a function to describe a structure from unlabeled data. The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data. There is reinforcement learning, as well, which is about taking suitable action to maximize reward in a particular situation. It is employed by various software components to find the best possible behavior or path it should take in a specific situation.

2.5.1 Anomaly Detection

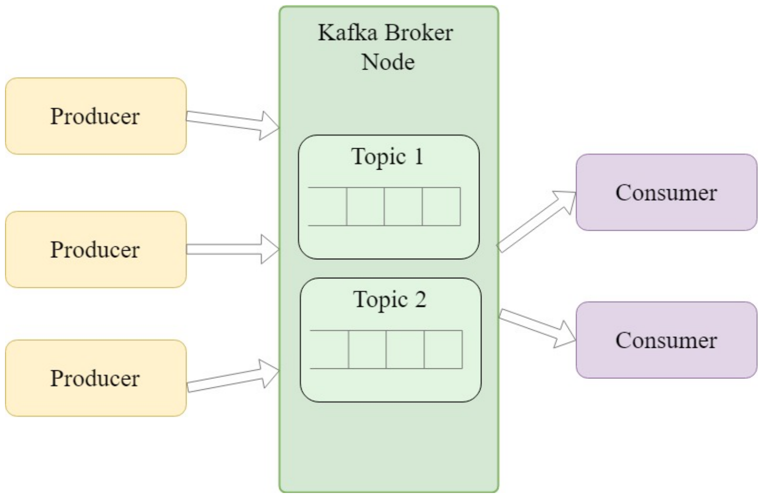
"*Observation which deviates so much from other observations as to arouse suspicion it was generated by a different mechanism*"

Hawkins(1980)

Anomaly detection describes the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data[9]. Typically, anomalous data can be connected to some kind of problem or rare event and it has many applications in business, from intrusion detection (identifying strange patterns in network traffic that may could signal a hack) to system health monitoring, and from fraud detection in credit card transactions to fault detection in operating environments. Density-based, Clustering-based and Support Vector Machine-based anomaly detection are some popular machine learning-based techniques for anomaly detection.

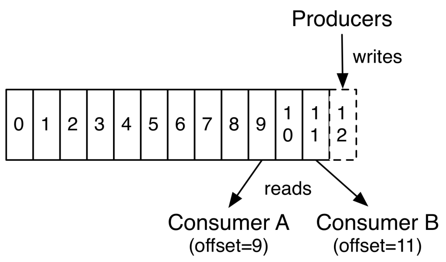
2.6 Apache Kafka

Apache Kafka is a distributed event streaming platform capable of handling a big number of events. Initially it was conceived as a messaging queue and it quickly evolved from messaging queue to a full-fledged event streaming platform.



2.2 Apache Kafka messaging system

Kafka uses topics for its utility. A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it. For each topic, the Kafka cluster maintains a partitioned log that can be described as a queue:



2.3 Kafka's topic instance

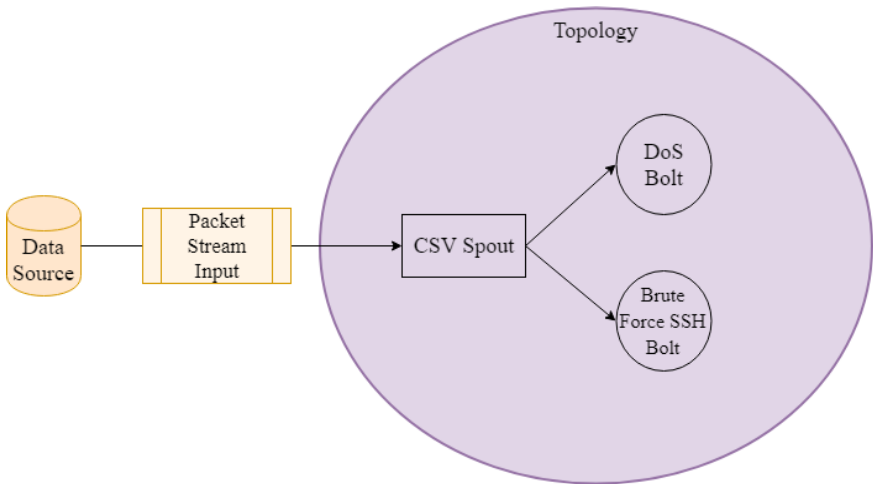
Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each record goes to one of them; in publish-subscribe the record is broadcast to all consumers. Each of these two models has a strength and a weakness. The strength of queuing is that it allows you to divide up the processing of data over multiple consumer instances, which lets you scale your processing. Unfortunately, queues aren't multi-subscriber– once one process reads the data it's gone. Publish-subscribe allows you broadcast data to multiple processes, but has no way of scaling processing since every message goes to every subscriber. In this project, Kafka's queuing service is being employed.

3. Amanda Application

3.1 Architecture

This chapter describes the architecture of Amanda, identifies its basic components and provides an in depth analysis of its functionality. This architecture is mandated by the use of Apache Storm as a distributed processing framework and it consists of a number of slave nodes (workers), a master node and a Zookeeper node along with an Apache Kafka component used as a queue for the input data. Apache Storm abstracts away the complexities of talking to different machines on a network allowing developers to focus on application functionality and leave most of the complexities of message passing, ensuring fault tolerance and distribution up to the framework. Our choice of Apache Storm as a distributing processing framework was motivated by its emphasis on real-time application support. Most other distributed processing frameworks like Apache Hadoop and Apache Spark do a terrific job in supporting applications that require batch processing, but provide no support for stream-based applications that need to work on data in real-time.

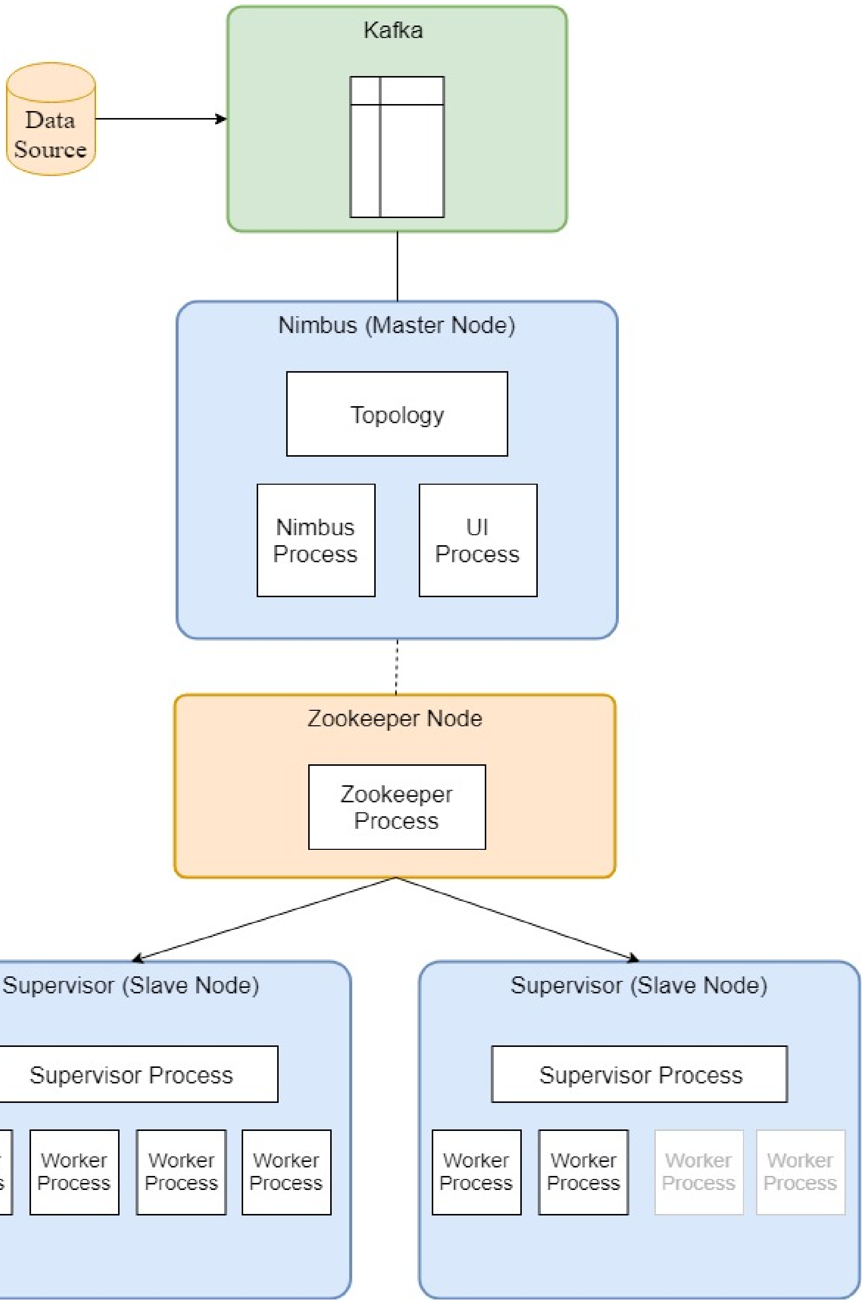
Topology is the top-level abstraction that you submit to Storm clusters for execution and includes the spouts and the bolts. It is a graph of stream processes where each node is a spout or bolt. Links between nodes inside the topology describe how tuples should be passed around. Each node in the topology of a Storm cluster executes in parallel. Our topology is implemented with the parallelism factor of 2. A topology runs forever, or until it is killed. Additionally, Storm automatically reassigns any failed tasks and guarantees that there will be no data loss, even if machines go down and messages are dropped. On our application we have used two bolts, each one for a different type of intrusion: Dos Bolt (Denial of Service Bolt) and Brute Force SSH Bolt.



3.1 Storm's topology

3.2 Application

The basic workflow of our system in terms of storm components is as outlined below. The incoming packet stream corresponds to an abstraction of the incoming packets, manifesting itself in code in the form of a Java stream created using the CSV file outputted during the pre-processing stage. In order to simulate a real-time system that has incoming data round the clock, we read the data stream in tuples, waiting for a few milliseconds after each read in order to make sure the previous tuple was processed. The CSV Spout functions as a buffer converting the incoming Java data stream into tuples that can be understood by the Storm subsystem. It is written entirely in Java, the code for which can be found in the Appendix. The tuples are then sent ahead to the two bolts, with each tuple replicated 2 times, one for each bolt. Each bolt corresponds to a processing entity that performs the classification task in addition to the tasks of loading the persisted classifier into main memory and converting each tuple into a data format compatible with the interface of the classifier.



3.2 Amanda's architecture

Every Bolt is a logical abstraction representing one of the two classifiers built during training. Each Bolt serves to identify attack instances of a particular input class, with the classes being restricted to the two types of attacks in the dataset.

The system consists of one master node (known as Nimbus in storm terminology), a number of slave nodes (ideally 2 or multiples of 2, for each classifier) and one Zookeeper node, which is responsible for maintaining the state of the system and interfacing between the master node and the slaves. The topology and the input Spout are both written using the Java programming language. This choice was dictated by Storm's built in support for Java. The Bolts are written using the Python programming language.

Apache Storm does the logical to physical mapping at runtime. It does this via the topology construct built into storm. A machine in a Storm cluster may run one or more than one worker processes for one or more topologies. Each worker process runs executors for a specific topology. Maven is commonly used for building Storm topologies, and it requires a pom.xml file that defines various configuration details and the project dependencies.

3.3 Supervised Learning

3.3.1 Dataset Selection

For our application we used a very popular dataset, NSL-KDD dataset. Data consists of around 5 million tuples with 41 characteristic each that are all labeled. This dataset satisfies the above features:

1. Realistic network traffic
2. Labeled dataset
3. Total interaction capture
4. Complete capture
5. Diverse intrusion scenarios

The dataset also provides a satisfying in size testing set for each intrusion scenario. Our project was adjusted to detect two different types of anomalous behaviour it has been trained upon: Denial-of-service (DoS) and Brute force running an SSH server (Brute Force SSH) attacks:

Denial Of Service: In computing, a denial-of-service (DoS) attack is an attempt to make a machine or network resource unavailable to its intended users.

Brute Force SSH: Brute force which is also known as brute force cracking, is a trial and error method used by application programs to decode encrypted data such as passwords through exhaustive effort (using brute force) rather than employing intellectual strategies. Brute force SSH refers to the process of using such an attack to break into systems running an SSH server.

The availability of a world-class labelled intrusion detection dataset steered us in the direction of using supervised machine learning as opposed to unsupervised approaches.

3.3.2 Offline pre-process

3.3.2.1 Classifier Selection

Selecting a suitable classifier is important. Often the hardest part of solving a machine learning problem, choosing the right estimator makes use of the following information for the given problem at hand are the dataset size, whether the dataset is labeled or not, the type of prediction or target classes and the type of data (text, multimedia etc). From the point of view of our implementation we considered to use models that use decision trees as classifiers. The advantage of using decision trees is that they are easy to interpret and explain. Decision trees handle feature interactions and they are non-parametric.

3.3.2.2 Training the model

The use of machine learning entails 'learning' with experience. In Supervised learning, this translates to building a model of the classification function from the data and their associated labels. We have used a model that advocates the use of two separate classifiers for identifying each type of attack to take advantage of the parallel processing techniques inherent in the use of a distributed system, and storm in particular. Each decision tree classifier was trained using a pre-defined ratio of normal packet data and attacking instances, the attacking instances all corresponding to the same attack type.

During the offline process, a training set is inserted as an input and after the application of the classifier it constructs a model algorithm that will evaluate new, non-trained data, to confirm, or not, the deviation of expected normal values in data. This persisted model is then loaded into the files with the code that the workers run. As for the testing, testing datasets can be used. A very efficient way of testing is cross-validation, which samples the training test and then uses the samples for the testing.

The process of deciding whether the results quantifying hypothesized relationships between variables, are acceptable as descriptions of the data, is known as validation. As there is never enough data to train your model, removing a part of it for validation poses a problem of underfitting. Reducing the training data risks to miss important patterns in the data set, which in turn increases error. What is required is a method that provides ample data for training the model and also leaves ample data for validation and K-Fold cross validation does that.

During the K Fold cross validation, the data is divided into k subsets. Now the holdout method is repeated k times, such that each time, one of the k subsets is used as the test set/ validation set and the other k-1 subsets are put together to form a training set. The error estimation is averaged over all k trials to get total effectiveness of our model. As can be seen, every data point gets to be in a validation set exactly once, and gets to be in a training set k-1 times. This significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in validation set. Interchanging the training and test sets also adds to the effectiveness of this method [10].

3.4 Performance Evaluation

As for the efficiency of the working application, the schema of the performance of the running containers representing the workers of Storm cluster of our application is following.

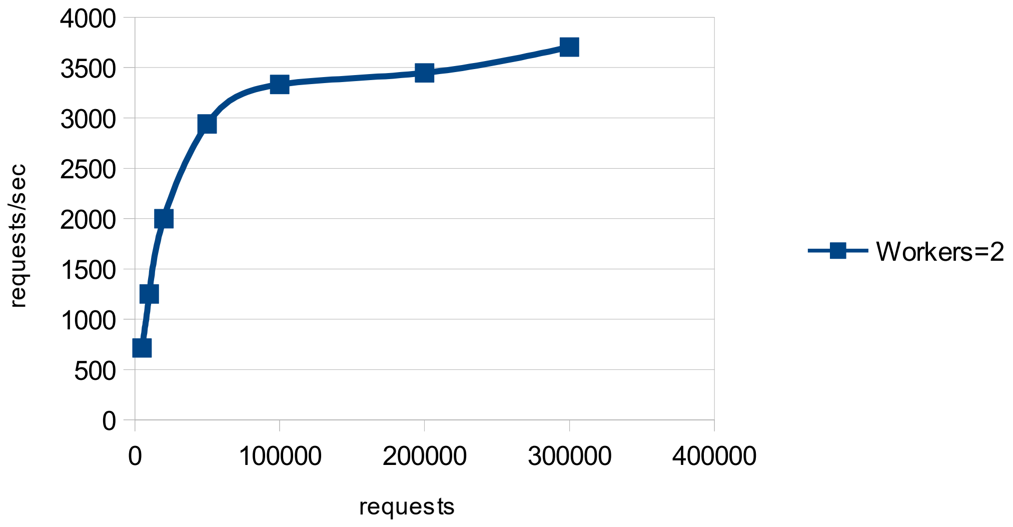
During the experimental process, we simulated the real-time data system by making the input handling system “sleep” for 100ms after every instance of tuples is inserted to the application as an input. This technique can secure that the processing of the current input will execute only after the previous set of tuples has been accepted in the system, in order to avoid the case of batch data as an input. In order to monitor the performance of the application, we call the Kafka producer component and pass as variables the number of the requests that hit the input of the application along with the time in milliseconds for the system to wait between producing each distinct message. During the monitoring process, the number of workers has been increasing, after killing the topology each time in order to rebuild the project with the new topology submitted.

Analyzing the graph above, we examine the performance of the distribution of workers on our Storm cluster which, on some points, is also a container scaling issue. While using two workers with the parallelism factor equal to 2 we can see the performance upturn, as the number of the requests/sec increase while hitting the

Initially, the configuration of the application in the topology used two workers to run on the Storm, which is translated in building a supervisor container with two worker threads running.

requests	sec	requests/sec
5000	7	714.29
10000	8	1250.00
20000	10	2000.00
50000	17	2941.18
100000	30	3333.33
200000	58	3448.28
300000	81	3703.70

Workers=2



The horizontal axis shows the requests (number of tuples) accessed, while the horizontal axis represents the access rate for each request in request per second. Starting with a relatively low number of requests, we can see the improvement of the response time, while when increasing the requests, the rate of this increase is lower because of the big load of the requests. The number of served requests per second is then, still increasing, on a lower rate though. This pattern is followed for each of the four different cases which were examined.

Following, we reconfigured the number of deployed workers setting it to four. After rebuilding the topology, the cluster still uses only one supervisor container to run the tasks, as each supervisor is capable of deploying up to four workers.

requests	sec	requests/sec
5000	4	1250.00
10000	6	1666.67
20000	9	2222.22
50000	17	2941.18
100000	32	3125.00
200000	60	3333.33
300000	76	3947.37

workers=4

Changing the number of the workers to six, leads to starting another one supervisor in order to run the extra two new workers to run. Of course the topology has to be rebuilt. We now have two supervisor containers running along with the nimbus container inside Storm's architecture. The results of the requests hit on the Kafka component while six workers threads are running is following.

requests	sec	requests/sec
5000	3	1666.67
10000	5	2000.00
20000	8	2500.00
50000	16	3125.00
100000	28	3571.43
200000	57	3508.77
300000	74	4054.05

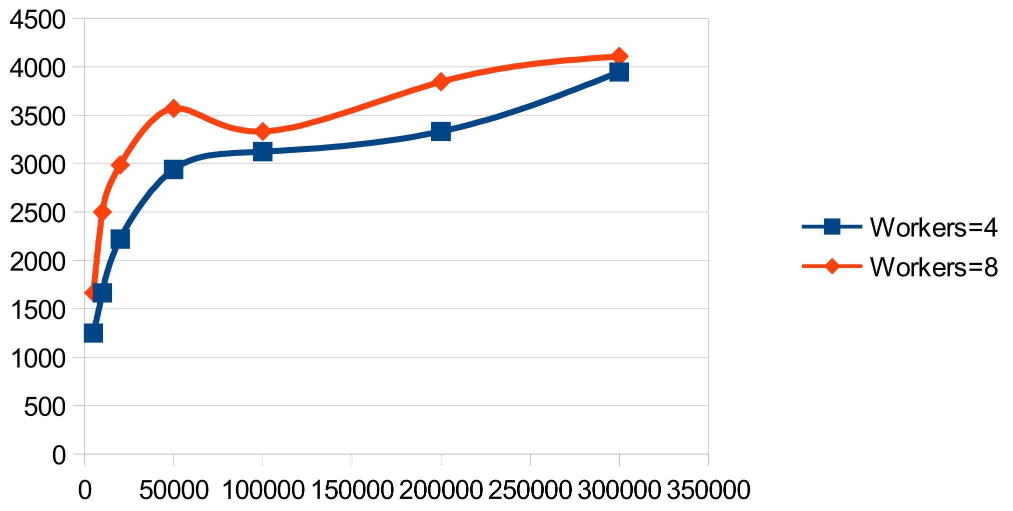
workers=6

In the final stage of the performance monitoring, we adjusted the number of workers to eight. The system is still using two containers who can of course afford the eight workers.

requests	sec	requests/sec
5000	3	1666.67
10000	4	2500.00
20000	11	2987.00
50000	14	3571.43
100000	30	3333.33
200000	52	3846.15
300000	73	4109.59

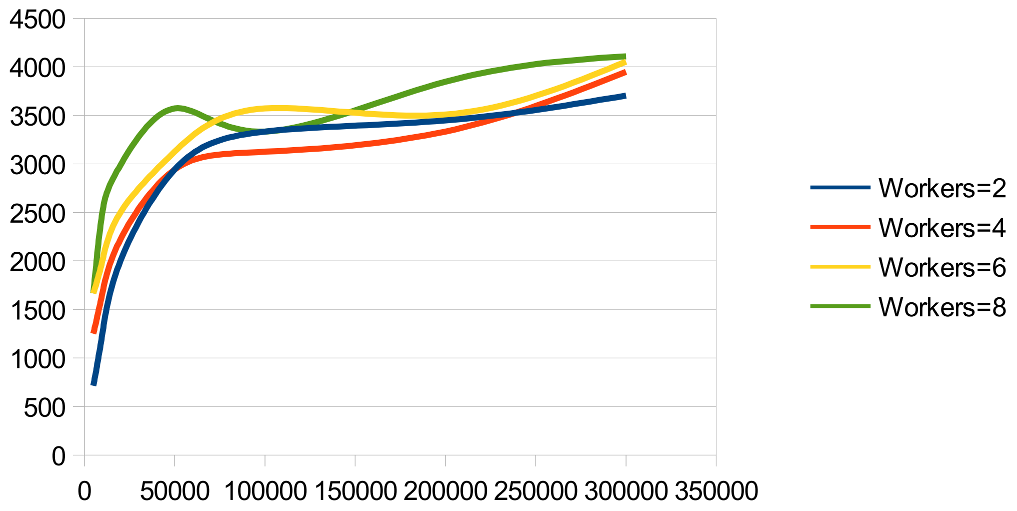
workers=8

The performance of the two cases were each supervisor container uses the maximum number of workers (workers=4 and workers=8) were analyzed below.

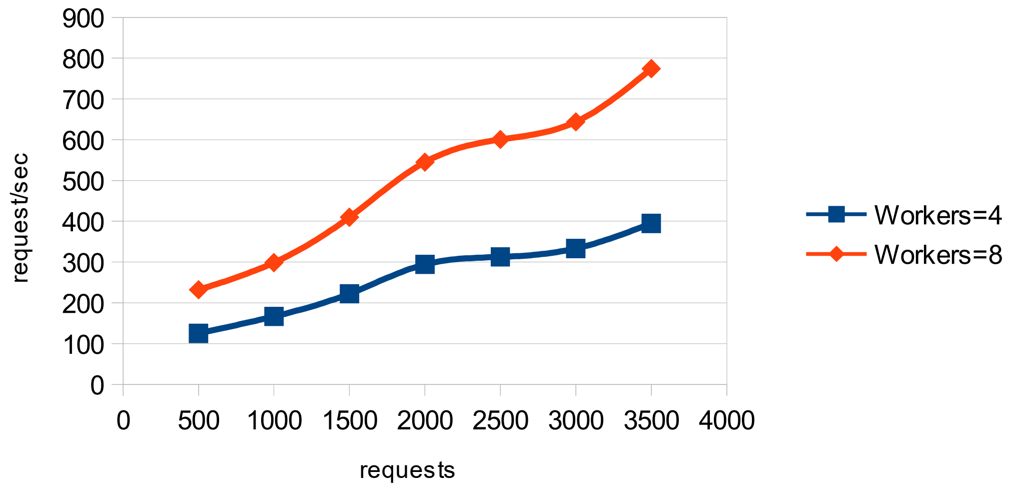


We can see that the parallelism of the processes running in the threads in docker containers is a significant factor that makes the deployment of 8 workers (two supervisor containers running in parallel) giving better response time for the same number of requests.

The following schema contains the four different graphs of the cases mentioned above:



A second stage of monitoring the performance of the application, based on the response time of the system given a different number of requests as an input was focused on a lower given number of requests to as an input to the system.



The improvement on the performance of the application while deploying an extra supervisor container, while receiving a lower number of requests, is presented in the graph above. What can be occurred is the better response time while using a double number of workers distributed in a double number of supervisor containers.

4. Conclusion and Future work

4.1 Conclusion

In this thesis we introduced, implemented and tested a concept architecture for deploying the Apache Storm cluster and the Apache Kafka queue on Docker containers running machine learning classifiers for the anomaly detection of real-time stream data input. Summarizing, the contribution of this work was to prove the enhancing performance of the system while deploying a relatively bigger number of workers that results to reduction of resources usage. Therefore, scalability can prove feasible and show better results on the response time of the system.

4.2 Future Work

For future work, we can focus the research on scaling our application. A promising concept will be to execute cluster-level operations such as scale up or scale down without affecting the already deployed applications. A good way to achieve this is the use of the components of Docker volumes which can support the scalability, which is already supported in Docker, but without having to rebuild the project and stop the running application. Docker volumes are mainly used for backup purposes as the user is able to save the data of the running container so it can be rebuilt later and continue its process.

The application we built is deployed locally on a machine which runs the Docker and its components offline. A significant contribution to the built system suggests the online deployment of the application hosted on a server that will be able to support an integrated online application to achieve the distributed functionality of detecting anomalous behavior of data streams in systems.

Our system is designed to monitor the functionality of running the anomaly detection algorithm in a Storm cluster. Yet, it can be easily extended, not only to track the not normal data behavior but also to take preventive action after the anomaly detection utility, including things like making the application run online or changing firewall configurations on a real time basis to blacklist connections belonging to certain IP addresses that are thought to belong to the attacking entities. Additionally, we can add unsupervised learning methods to predict novel attacks such as with the use of Neural Networks, not only the ones that our model is trained to.

By default Storm distributes workers for each topology across the cluster. Additional features like the ability to change the isolation configuration dynamically can be implemented in order to reduce query topologies to a single worker per topology which is expected to perform better.

5. References

[1] "What is Cloud Computing?" Amazon Web Services, 2013-03-19, Retrieved 2013-03-20.

[2] https://www.techopedia.com/definition/719/virtualization

[3] The Docker Book: Containerization Is the New Virtualization By James Turnbull

[4] https://searchitoperations.techtarget.com/definition/application-containerization-app-containerization

[5] Oestreich, Ken (2010-11-15). "Converged Infrastructure". CTO Forum. Thectoforum.com. Archived from the original on 2012-01-13. Retrieved 2011-12-02.

[6] "Cloud Computing: Clash of the clouds". The Economist. 2009-10-15. Retrieved 2009-11-03.

[7] https://www.bernardmarr.com/default.asp?contentID=1079

[8] https://intellipaat.com/blog/what-is-apache-storm/

[9] Zimek, Arthur; Schubert, Erich (2017), "Outlier Detection", Encyclopedia of Database Systems, Springer New York

[10] https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f

"A Study of CrossValidation and Bootstrap for Accuracy Estimation and Model Selection", Ron Kohav, International Joint Conference on Articial Intelligence (IJCAI), 1995

"Real-Time Network Anomaly Detection System Using Machine Learning" , Shuai Zhao, Mayanka Chandrashekar, Yugyung Lee, Deep Medhi, 2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)

"An overview of anomaly detection techniques: Existing, solutions and latest technological trends", Animesh Patcha, Jung-Min Park, Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University

Unsupervised Learning, Peter Dayan, Wilson, RA & Keil, F, editors. The MIT Encyclopedia of the Cognitive Sciences

"A Survey on Unsupervised Machine Learning Algorithms for Automation,

Classification and Maintenance", Memoona Khanum, Tahira Mahboob, Warda Imtiaz, Humaraia Abdul Ghafoor, Rabeea Sehar, International Journal of Computer Applications (0975 – 8887), Volume 119 – No.13, June 2015

https://storm.apache.org/
https://zookeeper.apache.org/
https://kafka.apache.org/
https://www.docker.com/
https://flink.apache.org/