University of Science and Technology in Bydgoszcz

Institute of Telecommunications and Computer Science

PhD Thesis

Sebastian Łaskawiec

Effective solutions for high performance communication in the cloud

Supervisors
Michał Choraś
Tomasz Andrysiak

Bydgoszcz 2020

Acknowledgments

First and foremost, I would like to sincerely thank my supervisors, dr hab. inż. Michał Choraś, dr hab. inż. Tomasz Andrysiak and dr hab. inż. Rafał Kozik for a lot of support and guidance throughout the time of doing the research, as well as writing the PhD thesis.

I have been extremely privileged to work with an amazing team focusing on delivering the best open source caching solution in the world. I have always received a lot of positive and constructive feedback from Tristan Tarrant, Gustavo Fernandes, Dan Berindei, Galder Zamarreño, Bela Ban and many others.

I would also like to thank both my managers, Pedro Zapata Fernandez and Bolesław Dawidowicz for helping me to promote Intelligent Operator idea and finding proper balance between work and academic research.

I am deeply grateful to my whole family for motivating me to finish the PhD thesis in a timely manner.

Last but not least, I would like to thank my wife, Aleksandra Łaskawiec for a lot of support and understanding during the endless evenings spent in front of a computer trying out different solutions for a specific problem.

This thesis is dedicated to my father, Waldemar Łaskawiec, who gave me a lot of inspiration to start a PhD but sadly passed away before the completion of this thesis.

Contents

Chapter 1

Introduction

1.1 From terminals to cloud computing

Well known companies, such as Amazon, Apple or Facebook, offer their services around the world thanks to the dynamic infrastructure offered by the cloud. This would not be possible without the first data network application written in 1960's [56][61], that had used a network connection between a mainframe computer and a terminal. In the 1970's, with the increasing popularity of network equipment, a new standardization effort was made by the hardware vendors and a new layer of software was created - the middleware. Middleware layer had exposed a standardized Application Programming Interfaces (APIs) to the software engineers allowing them to write programs using the hardware equipment. Only a few years later, in 1989, after transferring the data between a client and a server using the Hypertext Transfer Protocol [110], Timothy John introduced the term "World Wide Web". It is estimated that the Internet had more than 20 million of users 6 years later [115]. The next breakthrough event took place in 2003, when Xen, the first virtualization hypervisor was invented [16]. Starting from that moment, it was possible to run a guest Operation System within a host machine. Only 3 years layer, Amazon announced the Elastic Compute Cloud Beta project, which is known as Amazon EC2 today[8]. Amazon was the first widely-known public Infrastructure-as-a-Service offering. This approach was adopted by the Open Source projects and in 2010 OpenStack [91] project was founded. The Virtual Machine approach was very popular at that point of time. However, many developers noticed that running multiple applications in the same Virtual Machine often leads to problem with conflicting library versions. This particular problem was solved by the Linux Containers project (LXC), which was popularized by the Docker company in 2013. Containers allow to package application, with all dependent libraries and an Operating System together. Once the application is packaged, it is being stored in an immutable image and can be

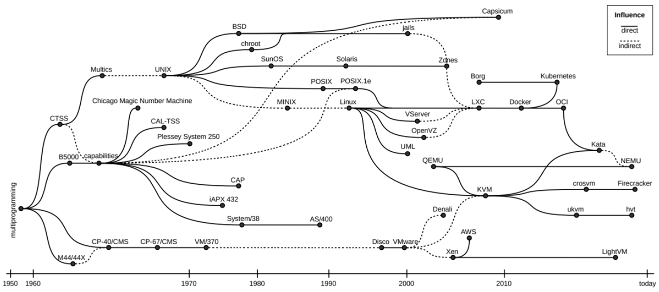


Figure 1.1: Container technology history [90]

run on a host machine (typically as a Linux process). Open Containers Initiative (OCI) takes containers one step further and standardizes APIs used to build, store and run container images. Maintaining such an ecosystem of different applications running at scale is very challenging. Based on a lot of previous experience with Borg [121], in 2014 Google founded an Open Source project for orchestrating Linux Containers, called Kubernetes [12]. Today, Kubernetes is the most popular container-based cloud orchestrator [105]. It has been used as a core for private, public and hybrid cloud offerings, such as:

- Tectonic (by CoreOS)
- Google Container Engine (by Google)
- or OpenShift Online (by Red Hat)

Throughout the history, most of the mentioned technologies had some impact on each other. Figure 1.1 presents interesting relations that explain the connection between Linux Operating System and container technology (through LXC project, Docker containers and finally, standardized OCI initiative).

Even though container-based clouds require different approach for designing applications from the traditional Virtual Machines, they are often picked by the new projects (new projects are also called greenfield projects) and start-up companies. One of the deciding factors of this success is the rapid growth of the Open Source ecosystem around project Kubernetes. Today, it offers hundreds of projects solving many of the common problems, such as application metric collection (project Prometheus), tracing requests inside the cloud (project Jeager or OpenTracing initiative) or Continuous Integration and Continuous Deployment (project Jenkins). Kubernetes has also been designed with extensibility in mind. Therefore, it has often been picked as the foundation for emerging technologies, like Serverless [101].

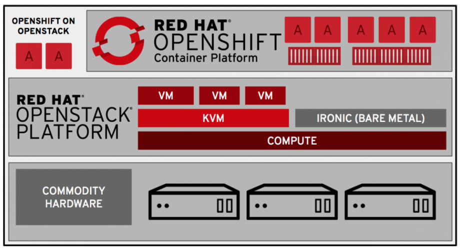


Figure 1.2: OpenShift on OpenStack [2]

1.2 Modern, container-based cloud

Modern cloud environments need to operate as multi-tenant environments. Typically, every physical host is split into smaller worker nodes. This step might involve using some sort of virtualization technology. A worker node is then responsible for running application workload. One of the commercial examples of such a setup might be OpenShift on OpenStack - often referred to as “Triple-O”. Figure 1.2 presents such a setup, where OpenStack platform uses the KVM project to provide Virtual Machines that will be used as OpenShift worker nodes.

In a container-based cloud, application workload is typically run using container technology. This setup has its justification in the security aspect of the cloud. Virtual Machines are run by the host as processes, whereas containers share a kernel with a host. To compromise this setup, an attack vector requires escaping from a sealed environment twice - from a container to a worker node, and then from a worker node Virtual Machine to a physical host. However, since the hardware resources are shared among all application workloads, it is possible to perform a side channel attack (from one application to another). Most of the attack vectors rely on certain hardware vulnerabilities related to speculative execution—including Spectre, Meltdown, Foreshadow, L1TF, and other variants. Even though the security aspect is important for the cloud vendors, it is often outweighed by the performance gains of a multi-tenant, container-based environment for running application workloads. Compared to Virtual Machines, container-based workloads are significantly faster [122] (performance evaluation presented in Figure 1.3 shows container technology almost twice as fast as the Virtual Machines).

Containers technology allows cloud administrators to control application workload by controlling the amount of resources an application consumes and isolating application one from another. This has been achieved by combining CGroups [58], Linux Namespaces [59] and Linux Capabilities [57] in the LXC project. Many Linux Operator System implementations (such as Fedora or Red

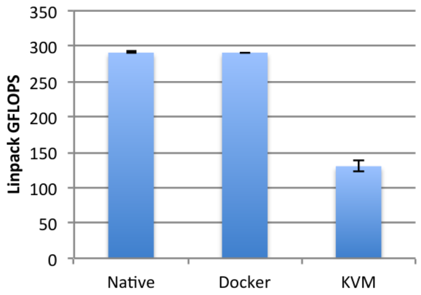


Figure 1.3: Performance comparison between Virtual Machines and Containers [122]

Hat Enterprise Linux) also incorporate security-related projects, such as SELinux [102] or AppArmour [10].

In order to schedule application workloads, a cloud needs an orchestration layer. There are a few solutions competing in this area, including Mesos [9], Docker Swarm [25] and Kubernetes [55]. From practical perspective, Mesos has proven to support large-scale systems (hundreds or thousands of worker nodes). This however comes with the cost of extensive complexity, which is not very practical for small and medium systems. Docker Swarm uses Docker native APIs, which might be considered as both an advantage and disadvantage. Projects such as Podman provide alternative container runtime and are not compatible with Docker Swarm. The last example, Kubernetes, is an opinionated orchestrator created by Google. The implementation contains lots of good practices used by Google's internal cloud, called Borg [107][121]. At the time of writing this thesis, Kubernetes is one of the most popular solutions used for orchestrating container workloads [108] [105].

A typical Kubernetes deployment consists of several worker nodes and a much smaller number of supervisor (also called Master) nodes. A high-level architecture has been shown in Figure 1.4 [65].

A Master node contains several components, including:

- Scheduler - responsible for assigning application workloads to specific worker nodes
- Etcd - a distributed data store, which stores cluster configuration
- API Server - used for reading and modifying cluster (and application) configuration objects
- Controllers - a set of objects used to maintain current cluster state

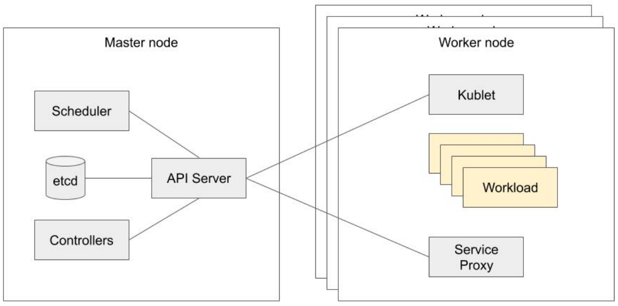


Figure 1.4: Kubernetes Architecture [65]

A worker node contains a much smaller list of components, such as:

- Service Proxy - used for communicating with application workloads
- Kublet - a supervisor process that manages application workload running inside a worker node
- Workload - a set of applications created by developers

A typical workflow for deploying application workload starts with building container image and uploading it to a container registry (one of the commercial examples is project Quay created by CoreOS). The next step is to send an application description to Kubernetes API Server. Depending on special application needs, there are several different constructs that might be used, including “Deployments”, “StatefulSets” or “DaemonSets”. Kubernetes also allows to store application configuration (in a “ConfigMap” object) or confidential data (in a “Secret” object), such as TLS keys and certificates. The final step requires defining application connectivity properties. For routing inside a cluster, one might pick “Service” objects (which as based on Netfilter). For the ingress traffic, there are many different concepts, including an “Ingress” (which is a Reverse Proxy) or “Load Balancer Services”. Once all metadata gets uploaded to the API Server, the Scheduler assigns a worker node to run the workload. The scheduling mechanisms takes a number of different factors into consideration, including node capacity or current load level. Once a node is assigned, a Kublet process downloads the application image and starts the container process.

Separating responsibilities between cluster administrators and developers has proven to be one of the biggest strengths of Kubernetes orchestrator. Each group has its own set of tools required to maintain the system in good health.

Administrators for example might be very interested in project Cincinnati [18], which allows automatic worker node upgrades with zero downtime. On the other hand, developers might be interested in Istio project [46] that enables advanced routing capabilities required by Microservices Architecture. Kubernetes modular architecture allows to customize the cluster to meet all the needs from both the application developer and cluster administrators.

Kubernetes also enables portability of an application workload between different cloud vendors. Apart from some of the corner cases, developers should see no difference between running their workload on Amazon AWS, Microsoft Azure or Google Compute Cloud. Most, if not all, of the application workload metadata should remain the same. This property is often referred to as Cloud Native [47] and has been used as the foundation for creating Cloud Native Computing Foundation (CNCF) [19]. Today, the foundation has been used to govern many Kubernetes-related project, such as Prometheus, Etcd, Helm or gRPC. CNCF's governing board contains people from multiple commercial companies, including Red Hat, Google, Apple, Oracle and many more.

1.3 Interconnected clouds

Making systems highly available and resilient (allowing them to be recovered from a disaster) often requires running application workload on multiple clouds. This may involve using multiple public clouds (where all the nodes are hosted by a cloud vendor) or interconnecting public clouds with private ones (a private cloud runs in a proprietary infrastructure owned by a company typically owning application workload). Below is the list of the commonly used names in the industry for specific scenarios [17]:

- Hybrid cloud - where the public and private clouds are interconnected
- Inter-cloud - a cloud that is capable of transferring the workload to other clouds
- Federated cloud - multiple clouds managed as if they were a single cloud instance
- Multi cloud - more than one public or private clouds together

At the time of writing this thesis, Kubernetes does not support Federated model. There is an ongoing effort to implement it as an extension. One of the most promising solutions is called Submariner [106]. Figure 1.5 shows the high-level architecture of a multi cloud system based on Kubernetes. The aim of the project is to allow containers deployed in two different clusters communicate with each

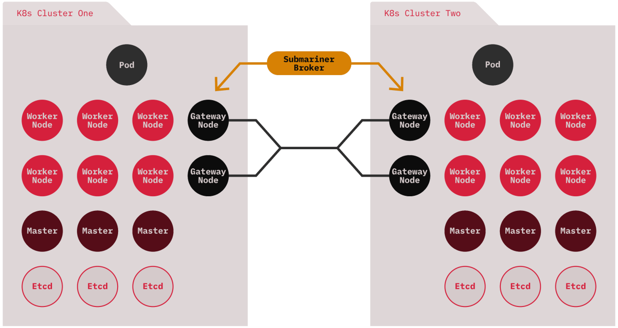


Figure 1.5: Submariner architecture [106]

other in a secured way. However, at this moment, Submariner does not support any type of DNS-based discovery, so forming a cluster of an application workload might be difficult.

Solutions presented in this thesis do not take Submarier into consideration. At the time of writing, it is considered as an immature and unstable project; not ready to be used in the production.

1.4 Recent problems in cloud environments

In recent years, cloud computing is being massively adopted by commercial companies. Each branch of the industry has its own limitations and requirements regarding the deployment and operational model for cloud computing. Most of the challenges might be divided into the following groups [1]:

- Application deployment strategies
- Authentication and authorization (both users and services deployed in the cloud)
- Managing state in the applications
- Application auditing
- Application auto-scaling
- Application configuration management

[1] Dedicated groups have been inspired by Kubernetes Special Interest Group alignment: https://github.com/kubernetes/community/blob/master/sig-list.md

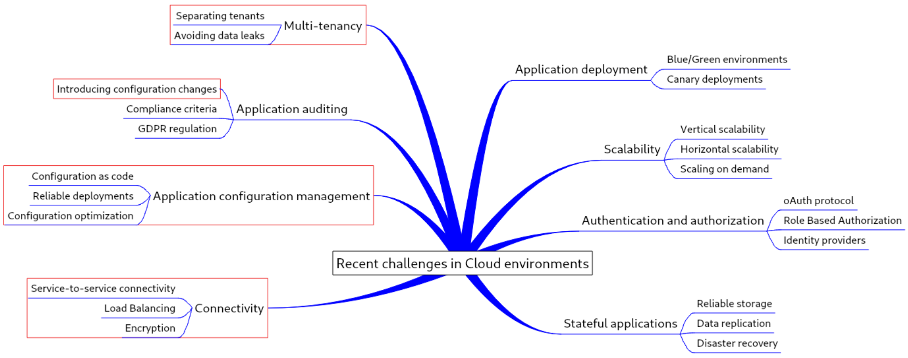


Figure 1.6: Recent cloud challenges

- Networking and connectivity
- Multi-tenancy support

Among other groups of challenges, there is one very specific that is connected to all the other aspects - Security. StackRox report mentions nearly half of the surveyed companies delayed moving their application into production because of the cloud security considerations. What is even more worrisome, 94 percent of respondents admitted to experiencing a security incident in the last 12 months. The vast majority of incidents were connected to misconfiguration (either on container orchestration or application level). The same research also show there is the growing demand on interconnecting clouds, especially in public domain (multiple public clouds used for deploying a large system) [105].

Security related challenges might be divided into the following groups [94]:

- Transport security - providing confidentiality and integrity of data sent to and from an application
- Protecting the data against side-channel attacks - making sure private resources cannot leak from the cloud provider infrastructure
- Protecting the data against leakage - addressed on many levels, protecting data from being stolen

Most of the problems are connected with each other. Nevertheless, some typical challenges were gathered with corresponding groups in Table 1.1. Figure 1.6 represents a diagram of recent challenges with typical examples. Red boxes represent the research areas covered in this thesis.

The solutions proposed in this thesis relate to the last three groups. To some degree, they also correspond to the security aspect, especially data encryption and authenticating clients using a certificate.

Challenge group	Typical examples of the challenge group
Application deployment strategies	Deploying multiple replicas of an application
Authentication and authorization	Authenticating application users, authorizing interaction between deployed microservices
Managing state in the applications	Storing data between application restarts
Application auditing	Identifying who and when changed the application configuration
Application auto-scaling	Under what circumstances an application should be scaled out / scaled in
Application configuration management	**Managing application configuration**
Networking and connectivity	**Using custom binary protocols in the cloud**
Multi-tenancy support	**Dealing with multiple clients using the same application**

Table 1.1: Recent challenges in the cloud environment

Chapter 2

Aims of the thesis

The research aim of this thesis is to propose solutions for improving overall communication performance with applications deployed in the cloud.

In order to reach that goal, the following tasks were defined:

1. To propose new methods for separating data between tenants and lower overall memory footprint of the server. This includes designing a new method for using TCP connection properties to identify individual clients and designing a new method for accessing the client data from the outside of the cloud.
2. To propose new methods for exposing clustered applications hosted in the cloud for outside consumption, which includes designing an improved solution for enabling client side load balancing for the server deployed in the cloud.
3. To propose new solutions for switching communication protocols for the application deployed in the cloud. The solution requires designing new methods for switching to custom binary protocols reusing the same TCP connection.
4. To propose new solutions for application configuration management and finding common configuration mistakes. The proposed system should also help maintain configuration for optimized performance and lowest possible memory footprint.

The thesis statement is as follows (Listing 2.1):

It is possible to improve communication performance, defined by either throughput or latency, and lower the memory consumption by using new solutions and algorithms for protocol negotiationand client side load balancing between a client application and a service deployed in the cloud.

Figure 2.1: Thesis statement

2.1 Scientific achievements

The major contributions for the thesis are the four solutions improving network communication performance, lowering application footprint, as well as helping optimizing application configuration:

1. The proposed solution for service name indication for multi-tenancy routing in cloud environments.
2. The proposed solution for identifying tenant using TLS/SNI Hostname field.
3. The proposed solution for exposing clustered applications deployed in the cloud.
4. The proposed solution for automatic detection of application misconfiguration.

The following articles published in Polish and international journals prove that the solutions in this thesis are valuable from the academic point of view and the accessed challenges are valid:

1. Łaskawiec S., The evolution of Java based software architectures, Journal of Cloud Computing Research, 2016.
2. Łaskawiec S., Choraś M., Considering service name indication for multi-tenancy routing in cloud environments, International Conference on Image Processing and Communications, 2016.
3. Łaskawiec S., Choraś M., Kozik R., Switching Network Protocols to Improve Communication Performance in Public Clouds, International Conference on Image Processing and Communications, 2018.
4. Łaskawiec S., Choraś M., Kozik R., New solutions for exposing clustered applications deployed in the cloud, Cluster Computing, Volume 22, Issue 3, pp 829–838, Springer, 2019 (IF=1,851).

Apart from the academic research journals, three United States Patents have been filled for the solutions aiming for optimizing application configuration:

1. Secure configuration corrections using artificial intelligence (0816028.00312/20191312) - A proposal of a system capable of finding common configuration mistakes and advising application administrator how to correct them. The proposal leverages Machine Learning techniques to reference configuration (from a Knowledge Base) and metrics with the current system state. Once a configuration mistake is identified, the proposed system makes a suggestion on how to improve it.

2. Secure detection and correction of inefficient application configurations (0816028.00312/20191312) - A system capable of automatic detection and correction of application level configuration. The system leverages Machine Learning techniques trained on the metrics obtained during the testing phase. The system training dataset also consists of application configurations, which are typically tested the during project testing phase in a lab. This approach allows the system to learn upon the best known configuration, incorporating all best practices from application developers.

3. Testing and selection of efficient application configurations (0816028.00321/20191349) - A fully automated system proposal for learning correct application configuration based on the performance metrics and applying them automatically for new deployments. A reference environment, such as a performance testing lab might be used for training the Machine Learning model. However, in this proposal it is not strictly required.

Some parts of the work related to this thesis were presented during many informal meetings, including:

1. A seminar on Memory Consistency Models, presented at the University of Science and Technology in Bydgoszcz in 2016

2. A presentation on JGroups Clustering Framework, presented at Virtual JBoss JUG as well as at Toruń Java User Group in 2015

3. A presentation on Infinispan data grid project presented at JDD Conference as well as at Toruń Java User Group in Kraków in 2015

4. A presentation on Infinispan data grid project dedicated for students, presented at Nokia in 2016

5. A presentation on Linux Containers technology, presented at Toruń Java User Group in 2018

2.2 Structure of the thesis

The thesis contains 3 different solutions for optimizing network connection and 1 solution for an expert system discovering common configuration mistakes in application deployed in the cloud.

Chapter 3 contains a literature overview for all the 4 presented solutions. Along with the solutions available on the market, this chapter also provides necessary background and context for specific solutions. The final section of the chapter provides a discussion on the work related to this thesis.

Chapter 4 introduces the first proposal of this thesis - Multi-tenancy support based on TLS/SNI. The aim of the solution is to lower the server memory footprint by sharing a single data container instance by many users (tenants). The first section of the paragraph provides an introduction to the topic and is followed by the solution proposal. The next section provides experiments results followed by the results interpretation. The last two sections introduce the limitations of the solution and provide a discussion on further work.

Chapter 5 presents the second proposal - exposing a clustered applications deployed in the cloud. The main goal of the solution is to use a Load Balancer per every application instance and client-side Load Balancing technique to increase the overall system throughput. The chapter uses the same structure as the previous one - an introduction is followed by the solution proposal, experiment results, results interpretation, limitations and a discussion on further work.

Chapter 6 introduces the third proposal - using different techniques to switch to a custom binary protocol for the client/server communication. The main goal is to switch to the fastest possible protocol supported by both a client and a server. The structure of this chapter is similar to the previous ones.

Chapter 7 presents the last proposal - leveraging Operator Framework along with Machine Learning to detect incorrect application configuration. This solution enables domain experts (like product Support Staff) to detect application misconfigurations on client's site automatically and suggest how to fix them in an automated fashion. The structure of this chapter is similar to the previous ones; however, a discussion on further work introduces a fully-automated system for optimizing application configuration. The solutions described in this chapter have been used to file two United States Patents.

Chapter 8 concludes the thesis and provides a discussion on all the presented solutions.

Chapter 3

Related work

3.1 Dynamic load balancing

With the increasing popularity of the Virtual Machines, some of the researchers and engineers focused on exposing multiple domains in a single server. There are 4 different approaches to this problem [119]:

- Client-based approach - The client decides what IP address should be used for sending requests.
- DNS-based approach - A DNS server is responsible for exchanging the domain name with proper IP address.
- Server-based approach - One server delegates requests to the others. A data grid application is a common example of such a system.
- Dispatcher-based approach - A dedicated component (typically a Load Balancer) is responsible for redirecting the traffic to the proper server. Typically, this happens on URL level.

One of the biggest milestones in this area was publishing RFC7230 [92] and releasing Apache 2.2 [72] with a “Name-based Virtual Host Support“. The RFC defines, how the ”Host“ HTTP header shall be used for dispatcher-based routing when hosting multiple Virtual Servers from a single IP address. The biggest downside of this approach is that HTTP headers are part of the payload and are encrypted when using TLS (see Section 4).

3.2 Accessing clustered applications deployed

The methods for exposing an application cluster deployed in the cloud to the outside world have been investigated since the early days of cloud computing.

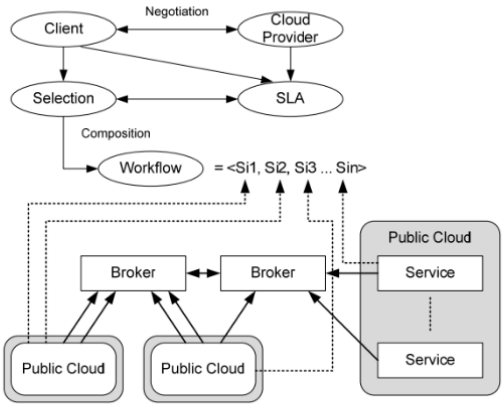


Figure 3.1: Cluster as a Service overview [1]

The “Cluster as a Service” [1] approach based on the RVWS (Resources Via Web Service) framework published in 2010 proposes an abstraction to expose a set of WSDL-based web services deployed on multiple cloud vendors. One of the most important characteristics of the framework is the ability to discover new clusters automatically. This is made possible by using a broker architecture which connects all the exposed clusters. A “client” application along with a “cloud provider” specifies a required SLA (Service Level Agreement) for the client’s request or work flow (Figure 3.1). An SLA is a type of contract between a cloud provider and a client that describes the performance and functional characteristics of a service.

The RVWS framework uses a Publisher Service to expose cluster characteristics in WSDL format. Based on this information, the Broker can decide which cluster best meets client requirements (Figure 3.2).

Even though the “CaaS” framework aims at different goals from those of this thesis, it relevantly solves both the discovery and connectivity problems within the cloud. The RVWS framework, which is a fundamental part of the system, focuses on batch processing tasks and WSDL-based web services whereas solutions presented in this thesis focus on high performance and low latency communication required by modern caching solutions and the gaming industry. To address this, the solutions presented in this thesis utilize basic parts of a “Publisher Service” and offer direct communication with the cluster (similarly to “CaaS”). Since both solutions allow the client application to communicate directly with the cluster, the security concerns remain similar. However, the WSDL-based solutions are often based on the HTTP protocol rather than TCP connection, with custom binary protocols (network protocols which use binary arrays to encode commands and data, often used in caching solutions). Since both edge routers and reverse proxies

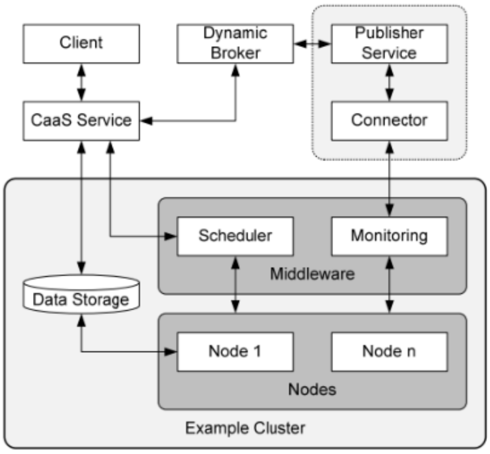


Figure 3.2: Exposing a cluster using Publisher Service [1]

offer many more configuration options for HTTP, the “CaaS” solution might be considered to be the more flexible solution in terms of application security.

Exposing a cluster within the cloud is also similar (to some extent) to multi-tenancy [97]. The distinction is that multi-tenancy is based on tenant recognition, whereas exposing a cluster requires recognizing a specific server (to which a specific request should be redirected). For HTTP-based protocols, there are a number of options including cookie-based routing, or using a Host HTTP header. However, high performance protocols are often limited to binary data based on TCP or UDP transport. Even though the TCP protocol itself does not have any support for these kind of solutions, encrypting traffic using Transport Layer Security (TLS) with Server Name Indication (SNI) allows the use of the HostName field [34], which contains a fully qualified server name. This approach has been successfully used in the Section 4 paragraph.

An interesting approach (analyzed in depth in Section 5) is to use a Load Balancer for each server deployed in the cloud, since most of the cloud vendors use high performance solutions for load balancing. A similar type of functionality has been provided by Fabric8’s Expose Controller [6]. This controller allows exposure of a Load Balancer per group of server instances (not a Load Balancer per server) and it does not provide any information about internal/external address mapping. Still, from a technical perspective, the Expose Controller was used as a source of inspiration for the technical solution presented in Section 5.

Finally, all the application instances need to operate on some sort of hardware. The scheduler is typically responsible for assigning an application replica to a node that runs the program. Even though scheduling algorithms are not strictly connected to the proposals mentioned in this thesis, it is worth mentioning that it is an interesting and very important area of research for cloud-related applications [3, 4, 64, 70].

3.3 Network protocols for client/server communication and encryption

Securing network connections between clients and a server is very important in heterogeneous cloud environments. Both clients and the server might be located in different data centers (and communicating over the WAN). Also, application developers should never trust the network environment of the cloud providers (as they have no control nor deep knowledge about its implementation details). One of the most commonly used solutions for connection security is using the Transport Layer Security (TLS) for securing connections.

A popular choice of a network protocol used for client/server communication involves using HTTP/1.1 or HTTP/2. The Hypertext Transfer Protocol (HTTP) was designed in mid-nineties and had several subversions (0.9, 1.0, 1.1) since then. In 2009, Google started to experiment with a protocol, called SPDY, which was designed to solve the blocking limits of HTTP/1. Two years later, the Internet Engineering Task Force (IETF) created a new version of the HTTP protocol, called HTTP/2, which was based on SPDY. The main goal was to decrease a web page loading times [99].

HTTP/1.1 is very simple at its design. It is a text-based, request/response protocol. Over the years, there were several improvements introduced into the protocol to increase its performance, namely the keep-alive feature (a single TCP connection is being reused for many HTTP requests/responses) and pipelining (sending more than one requests from the client at the same time). Figure 3.3 presents how the pipelineing feature works in the client - server communication scenario. On the left side, a client sends synchronous requests one after another. On the right side, the client sends two requests (the number of concurrent requests depends on the client's configuration) at the same time and waits for both responses. However, with the increasing network payload size, HTTP/1.1 suffers from Head Of Line (HOL) problem, where the connection pool (from the piplining feature) gets exhausted by transmitting larger objects. With the typical web page size increasing, that becomes a problem.

HTTP/2 architecture is much more complicated and has the following characteristics:

- It is a binary protocol.
- It uses the HPACK algorithm to compress HTTP headers.

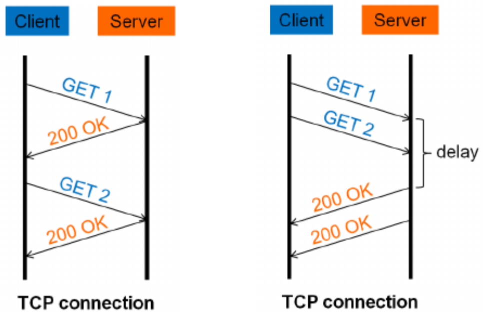


Figure 3.3: HTTP/1.1 request and response flow [99]

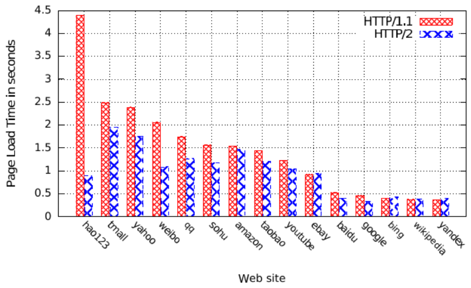


Figure 3.4: Performance comparison of HTTP/1.1 and HTTP/2 [99]

- It supports multiplexing (multiple data streams over the same TCP connection)
- It implements the Server Push feature (the server can upload certain resources to the client in advance).
- It uses TLS encryption by default.

As results show, HTTP/2 generally outperforms HTTP/1.1 without encryption (TLS) turned on [99] (Figure 3.4 and Figure 3.5). With an increasing payload size, the difference between HTTP/1.1 and HTTP/2 latency get larger. Since modern applications are designed as a single web page Javascript applications, this change is noticeable from the client's perspective (in the client's browser for example).

Some research groups focused on improving HTTP/2 performance report interesting progress (especially beyond 90th percentile, it is also referred to as “tail performance“) [51]. The authors used the RT-H2 Model to measure the total time needed to load a web page by a real user (the parameter is called PLT - Page Load Time). An experiment done on nearly 280 000 downloads

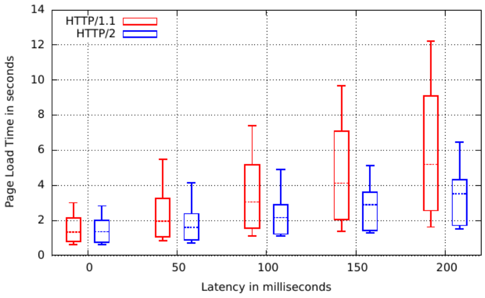


Figure 3.5: Latency comparison of HTTP/1.1 and HTTP/2 [99]

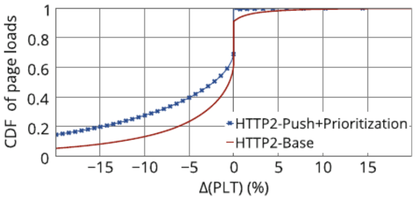


Figure 3.6: Performance improvement with HTTP/2 Push and Prioritization [51]

from Akamai's CDN showed that basic HTTP/2 features allow to improve the 90th percentile of PLT for nearly two thirds of the websites. Using HTTP/2 priorities and push feature allowed to extend this to other websites as well. Figure 3.6 demonstrates performance improvement by using HTTP/2 Push as well as Prioritization. The x-axis represents Page Load Time (PLT), whereas the y-axis represents Cumulative Distribution Function.

When HTTP gets secured by TLS, it is often referred to as HTTPS. Originally, this use case was designed for client/server applications that required confidentially or authentication between peers (such as e-commerce, banking or email). Today, most of the internet web pages are secured with TLS [41].

Each HTTPS connection starts with a TLS handshake sub-protocol shown in the Figure 3.7. Depending on the handshake settings, the server and the client need to perform 3 requests (often called RTTs - Round Trip Times). During the handshake, the Public Key Infrastructure (PKI) suite is being used to authenticate the server (and sometimes the client as well). Once the handshake is finished, both the server and the client are ready to transmit HTTP requests and responses encrypted and decrypted by the exchanged keys.

The Figure 3.8 presents TLS handshake costs. Scatter plot of the TLS handshake duration with respect to server distance(left). TLS handshake duration Cumulative Distribution Function (center). Ratio of TLS handshake bytes with

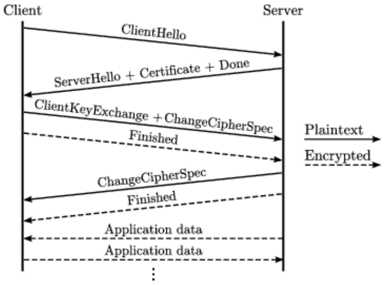


Figure 3.7: TLS handshake sub-protocol [60]

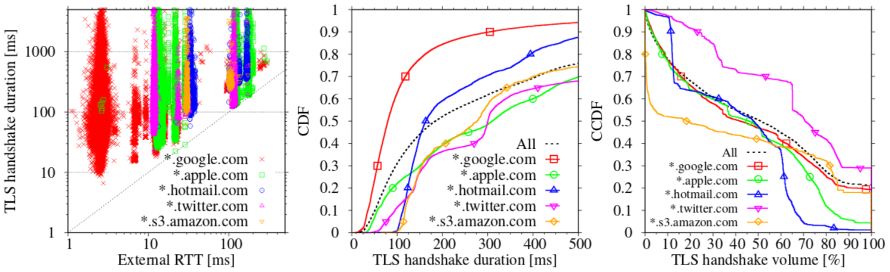


Figure 3.8: TLS in HTTPS performance results [22]

respect to total TCP connection bytes Complementary Cumulative Distribution Function (right). It is clear that extra latency introduced by TLS is not negligible comparing to the time needed to send request and response [22].

3.4 Machine Learning overview

Increasing computing power and storage capacity influences the rapid growth of Machine Learning industry. New algorithms allow to cluster calculations into multiple stages and together with cloud computing enabled processing large datasets. Modern systems are capable of giving users recommendation about the next movie to watch, what book to buy next and help protecting their email boxes from unwanted messages (spam filters). Machine Learning has also been used for scientific tasks, such as tracking animal migration patterns from satellite pictures or helping medical staff in diagnosing diseases[50].

Machine learning might be understood as an automated equivalent of statistical learning. Both approaches provide a set of methods and tools for understanding the data - a process of extracting meaningful information from a larger dataset.

In a general sense, the Machine Learning techniques might be divided into:

instance no.	Feature no. 1	Feature no. 2	Class
1	5	0.11	Good
2	6	5.98	Bad
3	2	4.11	Bad

Table 3.1: A dataset example

- Supervised Learning
- Unsupervised Learning
- Reinforced Learning

In many cases, the Machine Learning model is trained using a table-like data structure. Rows in the dataset are called ”instances“, columns - ”features“ and the output used for classification - a ”class“ or a ”label“. An example of such a table is presented in Table 3.1.

Supervised Learning requires building a statistical model and using it to predict the output (a class, binary value or a number) based on one or many inputs (features). Using Table 3.1 as an example, the model might qualify an instance to a specific class based on its features. During the Supervised Learning process, the model effectiveness might be accessed using different types of measurements. This approach differs from Unsupervised Learning, where there is no way of accessing an output of the model. The Unsupervised Learning technique assumes there is no labeled data in the learning dataset. Usually, this technique is used for pre-processing the data and dividing a large dataset into smaller groups [33][24]. Another kind of Machine Learning is Reinforcement Learning, which uses reinforcement signal (a scalar value) that constitutes a measure of how well the system operates [100].

Machine Learning algorithms and methods can be divided into groups for solving the following problems [82]:

- Predicting values (or regression)
- Item classification
- Anomaly detection
- Discovering structure

Predicting values based on learning dataset is often referred to as a regression problem. One of the most fundamental techniques to address a regression problem is the Linear Regression or Parametric Regression in general. The Linear

Regression technique predicts output y based on input x using Equation 3.1 [67]. The more general form that takes multiple features into consideration might be written as Equation 3.2. The subscript i is an observation index, whereas p designates independent variables [44].

$$y = ax + b \tag{3.1}$$

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \epsilon_i \tag{3.2}$$

Linear and parametric regression coefficients need to be found during the model training process. This process involves minimizing the mean of squared error (Equation 3.3) between measured and predicted values. There are two most frequently used methods to do this - Gradient descent or the Least Squares algorithms.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y_i})^2 \tag{3.3}$$

The final step of the training process is to access the accuracy of the model. The most commonly used technique is R^2, but there are others, such as Residual Standard Error (RSE) or F-Statistic (omitted in this thesis as it applies to multiple linear regression model).

RSE is computed using Equation 3.5 and Equation 3.4 formula. A useful way to think about RSE is that if a and b had been known, any prediction of the output value from the model would be off by RSE value on average. The question if this is acceptable or not depends on the context.

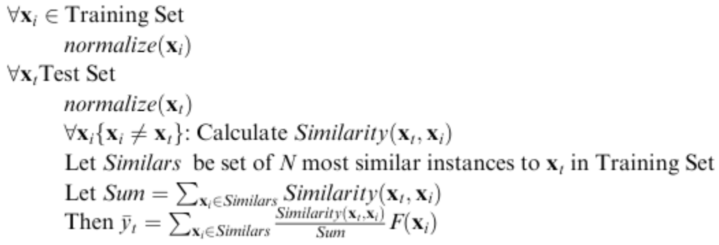
$$RSS = \sum_{i=1}^{n}(y_i - \tilde{y_i})^2 = \sum_{i=1}^{n}(y_i - (ax_i + b))^2 \tag{3.4}$$

$$RSE = \sqrt{\frac{1}{n-2}RSS} \tag{3.5}$$

The R^2 statistic is much more frequently used as its values are within $\{0..1\}$ range and can be calculated using Equation 3.6. Formally, TSS parameter can be expressed as $TSS = \sum_{i=1}^{n}(y_i - \bar{y_i})^2$, however its general sense is RSS for a model that always returns fixed values.

$$R^2 = 1 - \frac{RSS}{RSS\ for\ mean} = 1 - \frac{RSS}{TSS} \tag{3.6}$$

A Regression task might also be solved using Instance-Based Learning. One of the popular algorithms of this class is called Nearest Neighbor (or Proximity).



```
$\forall \mathbf{x}_i \in$ Training Set
    $normalize(\mathbf{x}_i)$
$\forall \mathbf{x}_t$Test Set
    $normalize(\mathbf{x}_t)$
    $\forall \mathbf{x}_i\{\mathbf{x}_i \neq \mathbf{x}_t\}$: Calculate $Similarity(\mathbf{x}_t, \mathbf{x}_i)$
    Let $Similars$ be set of $N$ most similar instances to $\mathbf{x}_t$ in Training Set
    Let $Sum = \sum_{\mathbf{x}_i \in Similars} Similarity(\mathbf{x}_t, \mathbf{x}_i)$
    Then $\bar{y}_t = \sum_{\mathbf{x}_i \in Similars} \frac{Similarity(\mathbf{x}_t, \mathbf{x}_i)}{Sum} F(\mathbf{x}_i)$
```

Figure 3.9: Nearest Neighbor algorithm pseudo-code [44]

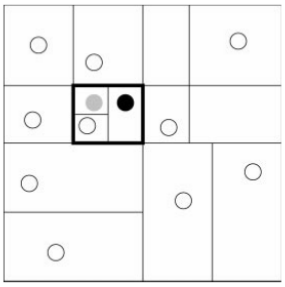


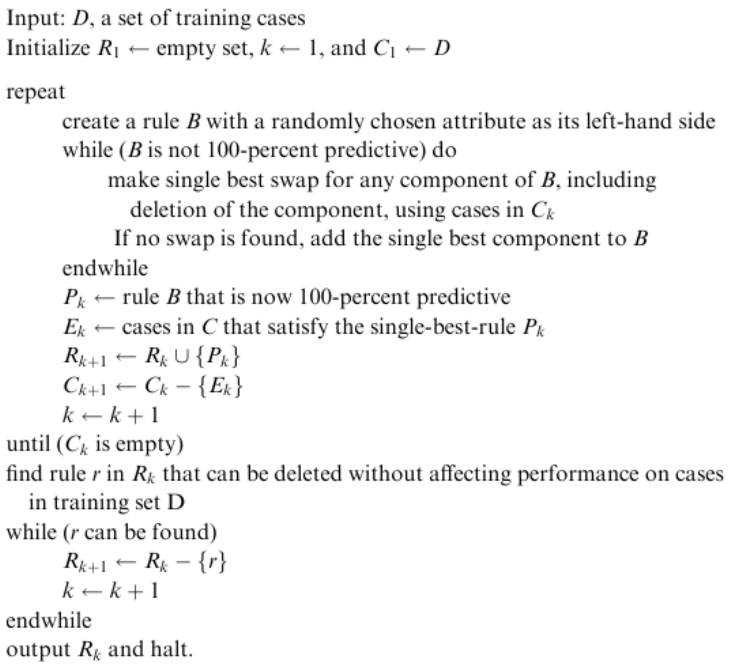
Figure 3.10: DNF representation [44]

The algorithm stores the entire dataset and predicts the output based on the Euclidean distance between a given set of features and the stored ones. Figure 3.9 presents the algorithm's pseudo-code, which relies on normalized data (so that all features values are within a continuous range between 0 and 1) and assumes all instances are equality meaningful.

Another similar class of algorithms is called Locally Weighted Regression, where features with certain values also have weights assigned to them. Weights can be distributed evenly (this class of algorithms is called Linear Local Models) or non-evenly (Nonlinear Models).

One of the interesting approaches to solving regression problems is to use a rule induction mechanism. Certain algorithms use Disjunctive Normal Form (DNF) for the process of learning decisions. One of the main advantages of this approach is its explanatory capability. Figure 3.10 shows a sample diagram, where the black dot is a query point. The shaded one is the nearest neighbor and the black box inside of the diagram is the query region of searching for nearest neighbor.

One of the challenges of the DNF-based algorithms is that the rules are not mutually exclusive. This may lead to a problem when for a given input, more than one output may be satisfied. One of the most common solutions is to apply priority or ordering of rules. Figure 3.11 represents a pseudo-code implementation for a



```
Input: $D$, a set of training cases
Initialize $R_1 \leftarrow$ empty set, $k \leftarrow 1$, and $C_1 \leftarrow D$

repeat
    create a rule $B$ with a randomly chosen attribute as its left-hand side
    while ($B$ is not 100-percent predictive) do
        make single best swap for any component of $B$, including
          deletion of the component, using cases in $C_k$
        If no swap is found, add the single best component to $B$
    endwhile
    $P_k \leftarrow$ rule $B$ that is now 100-percent predictive
    $E_k \leftarrow$ cases in $C$ that satisfy the single-best-rule $P_k$
    $R_{k+1} \leftarrow R_k \cup \{P_k\}$
    $C_{k+1} \leftarrow C_k - \{E_k\}$
    $k \leftarrow k + 1$
until ($C_k$ is empty)
find rule $r$ in $R_k$ that can be deleted without affecting performance on cases
  in training set D
while ($r$ can be found)
    $R_{k+1} \leftarrow R_k - \{r\}$
    $k \leftarrow k + 1$
endwhile
output $R_k$ and halt.
```

Figure 3.11: Swap-1 pseudo-code [44]

Swap-1 - a rule induction algorithm. The most important part of the algorithm is to build a rule from a set of components that will be evaluated to produce the output. The Swap-1 algorithm searches all the conjunctive components it has already formed, and swaps them with all the possible components it will build. This search also includes the deletion of some components from the rule. If no improvement is established from these swaps and deletions, then the best component is added to the rule. To find the best component to be added, the predictive value of a component, as a percentage of correct decisions, is evaluated. Once the rule is selected, the algorithm performs regression based on query point [44].

The final class of algorithms analyzed for this thesis are the tree induction algorithms, also known as regression trees. Tree induction algorithms construct the model by partitioning the dataset. Each node in a tree represents a splitting decision as shown in Figure 3.12. Regression trees are similar to the DNF-based algorithms and their decision boundaries (this term is used for classification algorithms but in case of regression trees, it also applies [67]) might be presented in a graph, similar to Figure 3.10. However, the thing that makes decision trees different from DNF-based algorithms is that the decision areas are mutually exclusive. Both the regression and decision trees have been described in detail in Section 7.1.1.

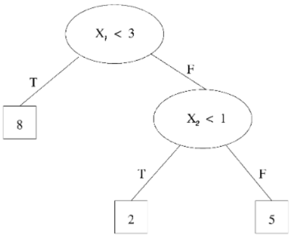


Figure 3.12: Regression tree [44]

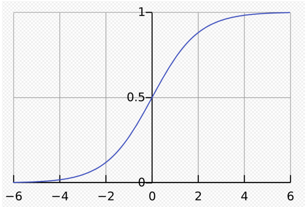


Figure 3.13: Logistic curve

In some cases, the model output is not quantitative (in other words, it is not a number) but rather is qualitative (it is a category). In this case, the output variables can be called ”categorical“. The process of predicting categorical data is called ”classification“ (or ”item classification“). The model that predicts a category based on an input is called a ”classifier“.

One of the simplest approaches to classification is to use a regression model with applied Sigmoid function function at the end. Figure 3.13 presents a diagram of a Sigmoid function (Equation 3.7).

$$S(x) = \frac{1}{1 + e^{-x}} \tag{3.7}$$

Another group of algorithms is similar to the DNF-based regression algorithms and is often referred to as rule-based classifiers. A commonly used example is k-DNF, where k is a number of disjunctions. The goal is to construct the smallest rule-set that is consistent with the training data [100].

The same symmetry applies to Regression and Decision Trees. Fundamentally, both approaches are very similar. The only difference is with the final outcome - Decision Trees produce a class in the output (as presented in Figure 3.14). Section 7.1.1 contains more detailed review on Tree-based Classifiers.

Another well-known group of algorithms is based on notion of perceptron. A single layered perceptron can be briefly summarized by the definition: If x_1 through x_n are input feature values and w_1 through w_n are connection

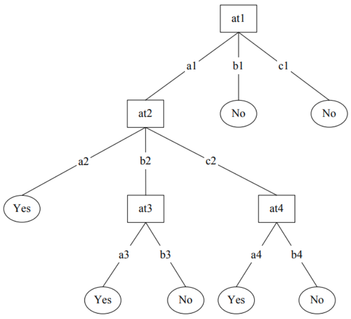


Figure 3.14: Decision Tree example [100]

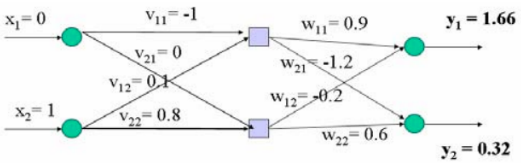


Figure 3.15: Feed Forward ANN [100]

weights/prediction vector (typically real numbers in the interval $[-1, 1]$), then perceptron computes the sum of weighted inputs: $\sum_i x_i w_i$ and output goes through an adjustable threshold: if the sum is above the threshold, the output is 1; otherwise it is 0. A popular example of such an algorithm is called WINNOW and its advantage is the fact that it can be trained in batches.

A single-layer perceptron-based algorithm is capable of classifying linearly separable set of instances. For more complicated use cases, a multi-layered perceptron-based approach needs to be used. Such an approach is often referred to as Artificial Neural Networks (ANN). Such networks typically consist of large number of units called neurons, joined together in patterns of connections as shown in Figure 3.15.

During the classification process the signal at the input propagates all the way through the net passing through certain neurons with activation threshold set. Units in the network are typically qualified as inputs, outputs and ”hidden units“, which are between an output and an input.

With the increasing computing power and large learning datasets (also called ”Big Data“), it is possible to create large Neural Networks. Such large networks have no theoretical limitations to what they can learn. The more data is fed into the network, the better it gets. Such techniques are also known as Deep Learning and are one of the most innovative fields in Artificial Intelligence [49].

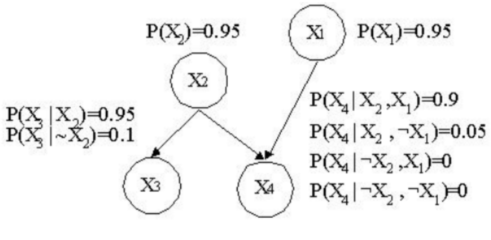


Figure 3.16: Feed Forward ANN [100]

A slightly different approach to classification is used by Bayesian Networks, which is based on estimating value from Equation 3.8. If $R > 1$, the result is i, j otherwise. Bayesian Network is a graphical model for probability relationships among a set of features. Figure 3.16 shows a simple network for classifying X based on 4 features.

$$R = \frac{P(i|X)}{P(j|X)} = \frac{P(i)P(X|i)}{P(j)P(X|j)} = \frac{P(i)\prod P(X_r|i)}{P(j)\prod P(X_r|j)} \tag{3.8}$$

Another interesting algorithm commonly used for classification is KNN - K-Nearest Neighbors. Conceptually, it is similar to the Nearest Neighbor algorithm but it calculates the proximity to K neighbors when qualifying instance to a given class.

The last class of analyzed classification algorithms is the Support Vector Machines (SVM). This class of algorithms uses the notion of ”margin“ - either side of a hyperplane that separates two data classes. Maximizing the margin and thereby creating the largest possible distance between the separating hyperplane and the instances on either side of it has been proven to reduce an upper bound on the expected generalisation error [100]. If training data is linearly separable, then a pair of (w, b) described by Equation 3.9.

$$\begin{cases} w^T x_i + b \geq 1, for\ all\ x_i \in P \\ w^T x_i + b \leq -1, for\ all\ x_i \in N \end{cases} \tag{3.9}$$

The decision rule has been presented in Equation 3.10, where w is termed the weight vector and b the bias.

$$f_{w,b}(x) = sgn(w^T x + b) \tag{3.10}$$

The solutions to both regression and classification might be used for anomaly detection. Anomaly detection is heavily used, e.g. in Finance, IoT (Internet of Things), Auditing or Healthcare. In most of the cases, Machine Learning techniques allow to monitor applications (or systems) and prevent incidents by proactive maintenance. Anomaly detection techniques are often used when all the three preconditions are met:

- Training data is labeled
- Anomalous and normal classes are balanced (in other words, the training dataset contains anomalies but this is not the majority of cases)
- Data is not autocorrelated (in such a case, time series analysis is more appropriate).

Gathering proper training set for anomaly detection is very complicated. Very often, there is not enough data or anomalous and normal classes are not balanced (e.g. there is only 1 case of anomaly per 10 000 normal cases). All common Machine Learning classifiers will often miss anomalies during the training (although there are some techniques that allow to solve this problem, for example - resampling anomaly into the dataset). In a standard Machine Learning process, anomalies are often treated as outliers and are removed from the training dataset.

Anomalies might be divided into types:

1. Point anomalies - if the value represented by data lies outside the overall pattern of distribution [120]
2. Contextual anomalies - if the value is an anomaly in a specific context
3. Collective anomalies - if a collection of data values are anomaly with the respect to the entire dataset

There are many different approaches to detecting anomalies in a dataset. The easiest (and probably the most common) approach is using the static rules. A static rule set might be written by a domain expert and might contain all the known anomalies. Assuming such anomalies do not happen often, this practical approach might fit many use cases.

In some cases, there is no training dataset and Unsupervised Machine Learning techniques need to be used. The easiest way to detect numerical anomalies is to use percentiles and histograms. A common example is when a data point lies beyond the 90th percentile of a all data values, it can be treated as an outlier or anomaly.

Choosing a proper anomaly detection approach if often a difficult task. Figure 3.17 presents a graph with explained algorithms and presents an intuitive way of picking proper solution [82].

The last group of algorithms are the algorithms for discovering some sort of structure (also called Pattern Recognition) in the data. Most of the algorithms use

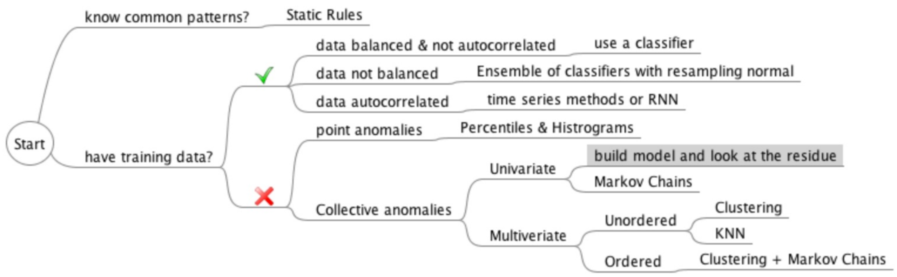


Figure 3.17: Anomaly detection technique algorithm

Unsupervised Machine Learning technique and are often used as a pre-processing step for building more advanced models.

A commonly used technique for pattern recognition is called the Principal component analysis (PCA). PCA finds the principal components (or eigenvectors) of a data distribution spanning a linear subspace of the feature space (often called eigenspace) [111]. The algorithm transforms data into a new coordinate system, so that the greatest variance becomes the first coordinate (called the principal component), the second greatest variance becomes the second coordinate and so on [84]. Combining empirical variance described in Equation 3.11 and projection coordinate described in Equation 3.12 the principal component might be calculated as shown in Equation 3.13 and further components as shown in Equation 3.14 [5].

$$Var_{emp}(Z) = \frac{1}{n-1}\sum_{i=1}^{n} z_i^2,\ where\ z\ are\ observations \tag{3.11}$$

$$z = x^T v,\ where\ x\ and\ v\ are\ column\ vectors \tag{3.12}$$

$$w_{(1)} = max_w \sum_{i=1}^{n} (x_i^T w)^2,\ where\ w^T w = 1 \tag{3.13}$$

$$w_{(k)} = X - \sum_{s=1}^{k-1} (X w_s w_s^T) \tag{3.14}$$

Independent Component Analysis (ICA) is another example of an algorithm capable of separating a stream of data into meaningful parts. The algorithm was invented for separating individual microphone signals from a combined signal. It assumes the mixed signal is the sum of its components and is unable to identify Gaussian components because their sum is also normally distributed [5].

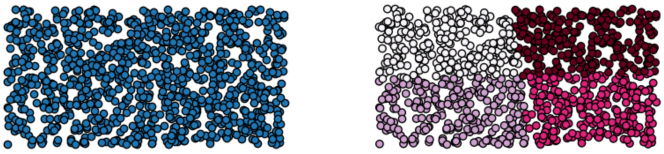


Figure 3.18: k-means example [5]

There is also a large group of algorithms that group individual data points into larger groups. Typically, such algorithms are called the Clustering Algorithms and the most commonly used examples are:

- K-means - which qualifies data points into larger clusters based on distance
- Gaussian mixture models - where data points are generated using multi-variate normal distribution
- Density-based clusters - clusters are formed by the existence of a minimum number of nearby data points

An interesting approach to clustering the data is used in the k-means algorithm. The algorithms uses cluster centers (centroids) into random positions and then minimizes Euclidean distance between each data point and its centroid [5] (formally, the objective function was written in Equation 3.15). Figure 3.18 shows an example run of the algorithm on randomly generated data points.

$$min_{C_1,\ldots,C_k,\mu_1,\ldots,k} \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2 \tag{3.15}$$

Introducing Machine Learning into a software development process is not an easy task. Getting the best results requires revisiting certain stages multiple times. Researchers from Microsoft present a typical Machine Learning workflow in Figure 3.19. The workflow consists of 9 stages. Some stages are data oriented (data collection, data cleaning, data labeling) and some are process oriented (feature engineering, model training, model evaluation, model deployment, model monitoring). Nevertheless, a Machine Learning model implemented in a typical software product requires continuous maintenance and tuning [95].

3.5 Expert Systems overview

A typical Expert System is software that is capable of analyzing given problem and proposing a solution to it. The decision-making process involves matching

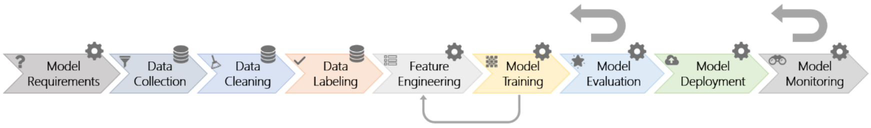


Figure 3.19: Machine learning workflow

evaluations of multiple fashion attributes and is commonly referred to as Multiple Criteria Decision Making (MCDM) [123]. An expert system simulates the performance of an expert and its knowledge base contains problem-related expertise.

The most important part of designing an expert system is to successfully extract knowledge from an expert, perform its analysis and model it properly in the system [52]. All phases are crucial for proper system functioning and lack of any step may result in:

- Solutions being picked up without deep problem analysis
- Wrong assumptions of operating suggested solutions as scale

Most of the problems mentioned earlier might be summarized in two fundamental questions:

- How to effectively extract knowledge from experts
- How to design systems to deal with problems with uncertain, value and imprecise information.

Knowledge Acquisition and Documentation Structuring (KADS) methodology addresses the former problem by viewing Knowledge Base Systems (KBS) as a modeling activity [15][53]. Author's key idea is that experts' knowledge has computational interpretation - it is executable. The Knowledge Base System is not just a container for knowledge but is rather an operational model that exhibits some desired behavior that can be observed in the system.

Dealing with uncertain values and imprecise information in the Knowledge Base System has been identified as the second important challenge of expert system design. Using statistics and Machine Learning are (among others) the techniques to deal with it (more information might be found in Section 3.6, Section 3.4 and Section 7.1.1).

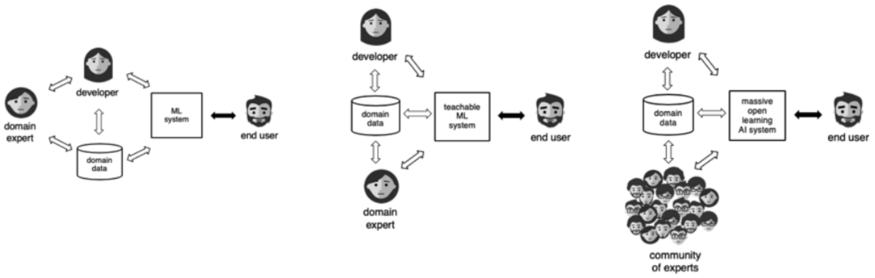


Figure 3.20: Three ways to develop a Machine Learning system [83]

3.6 Machine Learning in Expert Systems

In typical scenarios, in a software development company, a Machine Learning model is implemented by a ”developer“ with the help of a ”domain expert“. Knowledge extracted from domain experts is typically stored in some sort of data structure in a ”Domain Data“ repository (e.g. in the form of a spreadsheet). Once the system is created, its main goal is to communicate with an ”end user“. The system development process might happen in certain models [83] (shown in Figure 3.20):

- Mediating domain experts through developers
- Domain experts accessing the system directly (such a system is called a ”teachable system”)
- A community of experts accessing the system directly (such a model is called “massive open learning AI system”

When creating a Massive Open Learning AI System (MOLA), the researchers discovered that preparing data for Machine Learning model often requires labeling it. If this task is done by people, the amount of mistakes gets quite high. High level domain experts also fall into this category. The authors found this paradox as an interesting thing of human nature - “in order to teach machines, experts have to behave like one of them“. However, this domain knowledge transfer is required to train the Machine Learning model. One of the most interesting findings of the research was that introducing a larger group of experts (a community of experts) resulted in better behavior or a Machine Learning models.

Another interesting example of combining the Machine Learning methods along with domain experts knowledge is diagnosing sleep disorders using the Overnight polysomnography (PSG) method. This non-invasive method collects various physiological data which is then scored by sleep specialists who assign a sleep stage to every 30-second window of the data according to predefined scoring rules. A number of research studies showed that inventing an automated classifier for PSG does not align visual sleep specialist predictions, or the exact decision procedure is uninterpretable. The researchers solved this problem by introducing the Automated Sleep Stage Scoring Algorithm (AASM), which used both the rules engine and a Decision Tree Classifier. The proposed automated sleep scoring system consisted of five main steps. Prior to analysis, the data were pre-processed in accordance with AASM criteria for detecting REM phase settings. Then, the features based on the AASM scoring rules were extracted from the PSG signals. The third step entailed choosing an optimal threshold for each feature. A likelihood ratio Decision Tree Classifier was then utilized to perform the classification, and finally a set of temporal contextual smoothing rules was applied on the annotated data [54].

The research shows the potential of leveraging both highly qualified domain experts along with the Machine Learning methods. The output of this approach produced better scoring accuracy and better interpretability by visual sleep specialists.

3.7 Automated cluster maintenance system

Some researchers used Machine Learning techniques for automatic database (DBMS) configuration tuning [23]. The researchers mentioned two major difficulties found when implementing their prototype:

1. There are many configuration parameters available for tuning
2. The impact of many configuration parameters comes only from (expensive) experience

The authors used a combination of Supervised and Unsupervised Machine Learning techniques to select the most impactful configuration parameters, map unseen workloads to previous workloads from which the model can learn and recommend configuration parameter settings to the user. This approach is much more generic than other solutions implemented by database vendors (which leverage inside-knowledge to get the best tuning results). The prototype created for this research has been named OtterTune and uses high level architecture shown in Figure 3.21. The controller is responsible for collecting information about performance. This information is then sent to the manager and stored in the repository. Later on, this information is used to build a Machine Learning model and predict (suggest to the user) configuration settings.

Once the data is stored in the repository, a Machine Learning pipeline is being triggered (shown in Figure 3.21). Archived observations (from the

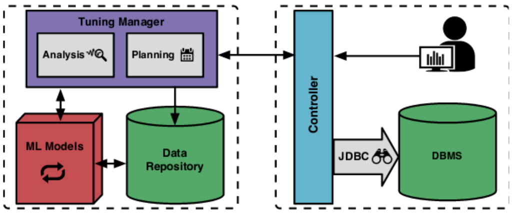


Figure 3.21: OtterTune architecture

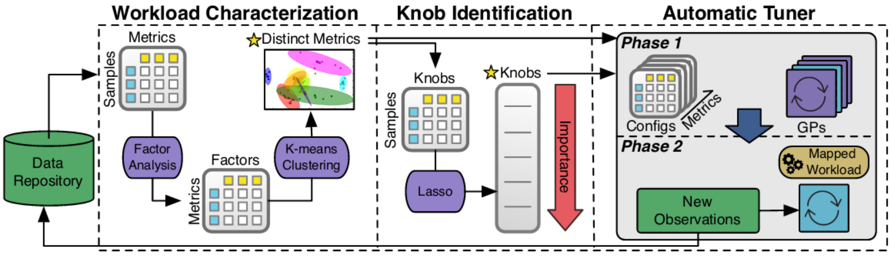


Figure 3.22: OtterTune Machine Learning pipeline

repository) are being passed to Workload Characterization section, where the most distinguishable metrics are being chosen. During the next step, in Knob Identification section, the most impactful configuration parameters are being identified. Finally, Automatic Tuner use metrics and configuration parameters (knobs) to come up with a configuration proposal. Once the configuration is put into life, the outcome metrics are being fed back to the system.

The authors used several Machine Learning techniques to achieve their goal. The first step is to reduce the number of metrics obtained by the system. In order to do that, a dimensionality reduction technique called Factor Analysis [30], in combination with k-means algorithm has been used. This allows to group metrics into meaningful groups. The next step is to identify configuration parameters most correlated with overall system performance, with a linear regression method called Lasso. Finally, Automatic Tuner uses the Gaussian Process regression to recommend configuration that it believes will improve the target metric (performance).

The results of this work are very promising. OtterTune generates configurations in under 60 min that are comparable to the ones created by human experts (in 94% of the cases).

3.8 Statistical significance for benchmark results

In order to check whether benchmark results differed from each other in any meaningful way, a test of significance needs to be performed for each pair of results. The test is based on a null hypothesis which assumes that both tested mean values are equal (see the equation below). The research hypothesis is that the means are different [62].

$$\mu_0 = \mu_1 \tag{3.16}$$

Since the samples are independent from each other, and the measured groups are not linked to each other, the significance test can be done using the "Two-Sample t-Test for Equal Means":

$$Z = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \tag{3.17}$$

where $\overline{X_1}$ and $\overline{X_2}$ are sample means, σ_1^2 and σ_2^2 are variances and n_1 and n_2 are sample sizes.

The final step is to prove that the calculated Z value is between $(-\infty, 1 - \frac{\alpha}{2} > \cup < 1 - \frac{\alpha}{2}, \infty)$ for the tested hypothesis that $\mu_0 \neq \mu_1$. For $\alpha = 0.05$ (99%), the tested interval is equal to $(-\infty, -2.58 > \cup < 2.58, \infty)$.

3.9 Related work discussion

Accessing services deployed in the cloud from the outside is very important, especially in interconnected cloud environments. Typical applications following microservices architecture often require HTTP-oriented communication; however, highly scalable, in-memory storage systems offer more advanced features when using custom binary protocols. Some intelligent client applications use the client-side load balancing techniques to optimize they way the data is accessed (a common example are consistent hash aware clients - a technique often used by in-memory data stores).

Section 3.1 presents different options for balancing the traffic in a dynamic fashion. The research mentioned in this thesis focuses on the Client-based approach, as it enables performance optimizations for accessing (and modifying) the data in an in-memory data store application hosted in the cloud.

Services deployed in the cloud often require configuring an additional component for ingress traffic. In most cases, such a component is either a HTTP Proxy or a Load Balancer. The former focuses on the HTTP protocol and enables broker approach as described in Section 3.2. The latter often allows exchanging binary data between a client and a server but may lack many configuration parameters, such as stickiness options. Data-related applications often transmit large portions of data between a client and a server. In many cases, this data needs to be sent in binary format. Literature analysis related to HTTP/1.1, HTTP/2 and TLS protocols (described in Section 3.3) proves that different protocols may help optimizing latency (or throughput) when communicating with an in-memory data store. Furthermore, the TLS protocol provides extension, such as SNI or ALPN, that might be helpful for routing traffic to a specific application instance or negotiating a communication protocol.

Maintaining high performance, clustered data storage system requires deep understanding of many configuration parameters. A very similar problem applies to very well known database systems, such as Postgresql. One of the proposals of this thesis introduces an automatic configuration corrections system using Machine Learning, which is an evolution of a similar concept described in Section 3.7. Based on practical experience and the related literature related to Expert Systems (described in Section 3.6) and Machine Learning (described in Section 3.4 and in Section 7.1.1) it is desirable to design an automated system capable of leveraging domain expert knowledge (oftentimes product Support Engineering Staff) to find common application configuration mistakes in an automated fashion. Such a system has been proposed in Chapter 7.

Chapter 4

Proposed solution for service name indication for multi-tenancy routing in cloud environments

4.1 Introduction

With the rapid growth of IaaS platforms, cloud vendors started experimenting with new type of offering - Platform as a Service. Platform as a Service approach allows to access a single application instance by many different clients. This allows vendors to better utilize resources and maintain a single application cluster instead of multiple ones. This setup performs very well for stateless applications, where a customer passes data into the service and gathers the results (for example a service for sending emails). However, operating data stores is much more complicated and requires separating data per clients (also known as tenant). The second concern is to maintain security context for accessing the data so that each tenant could access only their own partition.

Unfortunately, there is no generic solution for isolating the data between tenants and each service needs to come up with its own ideas. Database vendors often use database schema approach (schema per tenant), whereas data grid projects separate the whole clusters from each other. A lot of software deployed in the Cloud uses REST as the main communication interface. However, HTTP messages contain a lot of overhead and gaining maximum performance requires using transport protocols and sockets directly (UDP and TCP). A common way of securing a client/server connection is to use the SSL/TLS protocol. Additionally, a TLS extension called Service Name Indication (SNI) allows transmitting client

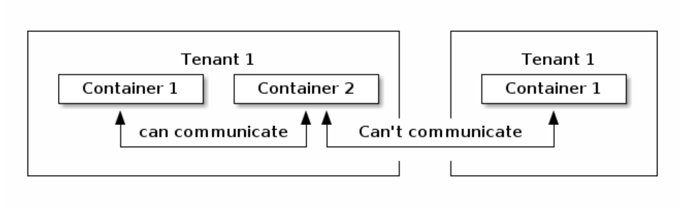


Figure 4.1: Multi-tenant communication

host name and choosing proper private key and certificate pair used for encryption on the server side. The same mechanism can be reused for choosing proper data partition simplifying data isolation aspect.

Running multiple applications in the cloud requires separating one of them from another. Oftentimes, such an application needs to be divided by tenants (Figure 4.1). Tenants can be coarse grained like companies or fine grained like projects or departments within a company. Applications within single tenant should operate within a single security domain, therefore, should be able to directly communicate with each other. Communicating to other tenants may need passing additional security checks (e.g. additional authentication or encryption).

The multi-tenant separation is often done in layer 2 and 3 of the OSI model and over the years the industry came up with different standards to achieve that [87]; however, the high level concept remains the same. Each VM (or container) is attached to a virtual NIC and packets are routed through a virtual switch. The implementation also needs to handle layer 3 and moving VM from one compute node to another. Some of the Cloud vendors mention what technology is used in their products, for example OpenShift by Red Hat uses Open vSwitches [79, 75] (Figure 4.2 and Google Cloud Platform developed their own project called Andromeda [38].

For some applications, tenant separation also needs to happen at the application level. The most common example is sharing a database installation between multiple users (or tenants) (Figure 4.3). Similar deployment model is often used on premise by larger organizations. A single database installation serves data for multiple projects.

The database industry came up with a concept of database schema [112] for supporting multi-tenancy. It allows to host multiple data containers in a single database instance. This approach requires a clear separation between data hosting layer and the transport in the application. This approach might not always be valid for high performance, since data intensive application of each tenant might have different transport requirements. A perfect solution should allow setting transport properties per each individual tenant.

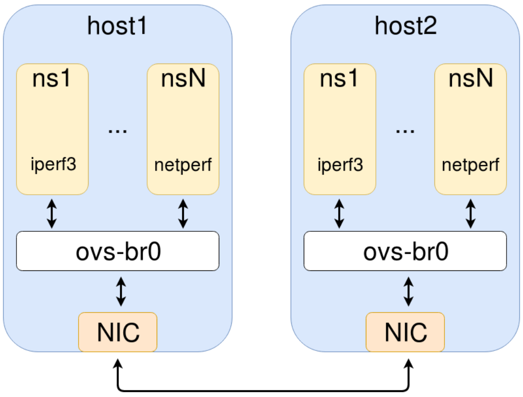


Figure 4.2: Open vSwitch architecture

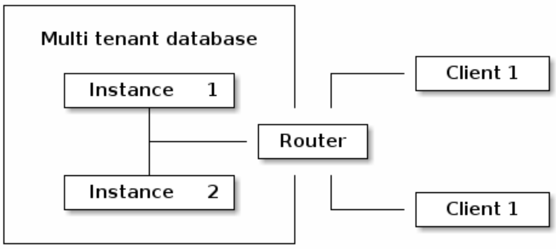


Figure 4.3: Multi-tenant application

Multi-tenancy aspect of an application becomes more important when considering a larger system. Sharing components between tenants allows to lower the operating costs by lowering resource consumption. At the same time, every large system exposing its functionality to multiple tenants needs to maintain a required degree of data isolation.

4.1.1 Transport Layer Security with Service Name Indication extension

The Transport Layer Security (TLS) is responsible for establishing (or resuming) a secure session between a server and a client. In order to decrypt the data, both the server and client need to follow a handshake procedure which involves the following:

1. Negotiating cipher suite
2. Authentication
3. Exchanging keys

When designing services are deployed in the cloud, the server needs to use different encryption keys for each client while its IP address remains the same. A similar problem occurs when hosting multiple virtual web servers on a single IP address and using the host header for multiplexing. For TCP or UDP based transports, the host header is encrypted and therefore the server cannot read it before finishing TLS handshake. In order to accomplish the handshake, the server needs to choose a proper certificate based on client's hostname (Service Name Indication) [68][116]. A simplified flow diagram might be found in Figure 4.4.

Typically, a Reverse Proxy in the front terminates TLS and sends unencrypted traffic to the service deployed in the cloud. However, in some scenarios, such a Reverse Proxy cannot be trusted. In this case, the hostname field from TLS/SNI is used to take routing decisions.

4.1.2 Data grid systems and memory consumption

The data grid market (and distributed caching in general) is very competitive. Different vendors try to propose the fastest solutions (with the highest throughput and lowest latency) and the smallest possible resource consumption. Storing data effectively is one of the most important deciding factors when considering deploying a large system in the cloud. Table 4.1 shows an annual cost per 1 GB of memory for a Virtual Machine. Considering a large amount of data and a large number of replicas, the total cost of hosting an in-memory data store system might turn out to be very high.

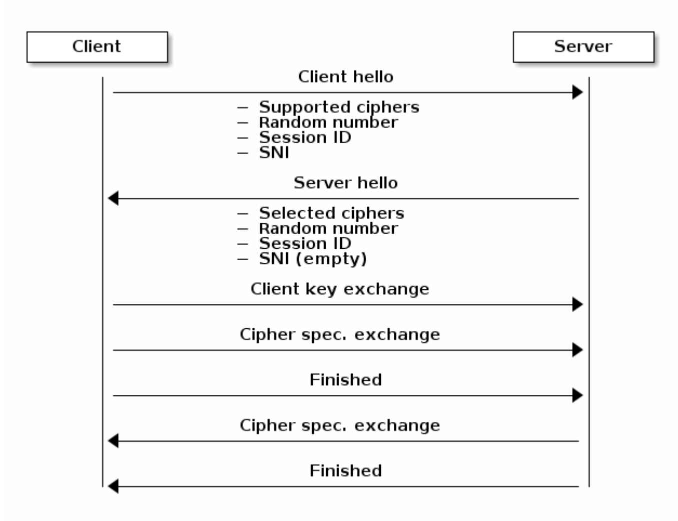


Figure 4.4: TLS handshake with SNI

Cloud Provider	Cost
AWS	78 USD per year per GB
Google Cloud	78 USD per year per GB
Microsoft Azure	64 USD per year per GB

Table 4.1: Memory pricing by the biggest cloud vendors [20]

Programming languages with managed runtime, such as Java, allow creating high performance systems but at the cost of high memory usage. Infinispan project (a data grid solution written in Java) reports over 400 MB of committed memory (more information might be found in Section 10.2) without any data loaded into it. Redis (a simple caching solution written in C++) on the other hand reports less than 10 MB. Even though data grid solution offer much better feature set, it is clear that 40 times lower memory footprint drastically lowers the system maintenance cost.

4.2 Proposed solution for identifying tenant using TLS/SNI Hostname field

The major contribution of this proposal is to design a solution that allows to decrease memory consumption per tenancy without compromising performance in a data grid system. This solution has been tested using a dedicated testing environment proving that the goal of decreasing memory consumption was achieved; however, the proposed solution introduced degradation of performance.

Achieving low memory consumption offered by lower-level programming languages (such as C/C++) is very hard when using Java (although there is ongoing work related to GraalVM and Quarkus projects to address this problem [42][86]). However, it is worth noticing, that in-memory data grid systems are often accessed by multiple clients (usually application servers in a cluster). Having this in mind, the memory consumption problem might be addressed in other way - by introducing multi-tenancy. This allows multiple client applications use their own segment of the data container in a secured fashion.

Authenticating and authorizing a client to access a data segment could be addressed at many levels. At the top level, there is an application, which may use a tenant-id and grant (or deny) access to a specific segment of the data. This requires a reliable mechanism for authentication and incorporating access rules to bind clients with the data segments. When considering a layer below - transport, many data store systems allow configuring transport-specific settings per each tenant individually. A common example could be two clients storing small amount of large objects, which requires a small amount of worker threads with large send and receive buffers and storing large amount of small objects, that requires large number of worker threads with small send and receive buffers. This also applies to security aspect - each client may require different authentication and encryption settings. In order to archive the biggest flexibility, it makes sense to provide transport security and authentication settings at the transport layer (and

Solution	Priority
Data container isolation	Must-have
Authentication	Must-have
Reverse Proxy support	Good-to-have
Small performance penalty	Good-to-have

Table 4.2: Requirements for proposed solution

not application layer). The ”hostname“ field from TLS/SNI allows the server to choose a proper private key for client/server communication and assuming the transport component can only be wired into single data segment, this ensures data separation between the tenants.

In a typical Cloud environment, hosted services are separated from the outside network and an additional routing component is needed to forward requests to inner servers from the clients (Figure 4.3). Most container-based clouds use a Reverse Proxy to handle ingress traffic to the cloud. However, Reverse Proxies are HTTP oriented and very limited when routing encrypted traffic. In such cases, the TLS/SNI extension allows to analyze ”hostname“ field (which is encrypted) and make routing decisions based on it. In a typical scenario, the ”hostname“ field contains a Fully Qualified Domain Name (FQDN) of an application.

Both using the TLS/SNI requirement for encrypted traffic in container-based clouds and providing the best flexibility when configuring encryption settings at transport layer in memory store projects are strong arguments for designing a TLS/SNI-based approach for multi-tenancy.

Apart from the thesis goals, a well designed solution for multi-tenancy support should fulfill secondary goals, that have been gathered in Table 4.2. The data container isolation as well as authentication are connected with each other. Data container access can only be granted upon authenticating a client application. Reverse Proxy support and small performance penalty are more practical requirements as they lower operational costs (by using the publicly available Reverse Proxy apart from expensive Load Balancers) and improve overall system performance.

The proposed multi-tenant router separates data containers (also known as Hotrod Servers) from TCP Servers (transport layer components) and starts a single frontend for all the inner services (Figure 4.5). This approach has smaller resource consumption (opposed to starting a frontend server per each Hotrod) and completely decouples forwarding requests from data container operations.

Each TCP Server uses Netty framework which is based on handlers architecture. All handlers are gathered together in a concept called Pipeline, where each handler is responsible for a piece of work (for example encrypting data using SSL or decoding data into user defined structures) [71]. This kind of

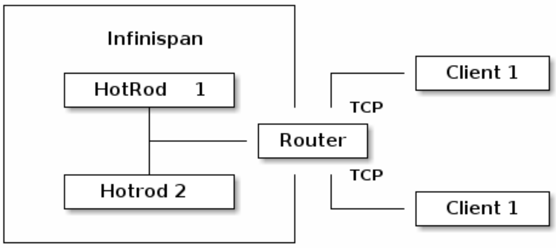


Figure 4.5: Multiple data containers and a router

pattern is very common when designing network protocols and is used by many popular libraries, for example JGroups framework.

The router uses concept called Routes for forwarding network traffic from an input to a proper Hotrod instance. Each route consists of ‘RouteSource’ and ‘RouteDesitnation’. A ‘RouteSource’ is responsible for recognizing used network protocol and gathering all necessary information to distinguish clients - in case of SNI based ‘RouteSource’ this would be also SNI Hostname. A ‘RouteDestination’ is responsible for forwarding data to a proper component - in case of Hotrod server it takes all its handlers and adds them to the pipeline (Figure 4.5).

The algorithm requires initiating the TLS connection between a client and a server (line #7). Since the solution assumes using encryption, every non-TLS connection is rejected (line #20 and #21). Once the TLS handshake procedure finishes, the algorithms extracts the TLS/SNI ”hostname“ field and stores it into `shn` variable (line #8). The next step is to analyze the routing table configured in the server and extract only those routes, that can be routed from TLS/SNI incoming source to Hotrod destination and match ”hostname“ field (stored previously in (line #8). This filtering has been implemented using lambdas in lines #12 - #15. It is expected that only one route matches these conditions. If none of them matches the criteria, an exception is thrown (line #17). Finally, the routing algorithms removes itself from communication pipeline and attaches handlers responsible for sending the traffic to proper Hotrod Server.

The algorithm was designed with performance in mind. The algorithm complexity analysis should take the worst case into account. Therefore, the Big-O notation is the best tool to do it [113], since it provides only an asymptotic upper bound for a given function $f(n)$. A formal definition of the Big-O notation is presented in the Equation 4.1.

```
Input:  Inbound TLS/SNI connection ic
        SNI Handler sh
        Routing table rt
        Connection pipline p
Output: RouteDestination rd

if(sh.handshakeAccepted(ic))
  shn = sh.getHostname()
  Collection<Route> r = rt.getRoutes()
  // Use lazy evaluation for routes
  Route rd = r
      .filter(ri -> ri.source() is SNISource)
      .filter(ri -> ri.sniHostName() == shn)
      .filter(ri -> ri.desitnation() is HotrodDestination)
      .getFirst()
  if(rd == null)
    throw Exception("No Route")
  pipeline.removeHandler(sh)
  pipline.addHandlers(rd.sestination().handlers())
else
  throw Exception("No TLS/SNI Connection")
```

Figure 4.6: Routing function implementation pseudo-code

$$g(f(n)) = \begin{cases} f(n) : there\ exist\ positive\ constants\ c\ and\ n_0\ such\ that \\ 0 \leq f(n) \leq cg(n)\ for\ all\ n \geq n_0 \end{cases} \tag{4.1}$$

The proposed algorithm's complexity is proportional to the number of routes in the routing table. Using a Big-O notation it might be written as $O(n)$.

An optimized implementation has been proposed as a multi-tenancy implementation for Infinispan project [1].

4.3 Experiment environment description and tools used for the evaluation

The experiments were performed in three stages - initial performance tests were executed on local laptop, validation tests were performed on OpenShift 3.11 cloud and the validation and performance tests were executed inside Red Hat performance lab using a tool called PerfAck.

[1] The code can be accessed using the following URL https://github.com/infinispan/infinispan/pull/4348

Test description	Iterations	Test result [ms/op]	± Error
1 Data Container without TLS/SNI router	31	3.046	0.096
1 Data Container with TLS/SNI router	31	12.725	0.379
2 Data Containers with TLS/SNI router	31	13.471	0.449

Table 4.3: Initializing connections

The initial tests were performed on Intel® Core$^{\text{TM}}$ i7-4900MQ CPU @ 2.80GHz, 16 GB RAM laptop using Linux OS (Fedora F23). All testing scenarios were written in Java using JMH [77] tool (a micro-benchmark framework).

The validation tests were performed using OpenShift 3.11 cloud deployed on AWS. The system consisted of 3 Master nodes and 2 Worker nodes. Such a system was initialized solely for running multi-tenancy test. Most of the background tasks were turned off to minimize interference with testing harness.

4.4 Experiments results

The implementation enables the TLS/SNI-based connections to Infinispan server using a Java Hotrod client. A 'RouteSource' and a 'RouteDestination' interfaces are generic enough to connect all the various types of services. However, this flexibility comes at the cost of performance. In order to determine how much overhead it adds, a JMH [77] test was performed and the results were gathered in Table 4.3 and Table 4.4. Every connection from a client is handled by the Hotrod Endpoint (a frontend component of Infinispan server), that may use the TLS/SNI router plugin. The plugin is responsible for connecting a client to a proper data container. The aim of both experiments is to measure performance overhead for both initiating a secured connection and performing operations against the data grid system. The tested scenarios include a system without proposed solution ("1 Data Container without TLS/SNI router"), a system with proposed solution and a single data container ("1 Data Container with TLS/SNI router") and a system with proposed solution and two data container ("2 Data Containers with TLS/SNIrouter"). The scenario with multiple data container might be referred as "multi-tenancy".

Table 4.5 and Table 4.6 contain calculated significance measure between each tested pair of benchmarks based on the procedure described in Section 3.8. For $\alpha = 0.05$ (99%), the tested interval is equal to $(-\infty, -2.58 > \cup < 2.58, \infty)$. The calculation results show that all the benchmark pairs are statistically significant.

Additionally, the implementation slightly increases heap usage as the new routing component is being used. The total amount of occupied in memory (RSS - Resident set size) by the data grid process has been gathered in Table 4.7.

Test description	Iterations	Test result [ms/op]	± Error
1 Data Container without TLS/SNI router	31	430.075	18.088
1 Data Container with TLS/SNI router	31	847.909	34.479
2 Data Containers with TLS/SNI router	31	1022.655	32.316

Table 4.4: Performing 10 000 Cache PUT operations (op in this context is equal to 10 000 put operations)

	2 Data Containers with TLS/SNIrouter	1 Data Container with TLS/SNI router
1 Data Container without TLS/SNI router	3.27	
Single server without SNI	58.49	103.91

Table 4.5: Significance measure matrix between benchmark results for initializing connections

	2 Data Containers with TLS/SNIrouter	1 Data Container with TLS/SNI router
1 Data Container without TLS/SNI router	9.52	
Single server without SNI	41.22	27.64

Table 4.6: Significance measure matrix between benchmark results for performing 10 000 Cache PUT operations

Test description	Occupied memory
Multi-tenancy turned off	482 MB
Multi-tenancy turned on	491 MB

Table 4.7: Total memory usage (RSS) with and without multi-tenancy feature

Number of tenants	Occupied memory
1	491 MB
3	164 MB
5	99 MB
10	49 MB

Table 4.8: Total memory usage (RSS) compared with the number of tenants

4.5 Results analysis

Deploying systems in a virtualized environment, like container-based clouds, gains popularity very quickly. When considering data store applications like databases or data grids, hosting a single service for multiple tenants is one of the best ways to lower operational costs by lowering memory consumption per single tenant (or user). Multi-tenancy is not an easy concept from the implementation perspective as it requires special care for separating data containers between tenants. In many cases, storing data in the cloud also requires confidentiality applied to both the data container and transport. The TLS/SNI router implementation turns out to be very helpful when addressing both those concerns at the same time. TLS ensures the connection is encrypted and SNI allows to recognize tenant and helps choosing the right partition to access.

In case of transmitting public data that do not require encryption, some additional HTTP headers can be used to help the router make proper data access decision. It is worth mentioning, that using other authentication methods is a strong requirement in this case. Otherwise, separating data between tenants could not be achieved. The router implementation is to be flexible enough to support all the scenarios mentioned above (additional RouteSource implementations are required in that case).

Using TLS/SNI for routing adds some additional overhead for communication (more than 1 milliseconds for initializing connection and 175 milliseconds for performing 10 000 operations). Additional delay for initializing connection is expected and usually does no harm, but adding 17 microseconds per each operation might be a bit too high for the performance-sensitive scenarios.

Introducing the TLS/SNI router implementation requires additional memory. For the testing environment, the total system occupied by the data grid grew by 1.8 percent (from 482 to 491 MB). However, with the increasing number of tenants, the effective amount of memory allocated per tenant gets lower. Table 4.8 shows the effective allocated memory compared to the number of tenants. As the results show, with 10 tenants, the occupied memory is less than 50 MB (which might be considered as a very good result for a Java Virtual Machine process).

The memory consumption test proves that the thesis aims have been partially achieved. Even though the lowering memory consumption goal has been met, the overall performance metrics with the TLS/SNI router are worse.

In the future, the routing algorithm can be further optimized to meet even more strict performance requirements. The Route filtering mechanism seems like a good candidate for optimization, but it can speed up only the initializing connections (after the handshake, the routing handler removes itself from the pipeline). Exposing settings for tuning connection pools, as well as using EPoll are much better candidates for optimization. Servers like Infinispan Hotrod are designed carefully for optimal performance and the router should expose as many configuration parameters as possible to allow fine tuning for achieving the best possible results in each scenario.

4.6 Limitations

Systems implemented using Java usually require large amounts of memory in runtime. This also applies to Infinispan. Even though a multi-tenancy implementation helps lowering memory footprint of a data grid server, its effectiveness is closely related to the number of tenants using the system. This means the proposed solution will not be very effective for a small number of users. In such scenarios, there are better tools that might be used for storing the data, such as data grid solutions written in C/C++ or Go.

Another limitation of using TLS/SNI is the necessity of encrypting the traffic. Although doing it is considered as a good practice in most of the cases, in high performance scenarios without storing sensitive data, it might be considered as a drawback. As shown in Table 4.3, initiating TLS connection is almost 10 ms slower than initiating a non-encrypted connection. Performance test results shown in Table 4.4 indicate the transmission that is almost twice slower over the encrypted transport. All this is expected but in some scenarios, application developers favor performance over security (especially if they trust their infrastructure).

4.7 Further work

After evaluating the prototype, the multi-tenancy implementation has been introduced to FeedHenry project [31]. The goal was to replace existing Redis-based cache infrastructure with a multi-tenant Infinispan data grid. During the evaluation, the FeedHenry team noticed that one of the key elements for a multi-tenant system is to be able to add new tenant with zero downtime. Since the implementation used a static routing table, fulfilling this requirement was impossible. Along with new requirements, the Infinispan team decided to re-architect the Infinispan Server to allow faster boot time. Faster boot time allows to workaround the dynamic reload problem with Rolling Upgrade procedure [126]. Once a new configuration is applied to the cluster, each server is dynamically replaced by a new replica with updated configuration.

The multi-tenancy feature was also backported to the product - Red Hat Data Grid. Since then, it became a fully-supported solution used by many customers of Red Hat company.

Chapter 5

Proposed solution for exposing clustered applications deployed in the cloud

5.1 Introduction

Components offered by modern cloud vendors are often dedicated to a standard web applications. In most cases, such an application consists of backend services, that implement business logic, a frontend web server, that communicates with users, and a data store - typically a relational database. When an application hosts content for multiple users, it is often recommended (or even required) to use a caching layer between the application backend and the data store layer. Caching is also a good fit for modern application architectures, such as microservices [56] where a single user request might cascade into multiple calls between backend services. Each call introduces additional latency and decreases the overall user experience.

Modern caching solutions, often known as Data Grids, need to respond to high application demands and offer data storage within a distributed system without a single master node. This goal can be achieved by using consistent hashing for data partitioning which allows data access with $O(1)$ complexity [32]. Consistent hashing can also be used by a remote client application for calculating which node in a data grid cluster owns the data, and how to access the data within the shortest possible path (without redirecting from one node to another).

Each system needs to be deployed on appropriate hardware in order to provide value to users. Each system may have different demands, and IT departments inside companies might focus on different characteristics (e.g. time to market, or time required to onboard a new development crew member). Cloud computing

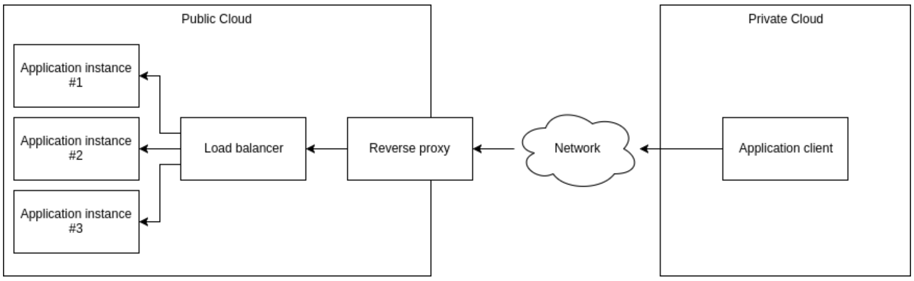


Figure 5.1: Accessing application instances between clouds

fits perfectly into those requirements and has therefore gained popularity very quickly [35]. Enterprise application demands very often require a hybrid approach between a private cloud (where all the components are installed in a standard on-premise data center) and a public cloud (where all the components are hosted by a cloud vendor). Those scenarios are especially challenging from the point of view of discovery and connectivity. Some cloud platforms (such as Kubernetes [55] or OpenShift [78]) encapsulate all the traffic within the cloud, exposing a single point for accessing inner components (Figure 5.1). Since most vendors aim for a standard web application which uses the HTTP(S) protocol for communication with the client, the component responsible for ingress traffic is usually a Reverse Proxy rather than an edge router or L4 (Transport Layer of OSI model) Load Balancer. A Reverse Proxy usually offers more features related to the HTTP protocol (such as load balancing and content compression) than an edge router (which is focused on L3/Network routing) or L4 Load Balancer (which uses transport layer information to make balancing decisions). This approach is very convenient from the security's point of view, where a single component is responsible for both defending the system against unwanted traffic (the firewall) and handling encryption/decryption (SSL termination).

Although this architecture makes a lot of sense for a typical application, some types of deployments need to consume topology information in order to optimize system performance. Common examples are high efficiency data grids and gaming systems. Many video games use RTP (Real-time Transport Protocol), which was originally designed for delivering audio and video content, for communicating with one or more dedicated servers. Those systems can be viewed as if they were using a client-based load balancing technique [119] where topology information is essential to fulfill all application requirements. Apart from caching and gaming, there are also a number of hybrid cloud applications where part of the cluster is hosted inside a public cloud and part of it is hosted within a traditional on-premises data center. This approach can sometimes be seen when using the quorum-based systems where the majority of instances are

Solution	Priority
Discoverability	Must-have
Connectivity	Must-have
Possibility of automation	Good-to-have
Small performance penalty	Good-to-have

Table 5.1: Requirements for proposed solution

hosted in the cloud for stability, while the minority are hosted within a data center to improve communication speed.

Enabling client-based load balancing requires achieving two major properties - discoverability and connectivity. Discoverability represents a way to identify topology information - in particular the number of server instances, exposed ports and IP addresses. This enables the client to configure the connection pool according to that information. Connectivity represents an ability to connect to those internal instances directly. It is also worth mentioning that both the caching and gaming industry have very high demands on application throughput at low latency. Therefore, an ideal solution needs to have minimal negative impact on both of those characteristics.

5.2 Proposed solution for exposing clustered applications deployed in the cloud

The major contribution of this proposal is to design a solution for exposing a clustered, consistent hash-based service deployed in the cloud, so that it can be accessed from an external topology-aware client. This solution has been tested using a dedicated testing environment and the results are satisfactory.

Apart from the aim of this thesis, solutions for exposing clusters deployed within the cloud to the outside world need to fulfill discoverability and connectivity criteria. A client application needs to be able to discover nodes in the cluster, as well as to establish connections to individual cluster members. A good solution also needs to introduce the smallest possible penalty on performance (defined by the throughput in operations per second). The final goal is full process automation which is required from a cluster auto healing perspective. A full list of requirements and priorities is presented in Table 5.1.

One of the initially evaluated solutions was securing the transport with TLS and use of the SNI extension to decide which server receives the request from the client (Figure 5.2). This solution is natively supported by many Reverse Proxy applications used by cloud vendors (such as HAProxy [43] or NGNIX [74])

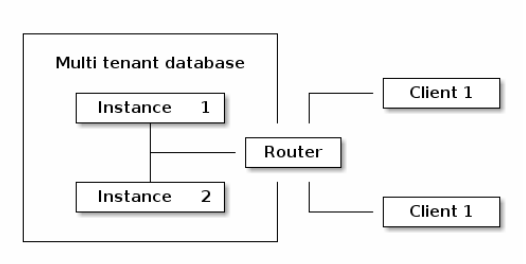


Figure 5.2: The router approach for system design

Test description	Result [ms/test]
Single server without TLS	430.075
Single server with TLS/SNI	847.909

Table 5.2: Performing 10 000 Cache PUT operations with different TLS configuration [97]

thus it can be easily automated. The biggest downside of this approach is that the performance is 30% lower (Table 5.2) than when using non-TLS encrypted transport.

An alternative approach is to use a custom Load Balancer application which is able to decode additional, non-standard TCP and UDP options. This options could be used to send and receive topology information. However, deploying a custom Load Balancer is not possible in many cases (usually it is a cloud administrative task). Even though this solution has been rejected from further exploration, using a dedicated component for routing behind a Load Balancer was used as a foundation for exploring Reliable Asynchronous Clustering [89] and Cross Site Replication using Gossip Routers [104]. At the time of writing this thesis, both solutions are being implemented into Infinispan project.

The next group of possible solutions are those based on the use of Load Balancers provided by the cloud vendors. For many cloud vendors, Load Balancers and reverse proxies are considered one and the same; however, they are considered separately in this research. Most solutions of this nature (such as Amazon Elastic Load Balancer [29] and Google Cloud Load Balancer [36]) are optimized to serve high volume traffic whilst introducing the smallest possible loss of performance. Network Virtualization efforts which integrate seamlessly with both Load Balancers and SDN are used within the cloud to achieve much better results than ever before [39][37]. To solve the connectivity problem, it is possible to allocate a Load Balancer per node in the cluster. Since creating a new Load Balancer and a new binary proxy very often requires a REST API

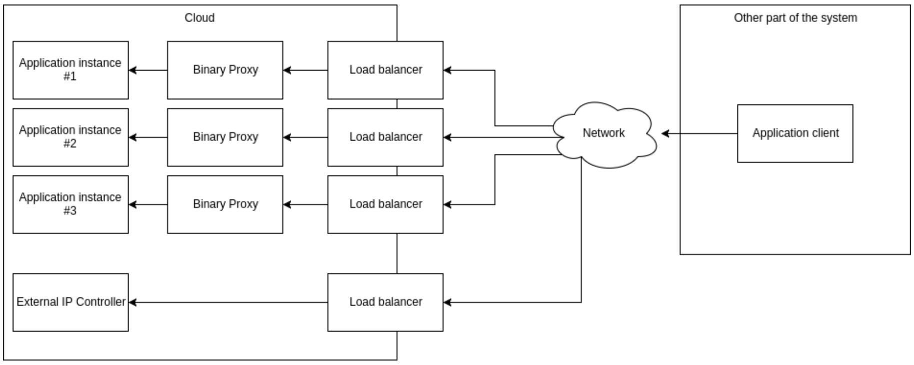


Figure 5.3: The proposed system

call to an API exposed by a cloud vendor, this process can be easily automated. Finally, the discoverability problem can be solved by creating a new application which exposes mappings between internal and external IP addresses for a cluster deployed in a cloud.

The final proposal is presented in Figure 5.3. Each Load Balancer instance has a publicly accessible IP address, and in fact, many cloud vendors treat Load Balancers as an external/virtual IP implementation. Each request is passed into a binary proxy (a proxy which forwards messages sent using binary protocols), then finally to an application instance. The motivation behind implementing proxying is that in container-based clouds, application instances can often be restarted (e.g. because of balancing hardware resources utilization or auto-scaling in and out process). The proxy acts as a safety buffer pausing any communication until application instances are ready to process the incoming requests. Another argument is that creating and destroying Load Balancer instances can take some time (on Google Cloud Platform it is a matter of minutes). With binary proxies, the number of create/destroy events can be limited.

The use of Load Balancers and binary proxies fulfills the connectivity requirement. A newly created component called an External IP Controller instance is responsible for automating the process of discovering new application instances and creating proxies, as well Load Balancers for all instances. This component also exposes a REST endpoint (which serves YAML-based content with internal/external IP address mapping) that helps the client application to determine which application instances are mapped to which addresses. This thereby satisfies the discoverability requirement. However, during application benchmarking it turned out that implementing binary proxy communication degrades the performance (Table 5.3).

Therefore, it was decided to remove binary proxy (along with all its benefits) and propose the final optimized prototype as own solution.

Test description	With binary proxy	Result [ms/op]
Perform 10 000 Put Operations	Yes	352.822
Perform 10 000 Put and Get Operations	Yes	131.926
Perform 10 000 Put Operations	No	131.926
Perform 10 000 Put and Get Operations	No	221.711

Table 5.3: Binary proxy benchmark results

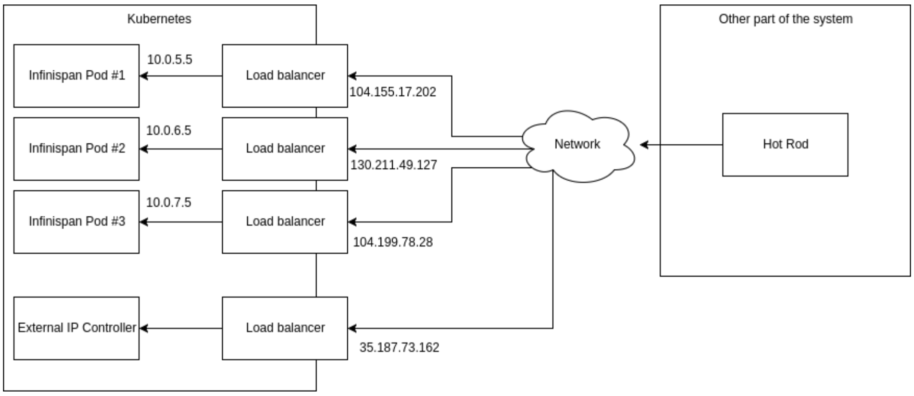


Figure 5.4: The diagram of the Evaluated system

The final prototype was implemented using open source technologies such as Kubernetes. The main argument for choosing Kubernetes is that it does not depend on any particular cloud vendor (in other words it can be hosted using any public cloud provider such as Google Cloud Platform or Amazon Web Services). As for client/server communication, Infinispan is a data grid implementation which uses a Consistent Hash algorithm for locating primary owner [32] nodes for a specific portion of the data. This optimization tactic allows to asses whether the prototype improves communication performance or not. Figure 5.4 presents the evaluated prototype system. Each application instance (deployed as Kubernetes Pod) has its own unique, local IP address, which is not accessible from the Internet. In order to make each instance available from the Internet, a Load Balancer was used. Each Load Balancers points to a single application instance, which allows to make the routing decisions very efficiently (the routing table contains only one entry). A Hotrod client was deployed on a local laptop and communicated over the Internet with application instances deployed in the cloud.

Hotrod is the name of Infinispan's custom binary protocol used for transmitting data to/from the Infinispan server. The Hotrod client obtains topology information upon its first connection to the cluster. The topology contains a list of servers and their internal addresses (10.0.5.5, 10.0.6.5 and 10.0.7.5). The External

```
1   Input: External address
2   Output: Internal address
3
4   externalAddress = internalAddress
5   if (needsExternalMapping())
6       externalAddress = getExternalAddressFromCache(internalAddress)
7       if (externalAddress == null)
8           externalAddress = queryExternalIpController(internalAddress)
9           putInExternalAddressCache(internalAddress, externalAddress)
10  return externalAddress
```

Figure 5.5: Hotrod internal/external address mapping algorithm

IP Controller service provides mapping (using a REST service) between internal (e.g. 10.0.5.5) and corresponding external addresses (e.g. 104.155.17.202). This information allows the client application to reconstruct the consistent hash provided by the Infinispan server. Each topology update follows the same update pattern on the client's side. The biggest advantage of this approach is that the modification only needs to be applied to the client's code [1], and the server is not aware of any translation process.

The algorithm used by the Hotrod client is presented in Figure 5.5. The algorithm takes an external address as a parameter and produces an internal address that is used between application instances to communicate with each other. In order to optimize the algorithm, a local cache has been used that contains already resolved mappings.

Lines #1 and #2 represent external and internal Hotrod Server address. The client checks if the external mapping addresses feature is turned on (Line #5) and performs translation. The code in Line #6 queries a local address translation cache (the cache uses expiration logic to avoid stale entries) and if an entry is not found, it queries the External IP Controller to obtain an external IP address corresponding to the internal one. Once mapping is obtained, it is being put into the local cache (Line #9).

The creation and removal of Load Balancers was implemented within an External IP Controller. The algorithm used by the controller was designed to run indefinitely. The application thread wakes up every 5 minutes to query the Kubernetes API for all application instances. To create a Load Balancer for each application instance, the controller needs to set marker labels on each instance. Later, Kubernetes Services selects proper instances based on a Selector, where the Selector query and marker labels need to match. Figure 5.6 presents the algorithm in pseudo-code notation. The algorithm uses the Reconciler Pattern

[1] The code can be accessed using the following URL https://github.com/infinispan/infinispan/pull/5164

```
1  while (true)
2    applicationInstances = queryPlatformForInstances()
3    for (applicationInstance : applicationInstances)
4        addMarkerLabels(applicationInstance)
5        ensureLoadBalancerIsRunning(applicationInstance)
6    removeUnnecessaryLoadBalancers()
7    sleep(5 min)
```

Figure 5.6: External IP Controller algorithm

[47], meaning it aggregates the current state of the system, assembles the desired state and performs reconciliation.

The endless loop was implemented using `while` loop (Line #1). The algorithm queries the platform (Kubernetes) for all Hotrod Server instances (Pods). Next, it iterates over the set and adds proper marker labels (Lines #4). Kubernetes platform uses Selectors that need to match objects with proper Labels. Each application instance needs Labels matching Selector used on a Load Balancer object. Line #5 is responsible for spinning Load Balancers up and writing their External IP addresses into a local cache (this part was omitted from the algorithm diagram). Finally, the algorithm removed all the redundant Load Balancers (if any). The next loop starts after a 5 minute period.

5.3 Experiment environment description and tools used for the evaluation

The experiments were performed in three stages - initial performance tests were executed on a local laptop, validation tests were performed using Minikube project [66] as well as on Google Compute Cloud.

The initial tests were performed on Intel® Core$^{\text{TM}}$ i7-4900MQ CPU @ 2.80GHz, 16 GB RAM laptop using Linux OS (Fedora F23). All testing scenarios were written in Java using JMH [77] tool (a micro-benchmark framework).

The validation tests were performed on a local laptop using Minikube project that simulates a running cloud using Virtual Machines.

The final test were performed on Google Compute Engine (a Kubernetes cloud maintained by Google) and on Google Compute Cloud (for benchmarking traffic within the same data center).

Test description	Short description	Average value [ms/op]	± Error
Topology aware client inside Kubernetes with Kubernetes internal addresses	L3 + Kubernetes internal	1288.437	136.207
Topology aware client inside Kubernetes with Kubernetes Load Balancers	L3 + Kubernetes internal + LB	1461.510	64.33
Simple client inside Kubernetes with Kubernetes Load Balancers	L1 + Kubernetes internal + LB	2163.873	141.332
Topology aware client inside the same data center with Kubernetes Load Balancers	L3 + Kubernetes external + LB	2465.586	85.034
Simple client inside the same data center with Kubernetes Load Balancers	L1 + Kubernetes external + LB	2684.984	114.993

Table 5.4: Benchmark results for performing 1.000 Put and Get operations

5.4 Experiments results

System benchmarks were established using topology-aware and a simple Hotrod client. The topology-aware client uses consistent hash information for sending each request to a specific server, whereas the simple client chooses a random server from a specified connection list. Each test consists of sending 10 000 Put Operations (which means inserting 10 000 random strings into the data grid) and 10 000 Put and Get Operations (where the Get Operation represents getting a value previously inserted by a Put Operation). Each test was run 31 times and an error margin was calculated using a 99.9% confidence interval. The results have been shown in Table 5.4 as well as in Figure 5.7. In each diagram describing the experiments results, the vertical axis represents time in milliseconds and each bar represents a values for a particular solution. The aim of the test is to verify what is the overhead of using Load Balancers in cloud (by comparing test results using topology-aware - ”L3 + Kubernetes internal“ and ”L3 + Kubernetes internal + LB“). The next goal is to verify if using topology aware client improves performance (by comparing ”L3 + Kubernetes internal + LB“ and ”L1 + Kubernetes internal + LB“). The final goal is to compare a system with and without the proposed solution (by comparing ”L1 + Kubernetes external + LB“, that represents the baseline, with ”L3 + Kubernetes external + LB“, that represents the proposed solution).

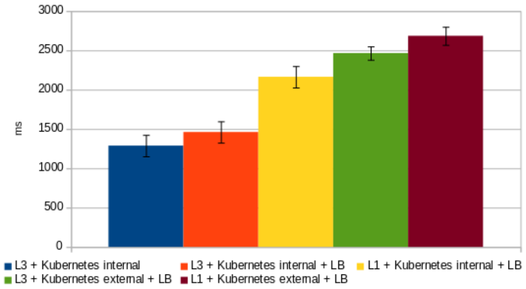


Figure 5.7: Benchmark results for performing 1.000 Put and Get operations

	L3 + Kubernetes internal	L3 + Kubernetes internal + LB	L1 + Kubernetes internal + LB	L3 + Kubernetes external + LB
L3 + Kubernetes internal + LB	-4.307			
L1 + Kubernetes internal + LB	-36.182	-11.254		
L3 + Kubernetes external + LB	-39.018	-10.719	1.481	
L1 + Kubernetes external + LB	-16.288	-12.703	-7.96	31.4

Table 5.5: Significance measure matrix between benchmark results

Table 5.5 contains calculated significance measure between each tested pair of benchmarks based on the procedure described in Section 3.8. For $\alpha = 0.05$ (99%), the tested interval is equal to $(-\infty, -2.58 > \cup < 2.58, \infty)$. The calculation results show that all the benchmark pairs, except ”L1 + Kubernetes internal + LB“ and ”L3 + Kubernetes external + LB“ one, are statistically significant.

5.5 Results analysis

Exposing services hosted within the cloud to external consumption is not very popular, yet it is very useful for some use cases. Many cloud vendors have sophisticated tools or templates that allow them to bootstrap such services very quickly. Automatic auto scaling based on CPU and memory metrics, health monitoring, and automatic backup management are only a few reasons why hosting applications within the cloud is preferred by many development teams. Many cloud vendors use Load Balancers and Reverse Proxies as the only way of reaching services hosted within the cloud from the the outside world. Very frequently, the process of provisioning Load Balancers is strictly connected to creating new firewall or port forwarding rules. The downside of using Load Balancers is that they are relatively expensive and very often charged on hourly basis.

The network infrastructure maintained by cloud vendors is very often optimized for using Load Balancers. Comparing "L3 + Kubernetes internal" and "L3 + Kubernetes internal + LB" benchmark results from the Table 5.4 showed only 13.43% lower throughput when sending data through a Load Balancer. The binary proxy implementation achieved much worse results. Comparing "Perform 10 000 Put and Get Operations" with and without binary proxy from the Table 5.3, resulted in 68.05% worse performance.

The communication performance degrades much further when using other Virtual Machines within the same cloud offering (but not in the same container cloud). An example in the benchmark used a Kubernetes cluster running on a CentOS VM and a client application deployed in a separate VM instance. Comparing benchmarks for a topology aware client ("L3 + Kubernetes external + LB" and "L3 + Kubernetes internal + LB") from the Table 5.4 showed 68.70% performance loss when the client is deployed in a separate VM instance. Benchmarks for a client that does not use consistent hash-based routing the results were much better and stabilized at 24.08% worse performance when the client is deployed in a separate VM instance. The figures might significantly vary from cloud vendor to cloud vendor, due to network optimizations used in the data center.

The solution proposed in this thesis helps to achieve better throughput when connecting a topology aware client to a cluster of servers deployed in the cloud. Comparing ”L1 + Kubernetes internal + LB“ and ”L3 + Kubernetes internal + LB“ from Table 5.4 indicates over 30% increased overall communication performance. The results for ”L1 + Kubernetes external + LB“ and ”L3 + Kubernetes external + LB“ represent less than 1% but it is worth mentioning that this includes latency between Google data center and testing environment. In a typical scenario, the link between the data grid client and the data grid cluster is expected to be much faster.

From the transport encryption’s point of view, adding TLS resulted in the 30% worse performance. The performance penalty was a deciding factor to skip the transport confidentiality aspect from this research. However, all the proposed solutions allow TLS to be turned on.

5.6 Limitations

Allocating a separate Load Balancer per each application instance might be very costly. A large data store system that consists of tens or even hundreds of application instances, exchanging large amounts of data with client application may result in very high operating costs. Table 6.1 contains pricing per allocating Load Balancers in major public clouds.

The proposed solution adds a new component on top of the existing data store system - External IP Controller. The controller is responsible for querying the data grid for the list of internal IPs and collecting corresponding external IPs (from Load Balancers). In most of the cases, this is a one-to-one mapping but some cloud vendors use multiple external IPs (or domain names) corresponding to a single Load Balancer (the Elastic Load Balancer exposes a separate IP per each Availability Zone). This may lead to additional complexity in either data grid client code or External IP Controller. It is also worth mentioning, that an additional component needs to highly available and accessible for all the clients.

Recently, a new initiative for interconnecting clouds has been founded - Submariner project [106]. Its aim is to allow communication between application instances deployed in different sites. At the time of writing this thesis, the project is in development stage but once it becomes stable, it will offer an easier way of communicating data grid service and its client than presented in this thesis. However, the use cases of communicating an external client with a data grid system deployed in the cloud will still remain valid.

5.7 Further work

Even though the benchmark results look promising, there is still the need to benchmark other scenarios (e.g. involving disk access) and different types of services. Furthermore, this research did not investigate the security aspects of the proposed approach. Exposing internal to external (and vice versa) address mapping might be used as a potential attack vector. Some existing implementations like Infinispan may mitigate this risk by putting address mapping into the server. This way the client obtains an already-altered list of servers and is not aware of any mapping process.

The approach presented in this thesis has been used as the foundation for other improvements, such as Reliable Asynchronous Clustering [89] and Cross Site Replication using Gossip Routers [104].

Chapter 6

Proposed solution for switching communication protocols in the cloud

6.1 Introduction

Modern application development trends show that Microservices Architecture deployed on a container-based cloud gains popularity very quickly[56]. Services in such a system often communicate with each other using the HTTP protocol and REST interface [93]. Modern container-based cloud solutions, like Kubernetes [55] or OpenShift [78] embrace that model and offer an intuitive way to build large systems using the provided building blocks. One of the most important platform components are Services, which represent an internal Load Balancers and are implemented by the IPTables (also known as Netfilter [114]) project. Routing traffic from the outside of the cloud is more complicated and can be achieved using one of the two ways - exposing a Load Balancer (also known as a Load Balancer Service) or adding a route to a publicly available router (also called an Ingress of a Router, often implemented using HAProxy [43] or Nginx [74]) [98]). Unfortunately, allocating a Load Balancer is often quite expensive, so most developers focus on using the public router. This enforces the HTTP based communication and offers little to none support for custom, binary protocols based on TCP or UDP transports.

6.1.1 Network traffic in container-based clouds

Most of the on-premise data centers as well as public clouds use a standard routing model, which consists of a Reverse Proxy, a Load Balancer and, one or many, application instances (Figure6.1). A typical web application fits perfectly into

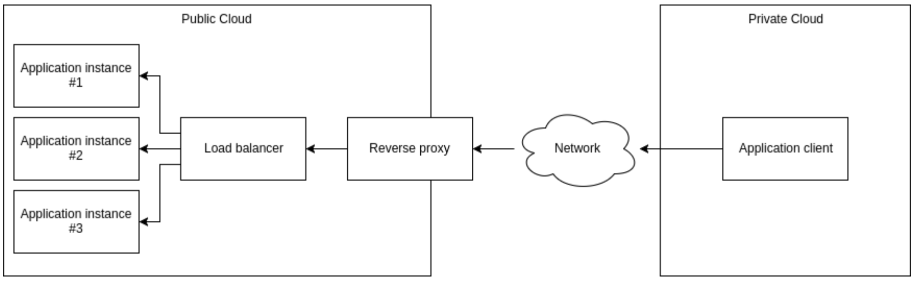


Figure 6.1: A standard routing model in modern application deployments

Cloud Provider	Cost
Amazon Web Services Elastic Load Balancer	0.025 USD per hour and 0.008 USD per GB
Google Cloud Platform Forwarding Rule	0.025 USD per hour and 0.008 USD per GB

Table 6.1: Load Balancer pricing by the biggest cloud vendors

this model. In most cases, an ingress traffic is generated by the end users who use their desktop or mobile devices. Therefore, a Reverse Proxy is an HTTP oriented component and offers only a limited set of capabilities for other network protocols, such as high performance binary protocols based on TCP (or UDP) transport. Some cloud vendors do not use the latest version of the Reverse Proxy software and do not fully support the HTTP/2 protocol [45]. This makes achieving decent performance in the cloud environment even more challenging.

Some of the cloud vendors allow users to allocate an externally reachable L4 Load Balancer (L4 is a transport layer of the OSI Model). Unfortunately, such solutions are often quite expensive considering large scale systems. Table 6.1 contains a pricing model for an externally reachable Load Balancer provided by major cloud vendors [11][40].

Adding a new route to an externally reachable Reverse Proxy is free of charge in most cases. Presented solution allows application developers to optimize costs by using a standard Reverse Proxy and increase the performance of their applications by enabling custom, binary protocols.

6.1.2 Multiprotocol applications

There are two types of communication protocol switching mechanisms - the HTTP/1.1 Upgrade procedure [92] and TLS/ALPN [96].

The HTTP/1.1 Upgrade procedure (sometimes called "clear text upgrade") is often used when the client and the server communicate using unencrypted TCP

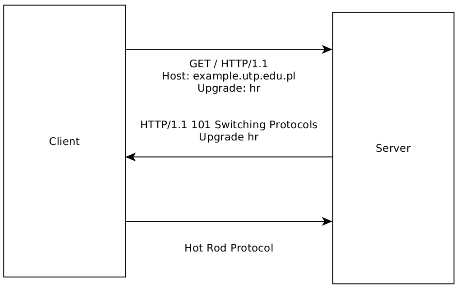


Figure 6.2: HTTP/1.1 Upgrade flow [92]

connection. The procedure allows the client application to send an HTTP/1.1 message containing an "Upgrade" header to invite the server to switch the communication protocol reusing the same TCP connection. The server may ignore such a request or send an HTTP/1.1 101 status code (Switching Protocols) and accept one of the proposed protocols by the client. After sending a HTTP/1.1 101 message, the server immediately switches to the new protocol reusing the same TCP connection. Figure 6.2 shows an example of upgrading existing HTTP connection to a custom Hotrod protocol.

It is also possible to encrypt the connection between a client and a server by using the TLS protocol. Initiating an encrypted connection requires completing handshake subprotocol (as described in Section 3.3). During the handshake, the client and the server may use one of the TLS extension. The ALPN extension (Application-Layer Protocol Negotiation) allows the client and the server to negotiate communication protocol for the given TCP connection. Figure 6.3 shows a simplified diagram of the TLS/ALPN communication protocol negotiation.

Both the TLS/ALPN and HTTP/1.1 Upgrade procedure are commonly used by web browsers and application servers to switch from HTTP/1.1 to HTTP/2. However, it is also possible to switch into some other specified protocol - even a custom binary one.

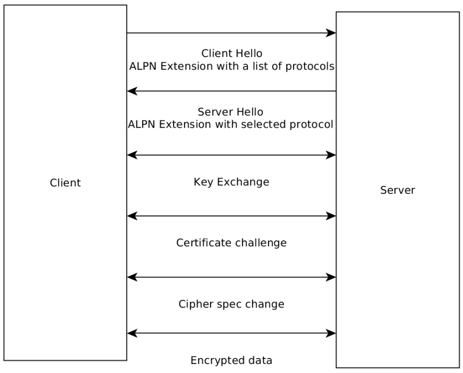


Figure 6.3: TLS/ALPN flow [92]

6.2 Proposed solution for switching communication protocols

The major contribution of this proposal is to design a new mechanism for switching communication protocols and enabling a client and a server to switch to a custom, binary protocol in the cloud. The proposed solution does not require allocating a dedicated Load Balancer and allows to reuse a publicly available cloud provider router. The mechanism has been tested using a public cloud environment and the results are satisfactory.

Services hosted within the cloud are often allowed to use any communication protocol. Most of the Open Source, container-based clouds use the SDN technology (Software Defined Network) to address layer 3 routing (the network layer of the OSI model). Therefore, the only limitation is to use IP-based routing (TCP is often not a requirement and other transport protocols, such as UDP can be used). Handling an ingress traffic in the cloud is often done by using an edge component, a Load Balancer or a Reverse Proxy. A Reverse Proxy is a component designed to handle the HTTP or TLS-encrypted traffic. An application hosted in a container-based cloud can use a Reverse Proxy without additional costs. Allocating a Load Balancer tied to an application very often requires paying additional costs. Therefore, it is highly desirable that the proposed solution should be capable of reusing a Reverse Proxy and not allocating a dedicated Load Balancer. Another requirement for the proposed solution is encrypting the traffic between the client and the server in order to protect sensitive and confidential data. In case of transmitting such data over the wide open Internet, it is necessary to protect the transport using the TLS protocol (described in Section 3.3). There

Requirement	priority
Switch to custom protocol	Must have
Possible TLS encryption	Must have
Simple client implementation	Nice to have
Expose all protocols using one TCP port	Nice to have

Table 6.2: Solution requirements.

are two additional lower-priority (nice-to-have) requirements for the proposed solution. The first is the switching procedure, which could be implemented in the client application side. The reason is to make it less error-prone. The second is to expose all the protocols on one TCP port. This lowers the costs on the service provider's side since there is the need for only one route.

The Table 6.2 contains all the gathered requirements with priorities assigned.

There are two known ways of negotiating communication protocol between a client and a server. The former is based on TLS/ALPN [96]. During the TLS Handshake subprotocol, the client sends the custom binary protocol as the first element of the TLS/ALPN protocol negotiation list. In most of the cases, the list contains only one protocol supported by the client (one of the most common exceptions is multi-protocol client applications; however, such application are out of the scope of this research). If the server does not support the proposed protocol, the connection is terminated. The latter solution is based on HTTP/1.1 Upgrade procedure [92] and can be used in all the situations where encryption is not necessary. In this solution, the client application needs to be adjusted to send an empty HTTP/1.1 GET request to the root context of the URL with an "upgrade" header. The header is allowed to contain multiple values; however, in most of the cases it will contain one element - the target binary protocol. The server might ignore such a request and in that case, the client terminates the connection.

Both the HTTP/1.1 Upgrade and TLS/ALPN procedures allow switching to a binary protocol and both of them allow to specify multiple protocols in the request (either using the TLS/ALPN or HTTP/1.1 Upgrade). In most of the cases TLS/ALPN is handled by the OpenSSL library, which supports this extension from version 1.0.2 [80]. In case of Java Programming Language, TLS/ALPN is supported from 11th edition (including LTS releases only). Implementing the proposed solution in earlier versions requires overriding all the classes related SSLContext and SSLEngine and manually handling the ALPN extension. The HTTP/1.1 Upgrade procedure is much more complicated since it needs to be implemented in the application level (rather than transport level). Adding a mechanism for multiple HTTP round trips and switching to a custom, binary protocol is often complicated, therefore it does not fulfill the "Simple client implementation" requirement.

```
1   Input:  Inbound connection ic
2           Routing table rt
3           UpgradeHandler uh
4           CommunicationPipeline cp
5   Output: SinglePortUpgradeHandler rd
6
7   Protocol p = uh.negotiate(ic)
8   Handler h = rt.getHandler(p)
9   if (h == null)
10    h = rt.getHandler("HTTP/1.1")
11  cp.addHandler(h)
```

Figure 6.4: Protocol switching implementation pseudo-code

Based on the literature overview from Section 3.3 as well as the growing popularity of inter-connected clouds (mentioned in Section 1.3), inventing a solution for switching to custom, binary protocols that could be used free of charge in a container-based cloud is highly desirable. Such a solution will offer the best performance without forcing users to use the HTTP protocols.

The prototype has been implemented using Infinispan Open Source project. Infinispan is an in-memory data store that offers many endpoint protocol implementations, including HTTP/1.1, HTTP/2, Hotrod, Memcached and Web Sockets. For the proof of concept implementation protocols were chosen - HTTP/1.1, HTTP/2 and Hotrod. Both the server and the client implementation has been done using Netty framework [73] and can be found in Infinispan ticket system as well as on Github project [1]. The algorithm is based on multi-tenancy router implementation [97], which has been modified to perform both TLS/ALPN negotiation and HTTP/1.1 Upgrade procedure. The router implementation manages the communication pipeline (Netty communication is based on Events, which are handled by Handlers within the communication pipeline; It is a "chain of responsibility" design pattern) as well as a full list of supported protocols by the server (which is managed in a routing table). The router requires at least one REST server implementation to be added, because HTTP/1.1 is used as a fallback protocol. A simplified algorithm implementation has been shown in the Figure 6.4.

Lines from #1 to #4 represent input arguments for the proposed algorithm. Since the implementation is based on multi-tenancy feature, they are similar to the ones introduced in Chapter 4. Both TLS/ALPN and HTTP/1.1 Upgrade logic were extracted into `UpgradeHandler` represented by `uh` variable. The mechanism negotiates target communication protocol proposed by a client application (Line #7) and applies it to the communication pipeline (Line #11). If none of the

[1] The ticket can be accessed using the following URL https://issues.jboss.org/browse/ISPN-8756

communication protocols were chosen, the algorithm falls back to HTTP/1.1 (Line #9 and Line #10).

The algorithm's complexity is proportional to the number of routes in the routing table. Using a Big-O notation it might be written as $T(n) = O(n)$.

6.3 Experiment environment description and tools used for the evaluation

The experiments were performed in two stages - the initial performance tests were executed on a local laptop and the validation tests were performed on OpenShift 3.11 cloud.

The initial tests were performed on Intel(R) Core(TM) i7-7600U CPU, 16 GB of RAM laptop using Linux OS (Fedora F25). All testing scenarios were written in Java using JMH [77] tool (a micro-benchmark framework).

The validation tests were performed using OpenShift 3.11 cloud deployed on AWS. The system consisted of 3 Master nodes and 2 Worker nodes. Such a system was initialized solely for running the switching protocol test. Most of the background tasks were turned off to minimize interference with testing harness.

6.4 Experiments results

All the tests have been done using Java Hotrod client (an Infinispan client library) and a custom implementation of the HTTP/1.1 and HTTP/2 clients (with both TLS/ALPN as well as HTTP/1.1 Upgrade). In all the tests there has been a singleton Infinispan Server instance deployed in the cloud and testing harness communicating with the server using different clients and communication paths. Tested configurations include direct communication without proposed solution (scenarios, where ”Negotiation mechanism” is “None“) and with it (other scenarios). The proposed solution was tested in multiple different configurations, including:

- Using TLS/ALPN negotiation mechanism and switching to HTTP/2 or Hotrod protocol
- Using HTTP/1.1 Upgrade procedure and switching to HTTP/2 or Hotrod protocol

Additionally, all scenarios were validated using a Router component provided by a cloud vendor (HAProxy) and using direct communication inside of the cloud. The aim of the experiments is to measure the performance overhead of each switching mechanism individually (”Connection type“ set to ”Direct“

Negotiation mechanism	Connection type	Target protocol	Iterations	result [ms/op]	± Error
None	Direct	HTTP/1.1	31	2.068	0.686
None	OCP Router	HTTP/1.1	31	1.087	0.154
TLS/ALPN	Direct	HTTP/2	31	5.063	0.531
TLS/ALPN	OCP Router	HTTP/2	31	6.576	1.535
HTTP/1.1 Upgrade	Direct	HTTP/2	31	3.310	0.864
HTTP/1.1 Upgrade	OCP Router	HTTP/2	31	4.464	1.617
TLS/ALPN	Direct	Hotrod	31	9.742	1.102
TLS/ALPN	OCP Router	Hotrod	31	10.401	1.302
HTTP/1.1 Upgrade	Direct	Hotrod	31	5.389	1.067
HTTP/1.1 Upgrade	OCP Router	Hotrod	31	8.594	10.122

Table 6.3: Initialize connection results

Negotiation mechanism	Connection type	Target protocol	Iterations	result [ms/op]	± Error
None	Direct	HTTP/1.1	31	0.472	0.330
None	OCP Router	HTTP/1.1	31	1.315	0.781
TLS/ALPN	Direct	HTTP/2	31	1.577	0.282
TLS/ALPN	OCP Router	HTTP/2	31	2.149	0.480
HTTP/1.1 Upgrade	Direct	HTTP/2	31	1.048	0.078
HTTP/1.1 Upgrade	OCP Router	HTTP/2	31	1.156	0.078
TLS/ALPN	Direct	Hotrod	31	0.269	0.039
TLS/ALPN	OCP Router	Hotrod	31	0.331	0.048
HTTP/1.1 Upgrade	Direct	Hotrod	31	0.193	0.037
HTTP/1.1 Upgrade	OCP Router	Hotrod	31	0.255	0.051

Table 6.4: Uploading 360 bytes to the server results.

- this excludes passing network traffic though a Reverse Proxy and makes the measurement results more stable). The next goal is to prove that is it possible to use all switching mechanisms provided by the proposed solution with scenarios involving Reverse Proxy (”Connection type“ set to ”OCP Router“). The final goal is to measure, which switching mechanism is the most effective and introduces the lest possible performance overhead.

The Table6.3 contains performance results for initiating connection and switching to different protocols.

Table 6.4 and Table-6.5 contain performance results for sending 360 and 36 byte entries (as a key/value pairs) into the server. The results have been visualized in Figure 6.4, Figure 6.6 and Figure 6.7. All the diagrams with the results use similar notation, the vertical axis represents time in milliseconds and each bar represents a particular solution.

Tables 6.7, 6.8, 6.9, 6.10, 6.12 and 6.11 contain calculated significance measure between each tested pair of benchmarks based on the procedure described in Section 3.8. For $\alpha = 0.05$ (99%), the tested interval is equal to $(-\infty, -2.58 > \cup < 2.58, \infty)$. Benchmarks that are not statistically comparable were gathered in Table 6.6.

Negotiation mechanism	Connection type	Target protocol	Iterations	result [ms/op]	± Error
None	Direct	HTTP/1.1	31	0.426	0.337
None	OCP Router	HTTP/1.1	31	2.310	0.761
TLS/ALPN	Direct	HTTP/2	31	0.754	0.196
TLS/ALPN	OCP Router	HTTP/2	31	0.649	0.100
HTTP/1.1 Upgrade	Direct	HTTP/2	31	0.267	0.062
HTTP/1.1 Upgrade	OCP Router	HTTP/2	31	0.305	0.055
TLS/ALPN	Direct	Hotrod	31	0.267	0.062
TLS/ALPN	OCP Router	Hotrod	31	0.302	0.052
HTTP/1.1 Upgrade	Direct	Hotrod	31	0.231	0.065
HTTP/1.1 Upgrade	OCP Router	Hotrod	31	0.243	0.040

Table 6.5: Uploading 36 bytes to the server results.

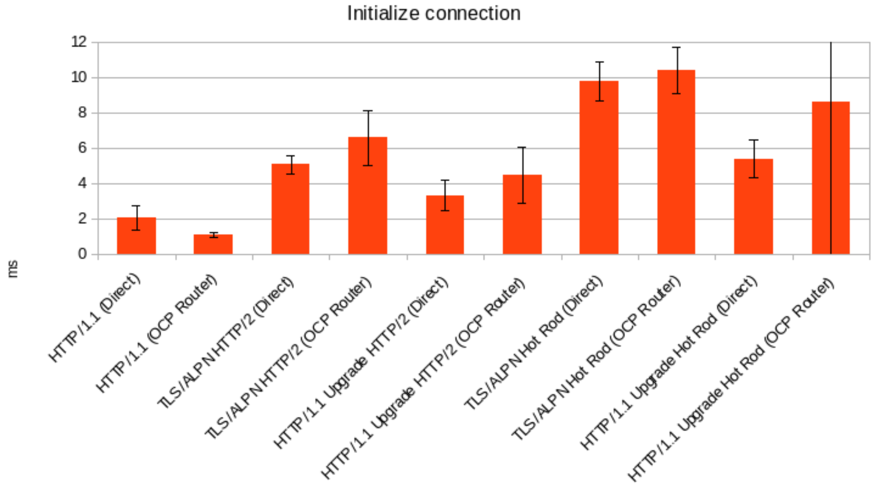


Figure 6.5: Initialize connection results.

Test type	Mode	Benchmark 1	Benchmark 2
Initializing connections	Direct	TLS/ALPN (HTTP/2)	HTTP/1.1 Upgrade (Hotrod)
Initializing connections	OCP Router	TLS/ALPN (HTTP/2)	HTTP/1.1 Upgrade (Hotrod)
Initializing connections	OCP Router	HTTP/1.1 Upgrade (HTTP/2)	HTTP/1.1 Upgrade (Hotrod)
Initializing connections	OCP Router	HTTP/1.1 Upgrade (Hotrod)	None
Uploading 360 bytes	Direct	HTTP/1.1 Upgrade (Hotrod)	None
Uploading 360 bytes	OCP Router	TLS/ALPN (HTTP/2)	None
Uploading 36 bytes	Direct	HTTP/1.1 Upgrade (HTTP/2)	TLS/ALPN (Hotrod)
Uploading 36 bytes	OCP Router	TLS/ALPN (HTTP/2)	None
Uploading 36 bytes	OCP Router	HTTP/1.1 Upgrade (HTTP/2)	TLS/ALPN (Hotrod)
Uploading 36 bytes	OCP Router	HTTP/1.1 Upgrade (HTTP/2)	None
Uploading 36 bytes	OCP Router	TLS/ALPN (Hotrod)	None
Uploading 36 bytes	OCP Router	HTTP/1.1 Upgrade (Hotrod)	None

Table 6.6: Statistically insignificant benchmark comparisons

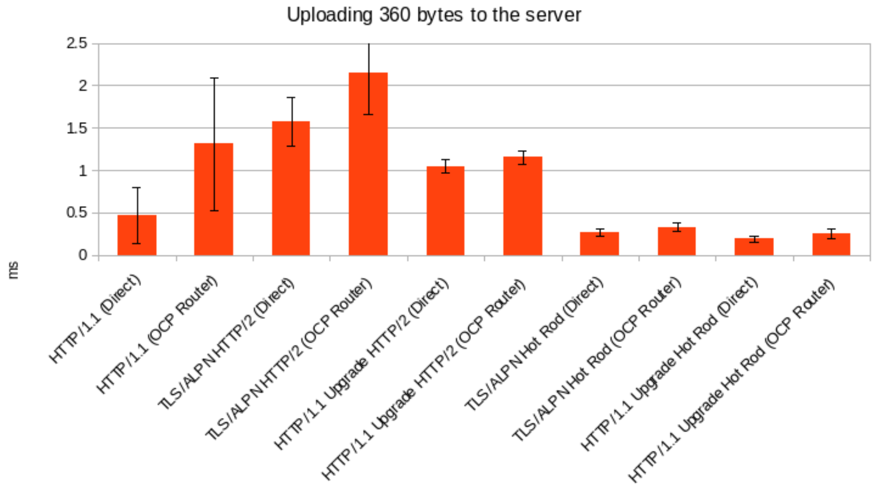


Figure 6.6: Uploading 360 bytes to the server results.

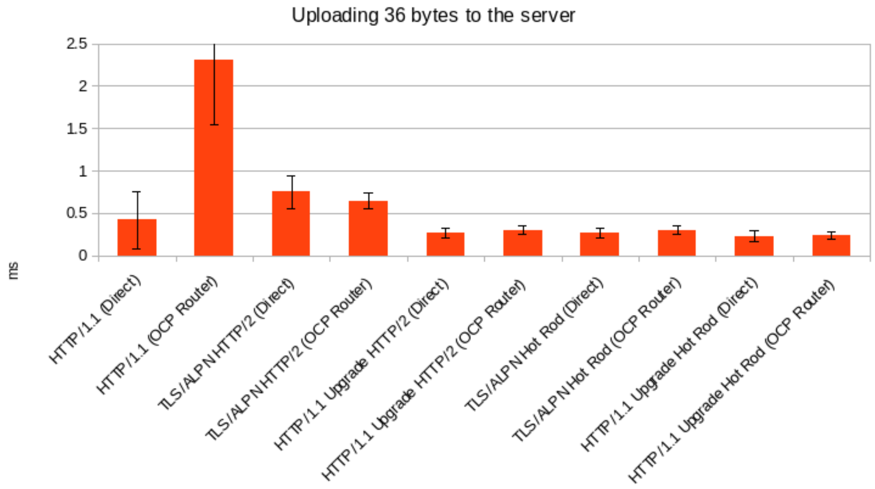


Figure 6.7: Uploading 36 bytes to the server results.

	TLS/ALPN	HTTP/1.1	TLS/ALPN	HTTP/1.1
	(HTTP/2)	Upgrade	(Hotrod)	Upgrade
		(HTTP/2)		(Hotrod)
HTTP/1.1 Upgrade (HTTP/2)	4.45			
TLS/ALPN (Hotrod)	-9.85	-11.83		
HTTP/1.1 Upgrade (Hotrod)	-0.70	-3.90	7.31	
None	8.89	2.90	15.23	6.74

Table 6.7: Significance measure matrix between benchmark results for initializing connection in direct mode

	TLS/ALPN	HTTP/1.1	TLS/ALPN	HTTP/1.1
	(HTTP/2)	Upgrade	(Hotrod)	Upgrade
		(HTTP/2)		(Hotrod)
HTTP/1.1 Upgrade (HTTP/2)	2.44			
TLS/ALPN (Hotrod)	-4.89	-7.37		
HTTP/1.1 Upgrade (Hotrod)	-0.50	-1.03	7.31	
None	9.16	2.90	5.35	1.91

Table 6.8: Significance measure matrix between benchmark results for initializing connection in OCP router mode

	TLS/ALPN	HTTP/1.1	TLS/ALPN	HTTP/1.1
	(HTTP/2)	Upgrade	(Hotrod)	Upgrade
		(HTTP/2)		(Hotrod)
HTTP/1.1 Upgrade (HTTP/2)	4.66			
TLS/ALPN (Hotrod)	11.83	23.01		
HTTP/1.1 Upgrade (Hotrod)	12.53	25.51	7.31	
None	6.65	4.37	3.64	-2.16

Table 6.9: Significance measure matrix between benchmark results for uploading 360 bytes to the server in direct mode

	TLS/ALPN	HTTP/1.1	TLS/ALPN	HTTP/1.1
	(HTTP/2)	Upgrade	(Hotrod)	Upgrade
		(HTTP/2)		(Hotrod)
HTTP/1.1 Upgrade (HTTP/2)	5.26			
TLS/ALPN (Hotrod)	9.70	23.20		
HTTP/1.1 Upgrade (Hotrod)	10.11	25.51	24.90	
None	2.34	-0.52	3.64	-3.49

Table 6.10: Significance measure matrix between benchmark results for uploading 360 bytes to the server in OCP router mode

	TLS/ALPN (HTTP/2)	HTTP/1.1 Upgrade (HTTP/2)	TLS/ALPN (Hotrod)	HTTP/1.1 Upgrade (Hotrod)
HTTP/1.1 Upgrade (HTTP/2)	7.76			
TLS/ALPN (Hotrod)	7.93	0.10		
HTTP/1.1 Upgrade (Hotrod)	9.71	25.51	32.34	
None	-5.57	-6.80	3.64	-6.99

Table 6.11: Significance measure matrix between benchmark results for uploading 36 bytes to the server in OCP router mode

	TLS/ALPN (HTTP/2)	HTTP/1.1 Upgrade (HTTP/2)	TLS/ALPN (Hotrod)	HTTP/1.1 Upgrade (Hotrod)
HTTP/1.1 Upgrade (HTTP/2)	6.10			
TLS/ALPN (Hotrod)	6.10	0		
HTTP/1.1 Upgrade (Hotrod)	6.52	25.51	25.5	
None	2.16	-1.19	-1.19	-1.46

Table 6.12: Significance measure matrix between benchmark results for uploading 36 bytes to the server in direct mode

6.5 Results analysis

Selecting network protocol for client/server communication in the cloud is not a trivial topic. Each application and each development team has its own requirements for the technology used in the project. Some of the commonly used requirements have been gathered in Table-6.2. Most of the languages and out of the box libraries provide support for the HTTP/1.1 protocol (some of them also offer seamless upgrade to HTTP/2 but at the moment of writing the prototype, most of the Java clients lack support for it), which was one of the major contributing factor for its popularity among application and library developers. However, in some scenarios, simplicity is not the deciding factor. Performance or asynchronous processing allows to increase system performance and lower operating costs at the same time (very often those two things are tied together - better performance allows to decrease the number of application replicas and lower the overall operational costs). In order to reduce costs even further, it is highly advisable to use a publicly available cloud router instead of a dedicated, often quite expensive, application Load Balancer. Such routers are often designed with HTTP/1.1 and TLS in mind. Supporting custom protocols can be achieved by encrypting the traffic or using HTTP/1.1 Upgrade procedure.

Table-6.3 and Figure 6.4 present the benchmark results for the time necessary for initiating connection to an in-memory data store - Infinispan. The first two rows (no switching mechanism and HTTP/1.1 used as a communication protocol) have been used as a baseline. Introducing a public router between the testing harness and the server (connection type equal to "OCP Router") made the connection initialization roughly 2 ms slower than without it. Each pair of results presents a similar pattern - using the HTTP/1.1 Upgrade procedure is slightly faster than TLS/ALPN. The worst result was observed for the Hotrod Protocol (in TLS/ALPN mode), which performs its own handshake procedure and exchanges server information during connection initialization. As the results show, exchanging this information takes quite a long time (comparing to the other measured protocols).

The next important benchmark is inserting 36 and 360-byte keys and values into the in-memory store. As expected, in all the cases using unencrypted connections was faster than the connections with TLS. The difference gets bigger with larger payloads. This is also expected, since encrypted payloads are larger than the unencrypted ones. The custom protocol used for testing performed more than twice faster than HTTP/1.1. HTTP/2 performance was slightly lower than Hotrod's but it was still faster than HTTP/1.1. At this point, it is also worth mentioning that the testing procedure was synchronous (the testing harness sent a request to the server and waited for response). Both HTTP/2 and Hotrod protocols support asynchronous processing and using it would make the performance gap even larger.

The results show that using a custom, binary protocol is the best option to achieve the high performance results. The proposed solution enables switching mechanism for both encrypted and unencrypted scenarios. Without it, using a custom, binary protocols would require either allocating an expensive Load Balancer or encrypting the whole traffic (even when it is not necessary). Benchmark results also indicate that a small slowdown during connection initialization and switching to a custom binary protocol makes a large difference in the overall data transfer performance. It is also worth mentioning that some deployments do not allow introducing a data grid client dependency to the application. In such cases, switching to HTTP/2, which is often available out of the box in many languages is highly advisable. The HTTP/2 protocol is considered as a middle ground between HTTP/1.1 text protocol and a custom, binary protocol. The Table-6.5 shows some of the use cases with suggested protocols and switching mechanisms. The table might be used as a guide for introducing specific solutions into new or existing projects.

6.6 Limitations

TLS/ALPN RFC [92] allows negotiating communication protocol only during the handshake procedure. Once the handshake finishes, there is no other way to change the protocol other than reconnecting. HTTP/1.1 Upgrade procedure

Use case	Key deciding Factors	Protocol Switch Mechanism	Target Protocol
Transmitting sensitive data at high speed	Performance Encryption	TLS/ALPN	Custom, binary protocol
Transmitting public at high speed	Performance	HTTP/1.1 Upgrade	Custom, binary protocol
Occasionally send short heartbeats	Fast connection initialization	HTTP/1.1	HTTP/1.1
Transmitting sensitive data without new dependencies	No new dependencies Encryption	TLS/ALPN	HTTP/2

Table 6.13: Use cases and recommendations.

[92] is more flexible in this matter - it allows initiating the switching procedure anytime by either the client or the server. In order to support both options, the implementation needs to follow the most limited switching mechanism - TLS/ALPN and negotiate communication protocol only during the handshake. For the same reason, it is also not possible to switch from a custom binary protocol back to HTTP/2 or HTTP/1.1.

Another practical limitation is focusing solely on TCP transport with both routing with a Reverse Proxy (a solution with a router) and client-server communication. Using HTTP over UDP is also not a very common scenario (used mainly for streaming data). From the in-memory store's perspective, maintaining a stateful connection between a client and a server has a lot of benefits from the implementation's perspective (e.g. sending notifications from the server to the client about a finished transaction - a paradigm often used in asynchronous or reactive programming model). Recent work on QUIC [48] (a multiplexted transport protocol over UDP) and HTTP/3 [14] (HTTP over QUIC transport) may result in better performance even compared to binary protocols. If so, many projects (including reverse proxies and in-memory data stores) will require many implementation changes.

6.7 Further work

Both research and implementation have been done using the Infinispan Open Source project. At the moment of writing this thesis, the server implementation has already been merged into the project, whereas the client part is still waiting to be reviewed by the Infinispan Team. In the near future this work will be extended to other server protocols such as Web Sockets or Memcached, and the results will be compared. Another goal is to create a multi-protocol client implementation which can switch communication protocols on demand. Such an implementation allows to address very interesting use cases that require different connection characteristics to perform different tasks. An example of such behavior is transmitting a large amount of encrypted traffic along with small portions of public data. In that case, the client could use a TLS/ALPN encrypted Hotrod connection along with HTTP/1.1 protocol.

Chapter 7

Proposed solution for automatic detection of application misconfiguration

7.1 Introduction

Recent reports show that application configuration, especially with the security and performance aspects in mind, is one of the most important challenges in cloud computing [105]. Configuration tuning of a large production system requires deep knowledge of each individual project used in all the deployed components. A recent survey mentioned over 30% of engagement requests for a large Posgresql service company were for solving configuration issues and perform tuning [13]. A much higher percentage might be observed on community project mailing lists or users forums, where Keycloak for example receives more than 20 queries for configuration advice a month (Keycloak is a security-related project that uses Infinispan very heavily) [69].

Helping users spot common configuration mistakes is not only a helpful idea but it solves real problems by lowering the overall number of support cases and allowing experienced support engineers to focus on complicated problems. At the same time, this approach helps in tuning performance of a running system and oftentimes makes the configuration more secure.

7.1.1 Machine Learning techniques for classification problems

Among other use cases (described in Section 3.4), Machine Learning turns out to be very helpful when classifying a data point (also called an instance) to a certain category (also called class).

		Predicted	
		Positive	Negative
Actual	Positive	True Positives (TP)	False Negatives (FN)
	Negative	False Positives (FP)	True Negatives (TN)

Table 7.1: Model accuracy measures for classification

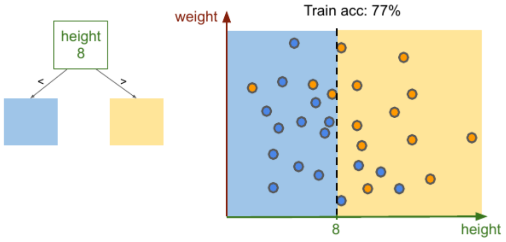


Figure 7.1: Decision Tree (1-level deep) [67]

Every classifier (a model capable of classifying a data point into a class) can be accessed in terms of accuracy. Table 7.1 shows differences between commonly used concepts of True/False Positives and True/False Negatives. Those concepts are used for accessing model's Accuracy and F1 metric described by Equation 7.1 and Equation 7.2.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{7.1}$$

$$F1 = \frac{2 * TP}{2 * TP + FP + FN} \tag{7.2}$$

Section 3.4 describes most commonly used techniques for classification, including Logistics Regression or DNF-based approaches, many recent Machine Learning competitions [118] have been won using Decision Trees (with Gradient Boosting technique [124]). Tree-based methods segment the predictor space into smaller regions using a method that allows to express prediction rules in a tree data structure. The decision-making process can be divided into steps. Each step makes the predictor space smaller. This process has been shown in Figure 7.1 and Figure 7.2.

Classifying data into specific segment can be written formally by using $R_1 = \{X|height > 8\}$ notation, however, a diagram is much more descriptive.

Training decision trees adjusts regions $R_1, ..., R_J$ to minimize RSS given by the formula in Equation 7.3. Oftentimes, this split is done using binary splitting and compensated by using other methods. A general algorithm for building a decision tree has been described in Listing-7.3.

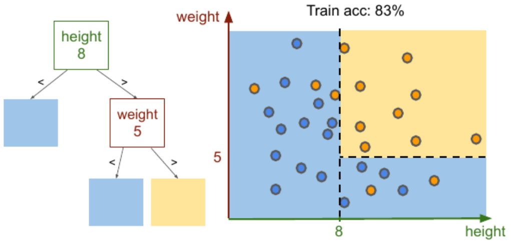


Figure 7.2: Decision Tree (2-level deep) [67]

```
1. Use recursive binary splitting to grow a large tree on the training
   data, stopping only when each terminal node has fewer than some
   minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain
   a sequence of best subtrees, as a function of $\alpha$
3. Use $K$-fold cross-validation to choose $\alpha$. That is,
   divide the training o bservations into $K$ folds.
   For each $k = 1, ..., K$ :

    (a) Repeat Steps 1 and 2 on all but the $k$th fold of
        the training data.
    (b) Evaluate the mean squared prediction error on the data
        in the left-out $k$th fold, as a function of $\alpha$.

   Average the results for each value of $\alpha$, and pick
   $\alpha$ to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the
   chosen valueof $\alpha$.
```

Figure 7.3: Building a decision tree algorithm [33]

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y_{R_j}})^2 \tag{7.3}$$

Building a classification tree is very similar to building a regression tree. The only difference is using a qualitative rather than quantitative output. This implies that RSS cannot be used for making the split decisions while building the tree. In practice, there are two measures very frequently used for growing a tree - Gini index (Equation 7.4) and Entropy [1] (Equation 7.5). Here $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class.

[1] Entropy can also be perceived (from practical perspective) as a measure of chaos in the system

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{7.4}$$

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} log - \hat{p}_{mk} \tag{7.5}$$

Reducing the amount of entropy is formally called Information Gain and can be calculated using Equation 7.6. Parameter Q describes information gain by making a specific split in the tree, k is the number of groups after the split, N_i is number of objects from the sample in which variable Q is equal to the i-th value [117].

$$IG(Q) = D_0 - \sum_{k=1}^{K} \frac{N_i}{N} D_i \tag{7.6}$$

There are many algorithms describing how to build the decision tree. The simplest ones are maximizing Information Gain performed by the splits (depending on one or many factors).

From practical perspective, decision trees have the following properties:

1. Data does not require normalization
2. Can easily be explained to people using a graph
3. Cannot predict values outside of the training data range

7.1.2 Operator Framework

Modern container-based clouds, such as Kubernetes or OpenShift, use Event Driven Architecture in its controller components (more on container-based cloud architecture might be found i Section 1.2). This approach allows decoupling an event store from the controller code that reacts on specific events. A typical Kubernetes Master Node runs an API Server connected to the data store - Ectd (a key-value based data store). All other components of the system are allowed to communicate only with the API Server. Kubernetes uses Controllers to react on changing state of the objects in the API Server. Typically, a notification mechanism (called "Watch") is used for it.

From the implementation perspective, Operators [81] allow developers to create their own data type, that will be available through the API Server (and stored in the data store) and write their own Controllers. The most basic implementation leverages Watch mechanism and reacts on changes to a custom resource.

```
The goal seeking behavior of the control loop is very stable.
This has been proven in Kubernetes where we have had bugs that
have gone unnoticed because the control loop is fundamentally
stable and will correct itself over time.

If you are edge triggered you run risk of compromising your
state and never being able to re-create the state. If you are
level triggered the pattern is very forgiving, and allows
room for components not behaving as they should to be rectified.
This is what makes Kubernetes work so well.
```

Figure 7.4: Explanation of level-based and edge-based events by Joe Beda, CTO of Heptio [47]

The authors of the Operator SDK made an interesting design decision - events are triggered as level-based, opposed to edge-based (more explanation might be found in Listing 7.4). This means, an event is not propagated to a Controller. A Controller needs to re-considerate the whole current system state and react to it [28]. This approach is also referred to as Reconciler Pattern [47]. Fundamentally, the Reconciler Pattern is very simple and can be divided into three steps:

1. Get current system state
2. Get desired system state
3. Perform a set of actions, so that the current system state matches the desired system state

From the conceptual perspective, Operators allow software vendors to automate management of their software in a cloud. Typical management operations involve:

- Automatic installation of the software
- Configuration management
- Automatic software upgrades (with no downtime if possible)
- Creating automatic backups and restoring them if necessary
- Integrating with monitoring and log collection stacks
- Horizontal and vertical auto-scaling
- Abnormal behavior detection

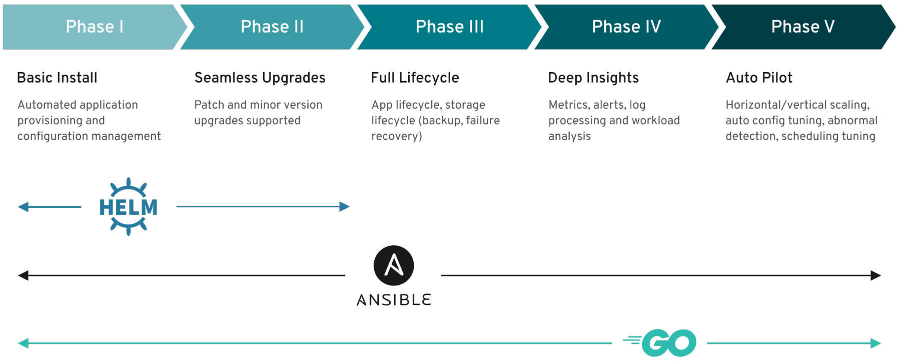

Phase I
Phase II
Phase III
Phase IV
Phase V
Basic Install
Automated application provisioning and configuration management
Seamless Upgrades
Patch and minor version upgrades supported
Full Lifecycle
App lifecycle, storage lifecycle (backup, failure recovery)
Deep Insights
Metrics, alerts, log processing and workload analysis
Auto Pilot
Horizontal/vertical scaling, auto config tuning, abnormal detection, scheduling tuning
HELM
ANSIBLE
GO

Figure 7.5: Operator Capability Model [109]

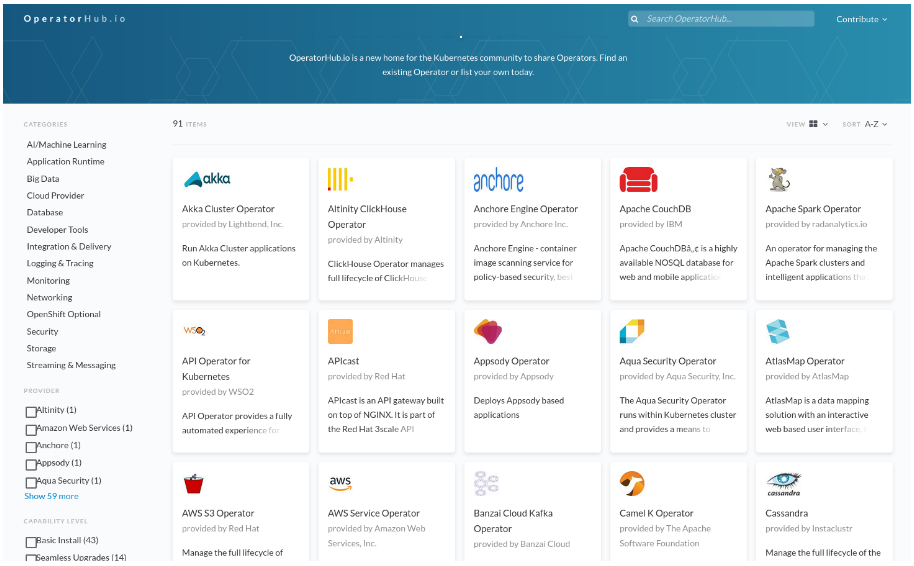
Depending on the provided capabilities, Operators are divided into ”Phases“ (Figure 7.5). The widest support is available when an Operator is designed using Go Language (the same that has been used to create both Kubernetes and OpenShift).

Operators created by different communities are gathered in a single place, called OperatorHub. The service has been launched by Red Hat in conjunction with Amazon, Microsoft, and Google. At the moment of writing this paragraph, there are 91 Operators that can manage different kind of software in a Kubernetes (or OpenShift) cluster - starting from Data bases (such as Postgresql) or providing more sophisticated application, such as Keycloak - OpenID Connect Server [2]. Current web site of the OperatorHub service has been presented in Figure 7.6.

7.1.3 NoOps initiative

Social Media focused on IT industry built large interest in no-operations (NoOps) initiative. This has been summarized in an article written by Adrian Cockcroft [7]. The term describes the way developers have worked at Netflix (even though the article describes the status of 2012, it is still up to date). In 2007, Netflix streaming services were still experimental. As many traditional companies, Netflix had an operations team responsible for keeping data center infrastructure running. Back then, the team was constantly overwhelmed by the day to day duties. Netflix as one of the first companies decided to move their entire infrastructure to Amazon AWS cloud and delegated most of the large-scale problems to the Amazon's cloud

[2] The major part of Keycloak Operator has been implemented by the author of this thesis - https://github.com/keycloak/keycloak-operator



OperatorHub.io

Search OperatorHub...

Contribute

OperatorHub.io is a new home for the Kubernetes community to share Operators. Find an existing Operator or list your own today.

CATEGORIES

AI/Machine Learning
Application Runtime
Big Data
Cloud Provider
Database
Developer Tools
Integration & Delivery
Logging & Tracing
Monitoring
Networking
OpenShift Optional
Security
Storage
Streaming & Messaging

PROVIDER

□ Altinity (1)
□ Amazon Web Services (1)
□ Anchore (1)
□ Appsody (1)
□ Aqua Security (1)
Show 59 more

CAPABILITY LEVEL

□ Basic Install (43)
□ Seamless Upgrades (14)

91 ITEMS

VIEW ▦ SORT A-Z

Akka Cluster Operator
provided by Lightbend, Inc.
Run Akka Cluster applications on Kubernetes.

Altinity ClickHouse Operator
provided by Altinity
ClickHouse Operator manages full lifecycle of ClickHouse

Anchore Engine Operator
provided by Anchore Inc.
Anchore Engine - container image scanning service for policy-based security, best-

Apache CouchDB
provided by IBM
Apache CouchDBâ„¢ is a highly available NOSQL database for web and mobile applicatio

Apache Spark Operator
provided by radanalytics.io
An operator for managing the Apache Spark clusters and intelligent applications tha

API Operator for Kubernetes
provided by WSO2
API Operator provides a fully automated experience for

APIcast
provided by Red Hat
APIcast is an API gateway built on top of NGINX. It is part of the Red Hat 3scale API

Appsody Operator
provided by Appsody
Deploys Appsody based applications

Aqua Security Operator
provided by Aqua Security, Inc.
The Aqua Security Operator runs within Kubernetes cluster and provides a means to

AtlasMap Operator
provided by AtlasMap
AtlasMap is a data mapping solution with an interactive web based user interface, t

AWS S3 Operator
provided by Red Hat
Manage the full lifecycle of

AWS Service Operator
provided by Amazon Web Services, Inc.

Banzai Cloud Kafka Operator
provided by Banzai Cloud

Camel K Operator
provided by The Apache Software Foundation

Cassandra
provided by Instaclustr
Manage the full lifecycle of the

Figure 7.6: Front page of the OperatorHub service

team (which has dedicated people and much more knowledge on how to keep things running at scale). This allowed Netflix to remove most of the operations team and delegate the day to day duties to the development teams.

The NoOps initiative has a lot in common with the Operator approach; however, the Operators take it one step further and automate most of the day to day administration and operation duties. Furthermore, abnormal behavior detection or automatic auto-scaling allows infrastructure to adjust to changing conditions [63].

7.1.4 Modern expert and recommendation systems implementations

Based on literature overview (from Section 3.6) and practical experience, expert systems require a problem description, that may consists of many different artifacts and an evaluation engine to produce an output. Depending on the knowledge base system design, such a problem description may consists of (but is not limited to):

- Problem description
- Runtime environment
- Different type of constraints

The most typical approach to expert system evaluation engine is to use a Rules Engine (such as Drools project [26]). A typical rule may be represented as an ”if

- then“ statement. Knowledge Base System typically contains hundreds (or even thousands) of such rules. Another approach is to use Statistical and Machine Learning. Finally, there are prototypes that leverage both these approaches [88]. The former approach (rule-based) represents more practical approach to the problem. In most of the cases, it is possible to adjust knowledge base rules to match a particular problem. The latter approach (statistical/Machine Learning) is by far more interesting.

In 1989 researches [125] noticed the potential of using Artificial Intelligence in expert system design. Their findings focused on learning from examples (referred to as Learning From Multiple Examples - LFME). The authors created two datasets with positive and negative examples, where negative ones were used to exclude given hypothesis from consideration. The next step in the process involved the conceptual clustering approach where the system aggregated a set of instances into classes of instances. Today, both techniques are referred to as Supervised and Unsupervised training. Conceptually, this research is very similar to the one presented in Section 3.7.

A hybrid approach (rule-based along with Machine Learning) might be observed in a similar category of systems - recommender systems (or recommendation engines). Recommender systems are the software tools used to generate and provide suggestions for items and other entities to the users by exploiting various strategies [27]. Typically, all recommender systems employ one or more recommendation strategies:

- Collaborative Filtering - people who had similar tastes in the past will also have similar tastes in the future
- Content-Based Filtering - people who liked items with certain attributes in the past, will like the same kind of items in the future as well
- Demographic Filtering - uses demographic data such as age, gender, education, etc.
- Knowledge-Based Filtering - users and items to reason about what items meet the users’ requirements, and generate recommendations accordingly

It goes without saying, that Machine Learning techniques fit perfectly into most of the filtering categories.

7.1.5 Available metrics for prototype evaluation

The prototype evaluation is based on a small portion of metrics available through Infinispan metrics endpoint along with extracted parameters from configuration. Table 7.2 shows all the metrics and configuration values used for the prototype. A full list of metrics might be found the Appendix (Section 10.1).

Metric name	Metric value	Metric type	Description
up	1	Runtime	1 if the service is running, 0 otherwise
jgroups_kube_ping_count	0	Configuration	The number of discovered KUBE_PING discovery protocols
jgroups_dns_ping_count	0	Configuration	The number of discovered DNS_PING discovery protocols
cluster_lock_timeout	60000	Configuration	Cluster lock timeout [ms]
jvm_threads_current	170	Metric	The number of threads used by Infinispan Server

Table 7.2: Metrics used for prototype evaluation

7.2 Proposed solution for automatic detection of application misconfiguration

The major contribution of this proposal design is a solution for automatic detection of application misconfiguration in a container-based cloud environment. This solution has been tested using a dedicated testing environment and the results are satisfactory

Many researchers and commercial companies, including Magalix [3] or Red Hat Artificial Intelligence Center of Excellence [4], have been working on optimizing workload in the cloud. Most of the work is focused on optimizing cloud infrastructure and cloud components for performance (or the lowest latency) but none of them deals with application-level configuration. There are many technical difficulties to do that, including:

- Different configuration formats
- Lack of training environment
- Lack of a standard infrastructure gathering metrics
- Different deployment scenarios in companies

Operator initiative improves this situation a lot. Custom resources and unified API Server access in Kubernetes create a nice and clean interface for configuring applications hosted in the cloud. On the one hand, vendors allow certain level of configuration to be set in provided Custom Resource Definitions [21], on the other hand, there is a standard way of modifying them (using Kubernetes API Server REST interface). Operators also have access to the application log and metric collection infrastructure hosted in the cloud. There are ways to limit this

[3] https://www.magalix.com/

[4] https://github.com/AICoE

Solution	Priority
Discover common configuration mistakes	Must-have
Solutions are easy to explain	Must-have
Simple deployment	Good-to-have

Table 7.3: Requirements for proposed solution

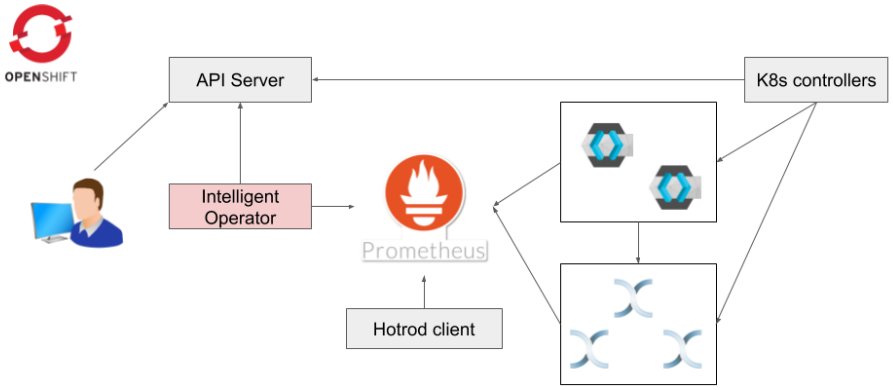


Figure 7.7: Expert system high-level design

access but, from practical reasons, it is a very uncommon thing to do. Typically, the metrics are collected using Prometheus project [85] and rendered to the users using dashboards. An Operator can also execute a query against Prometheus and gather all the collected metrics if necessary.

Apart from supporting this thesis aims, a solution needs to meet secondary goals, that have been gathered in Table 7.3. The most important goal is to discover common configuration mistakes and suggest the actions to fix them. It is also important that the decision process could be explained in a graphical form, since in most of the cases, the action reports will be read by well trained specialists team, such as project maintainers or Red Hat Support Team. The secondary goals require the project to be deployed in a simple way.

A high-level design for the proposed solution has been presented in Figure 7.7. An Intelligent Operator reads application runtime metrics from Prometheus and gathers application configuration from a Custom Resource Definition exposed by the API Server. It it worth mentioning that this proposal does not modify any Custom Resources. This responsibility lies on the human operator's side. Once a configuration is changed, Kubernetes Controllers (implemented in the Application Operator) react to the change and modify an application. Each application instance (represented by Keycloak or Infinispan logo in the diagram) reports its metrics back to Prometheus.

up	jgroups_kube_ping_count	ACTION
1	0	There is no KUBE_PING configuration and yet, we are running in Kubernetes

Table 7.4: Knowledge Base for detecting misconfigured discovery protocol

Clustering is one of the most frequently mentioned configuration issue among both the community project maintainers and Red Hat Support Team members. Both teams are overwhelmed by requests from clients seeking help with fixing common configuration mistakes, such as member discovery protocol configuration. Customers try to use protocols designed for the in-house data centers, such as TCPPING or MPING (both are part of JGroups project) instead of using KUBE_PING or DNS_PING. This configuration mistake can easily be spotted by using Machine Learning and inspecting application configuration.

Intelligent Operator uses a Knowledge Base (written as a spreadsheet) with wrong configuration examples (similarly to [125]) that is typically maintained by project community members and Red Hat Support Team. Columns in the spreadsheet represent metrics obtained from Prometheus as well as the parsed configuration properties. The last column is the recommended action, where the experts suggest how to fix the problem. A very simplified example of a Knowledge Base spreadsheet has been presented in Table 7.4. The example indicates that the application is running (”up“ metric is equal to 1) and there is no KUBE_PING present in the configuration (”jgroups_kube_ping_count“ is 0).

Finding a proper action depending on metrics is a typical classification problem. Instead of a class, a classifier needs to find the proper action. Assuming a Knowledge Base spreadsheet might be treated as training data, a classifier will only be interested in discovering actions (classes) it has seen before (as mentioned in Section 7.1.1, this is a limitation of tree-based classifiers). Having this requirements in mind, a simple decision tree classifier is perfect for this solution.

Intelligent Operator’s main control loop is triggered using both the timer and the watch of a custom resource. Every run is being called an Observation Period and consists of several steps (shown in Figure 7.8):

1. Gather metrics - querying Prometheus for getting all the possible metrics
2. Build Knowledge Base - parsing configuration and adding them thus combining it with the metrics
3. DecisionTreeClassifier - training decision tree classifier
4. Recommend optimizations - finding proper action

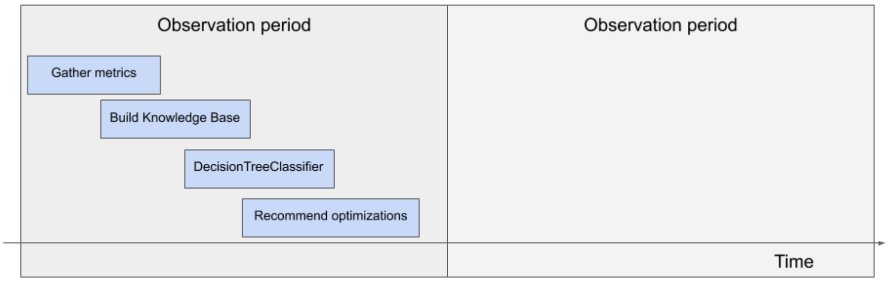


Figure 7.8: Expert system pipeline

cluster_lock_timeout	ACTION
metrics.filter(like='infinispan-app', axis=0).loc[:,'connector_replication_timeout'][0]	Cluster lock is too small

Table 7.5: Cluster lock configuration example

Combining both application configuration and metrics in the same Knowledge Base allows to mix both things together. One of the interesting examples is cluster lock timeout. Since it relies on replicating internal configuration cache in Infinispan, it needs to take replication timeout into consideration. This is a typical example, where deep project knowledge is required to come up with proper configuration. This example has been presented in Table 7.5.

The solution proposed in this paragraph has been used for filing 3 United States Patents with Red Hat affiliation. More information can be found in Section 2.1.

7.3 Experiment environment description and tools used for the evaluation

The experiments were performed in a single stage using OpenShift 4.1 cloud deployed on AWS. The system consisted of 3 Master nodes and 6 Worker nodes. Such a system was initialized solely for running the switching protocol test. Most of the background tasks were turned off to minimize interference with testing harness.

Most of the Intelligent Operator prototype has been implemented using Jupyter Lab [5] environment and rewritten into an Operator. Even though it is unusual to call Python code from Golang, scikit-learn package contains all the Machine Learning models required for the evaluation. All metrics and tests were performed against Infinispan Open Source project.

7.4 Experiments results

The aim of the experiment is to verify if the prototype implemented for verifying the proposed solution properly recognizes misconfigurations in a tested system.

The prototype used a simplified version of Knowledge Base stored in a spreadsheet. Its content has been presented in Table 7.6. Later on, the Operator parses cell by cell and either puts a defaults obtained from metrics or evaluates the expression inside each individual cell. Parsing results have been summarized in Table 7.7.

During the next step, the Operators trains "DecisionTreeClassifier" from scikit-learn package (a Tree-based classifier). Training outcome has been presented in a form of a graph in Figure 7.9. The final step is to predict an action based on metrics and configuration and print it to the standard system output.

The Accuracy for training set of the tree is 1. This indicated that the whole training set was stored into the model.

7.5 Results analysis

Intelligent Operator concept implementation in Python and Golang resulted in a clean and modular code. The approach of combining two programming languages in a single project worked very well for the prototype implementation.

The proposal helps both the community members as well as Red Hat Support Team in addressing most common configuration mistakes on customer site. The prototype has also been successfully used by the Solution Architects Team who install data grid system on the customer's site. It is also worth mentioning, that the solution works in the offline mode, which means it does not connect to a centralized server to produce a recommendation to improve the deployed system configuration.

The model used for the prototype correctly returned the actions based on the observed metrics and tested configurations. However, the way of writing wrong configuration examples into the Knowledge Base (the spreadsheet) is a bit unusual. It requires writing a few examples to get used to it.

[5] https://jupyter.org/

	up	jgroups_kube_ping_count	jgroups_dns_ping_count	cluster_lock_timeout	jvm_threads_current	ACTION
0	0.000	nan	nan	NaN	nan	The service is down. Please make sure it’s running
1	nan	0.000	nan	NaN	nan	There is no KUBE_PING configuration and yet, we are running in Kubernetes
2	nan	nan	0.000	NaN	nan	There is no DNS_PING configuration and yet, we are running in Kubernetes
3	nan	nan	nan	metrics.filter(like=’infinispan-app’,axis=0) .loc[:,’connector_replication_timeout’][0]	nan	Cluster lock is too small
4	nan	nan	nan	NaN	200.000	Lower the number of worker threads

Table 7.6: Raw Knowledge Base for Intelligent Operator

	up	jgroups_kube_ping_count	jgroups_dns_ping_count	cluster_lock_timeout	jvm_threads_current	ACTION	ACTION_ENCODED
0	0.000	1.000	0.000	60000	170.889	The service is down. Please make sure it's running	0
1	1.000	0.000	0.000	60000	170.889	There is no KUBE_PING configuration and yet, we are running in Kubernetes	1
2	1.000	1.000	0.000	60000	170.889	There is no DNS_PING configuration and yet, we are running in Kubernetes	2
3	1.000	1.000	0.000	5000	170.889	Cluster lock is too small	3
4	1.000	1.000	0.000	60000	200.000	Lower the number of worker threads	4

Table 7.7: Parsed Knowledge Base for Intelligent Operator

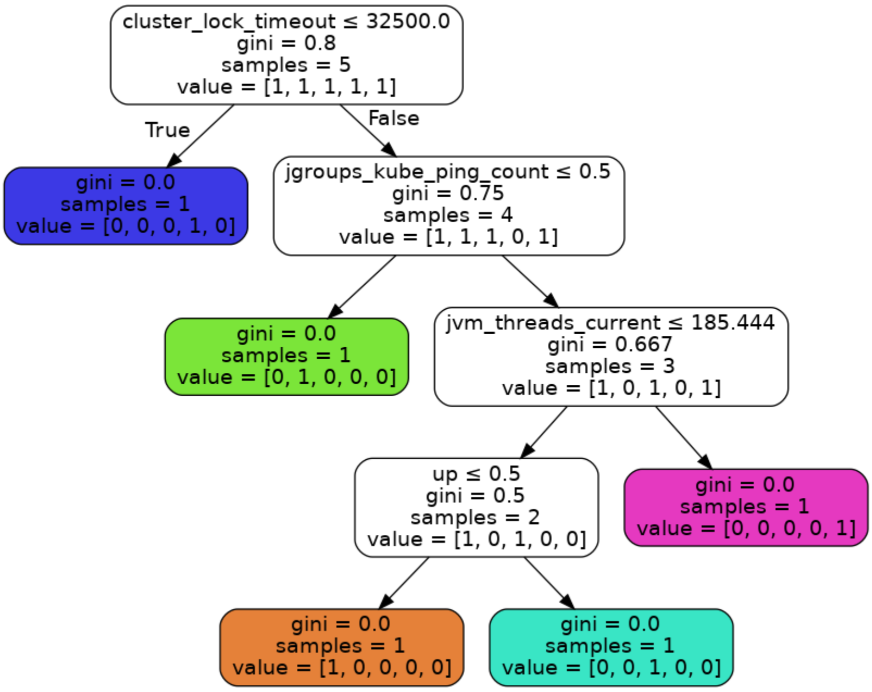


Figure 7.9: Decision boundaries of a Decision Tree Classifier

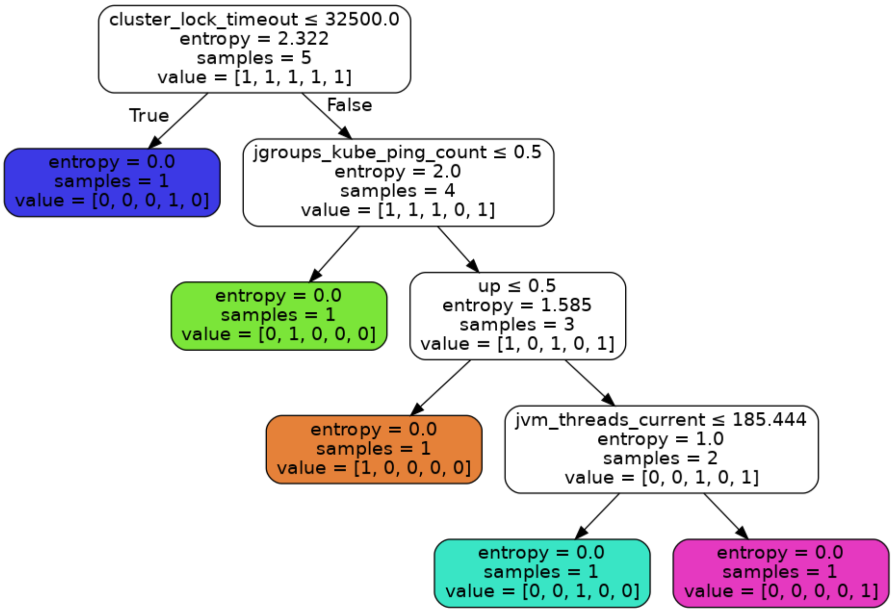


Figure 7.10: Decision boundaries of a Decision Tree Classifier (Entropy)

Decision boundaries or decision graphs generated for Decision Tree Classifier [103] can easily be explained by human eyes (Figure 7.9). The classifier used for the experiment used Gini index by default but this can be easily altered to use entropy (Figure 7.10). The switch however, did not affect the number of nodes in the tree.

Even though, the impact of the proposed solution on thesis' aims is less significant than other proposals', it plays a very important role in maintaining a healthy and efficient runtime configuration. Adjusting send and receive buffer sizes as well as adjusting the data container's size along with its eviction and expiration parameters helps in achieving the best performance results and lowering the memory footprint of the server.

7.5.1 Limitations

The proposed solution has been verified as a proof of concept. It does not meet commercial quality level and its implementation would need to be improved at all layers.

Parsing application level configuration files require supporting multiple file formats, including JSON, YAML, XML or properties files. The presented solution supports only XML files and parses them using XPath expressions. This approach needs to be changed in order to support all the file formats and different expression language should be chosen or invested.

The prototype uses a spreadsheet file as a knowledge base container. In order to make the project usable in the real world scenarios, this approach would probably need to improved. All companies with large Support Department (like Red Hat) use dedicated software modules to maintain communication with external customers and collect a set of known solutions to specific problems. The prototype should connect to those data sources and convert support cases to knowledge base entries automatically.

7.5.2 Future work

Intelligent Operator proposal is the first step towards implementing a fully automatic solution for finding, fixing and tuning the application configuration automatically.

During each new release of a software, vendors perform a full end to end and performance testing of their software. During the product testing phase, software is deployed on a dedicated cloud. All known and tested product configurations need to pass certain tests. This is a perfect opportunity to gather both the metrics and configurations to train the Machine Learning model. Later on, such a model can be deployed on a customer's site and automatically fix the encountered problems. This solution was called ”Secure detection and correction of inefficient application configurations” and has been filed as the United States Patent Request (more information can be found in Section 2.1). The main program loop has been summarized in Figure 7.11. The Train period takes place during the product testing phase in a performance laboratory. During this time, the Operator collects all configuration samples and gathers performance metrics using the pre-defined Key Performance Metrics obtained from a product team. In some scenarios this can be throughput or latency but other metrics might be used as well. The obtained metrics are used to train the Machine Learning model. Once the training is finished, the model is stored into a file and used in Observation periods on a customer's site. During this time, the model observes metrics and tries to predict the best matching configuration observed during the testing period. The model can only predict tested configuration from performance laboratory. Therefore, it follows best known practices and configurations.

One of the most important drawbacks of “Secure detection and correction of inefficient application configurations” is that it needs a training period in a laboratory. However, it is possible to treat the customer site as a training period. This approach makes a lot of sense for Kubernetes clusters hosted by a vendor for multiple clients. Common examples of such configurations are OpenShift Dedicated, Google Cloud Engine or Tectonic. With this, very

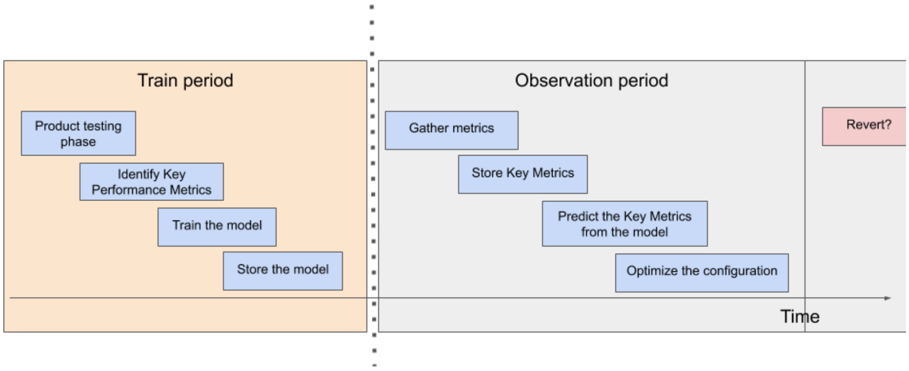


Figure 7.11: Secure detection and correction of inefficient application configurations pipeline

specific setup, cloud vendors (and their infrastructure services) have access to all customer deployments. This allows to create another solution - “AI configuration management advisor”. This approach does not require the training period and can be implemented in a product agnostic way. The main controller loop has been shown in Figure 7.12. The Operator requires the product teams to define Key Performance Metrics and Configuration Parameter Space. The former is used for training the model and comparing configurations between each other. The Operator is allowed to perform experiments with application configuration - change blindly parameters and observe how Key Performance Metrics react to it. This approach needs to be constrained by declaring Configuration Parameter Space. The idea is to define the allowed values for certain parameters (e.g. changing Infinispan Cache type between “replicated” and “distributed” and not allowing changing to “local”). The Operator keeps track of all the measured configurations and all previously tried configuration parameters.

The most interesting aspect of this solution is that it can work hand-in-hand with human operator (from operations team). Since the Operator keeps track of all the configuration changes - it can learn based on the human-generated configuration examples.

Intelligent Operator prototype is far from being finished. The tool has been designed to solve real problems by helping users as well as community project maintainers and Red Hat Support Team. At the time or writing this paragraph, there is ongoing discussion if such a tool is needed and whether the scope should be extended to a commercial product.

The idea might be also integrated with the Open Data Hub initiative [76] but this path has not been investigated.

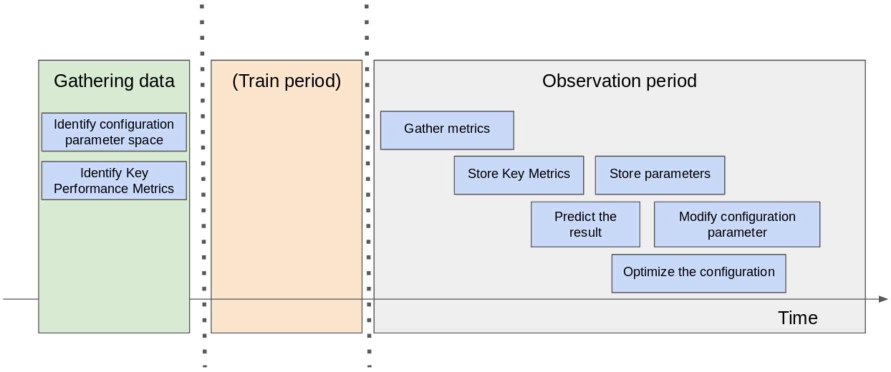

Gathering data
Identify configuration parameter space
Identify Key Performance Metrics
(Train period)
Observation period
Gather metrics
Store Key Metrics
Store parameters
Predict the result
Modify configuration parameter
Optimize the configuration
Time

Figure 7.12: AI configuration management adviser pipeline

Chapter 8

Conclusions

The list below presents the most important outcomes of the research focusing on innovative ideas and their practical aspects:

- The thesis statement has been positively verified. It has been proven that it is possible to improve communication performance, defined by either throughput or latency, by using new solutions and algorithms for protocol negotiation and client side load balancing between a client application and a service deployed in the cloud.
- The following methods and algorithms have been designed to meet the aims of the thesis:
  - The server memory footprint was lowered by introducing multi-tenancy support for Infinispan Server. Individual clients are recognized by analyzing the TLS/SNI Hostname field (which is not encrypted) and authenticated by using a X509 Certificate. This approach makes sure each client can only access their own data (and not other clients' - tenants'). At the same time, TLS/SNI can be used by a router in the cloud, allowing a client to connect to a data grid from the outside world. This work has been positively reviewed by Infinispan community and integrated into both Infinispan Open Source project and Red Hat Data Grid product.
  - Communication performance has been improved by enabling client side load balancing using multiple Load Balancers in front of a service deployed in the cloud. Even though this solution had positive reviews from the academic community, Infinispan project members decided to reject it as it required running a third-party component apart from the server. However, it has been used as the foundation for building cross-site replication feature for the clouds.

- Overall communication performance between a server deployed in the cloud and a client from outside of the cloud has been improved by enabling custom binary protocols to be used in a typical public cloud scenario. The proposed method uses both the TLS/ALPN protocol negotiation as well as HTTP/1.1 Upgrade procedure to reuse the same transport connection and switch to the fastest possible protocol. The proposed solution has been positively reviewed by the Infinispan community and has been enabled as a default connection negotiation method in Infinispan 10 series.
- An expert system has been proposed to help users to fix common configuration mistakes. The solution has been built on top of Operator concepts for the cloud. The idea has been turned into three United States Patents and has the potential to be turned into a community project or even into a commercial product.

Many solutions presented in this thesis have practical impact. Some of them were integrated into main source code repository and have been improved since then. This proves they solve real problems encountered by the users.

Intelligent Operator concept has a lot of potential in it. Even though it has been proven to work as a proof of concept, it has the potential to become a fully autonomic, configuration management and tuning system.

Chapter 9

Glossary

- (to) backport - Copy given functionality from an open-source project (called also upstream in this context) to a commercial product. A common example is the multi-tenancy feature, which was created for Infinispan project (upstream) and was backported to the Red Hat Data Grid product.

- Container image - An immutable file, typically containing an application with its dependent libraries and an Operating System, that can run an isolated process on a host machine (typically as a Linux process). An image is usually built using Docker Daemon or project Buildah.

- Docker container - See Linux Container.

- Ectd - A key-value based data store.

- (Amazon) EC2 - a public cloud offering by Amazon.

- GCE - Google Container Engine - a public Kubernetes offering from Google.

- GCP - Google Compute Cloud - a public cloud offering from Google.

- Greenfield project - a new project that starts up.

- Kubernetes - An Open Source, container-based cloud.

- Linux container - A running Container Image on a host machine.

- LXC - Linux Container project.

- MCDM - Multiple Criteria Decision Making.

- Microservice - A small, isolated piece of software designed to operate within a larger system.

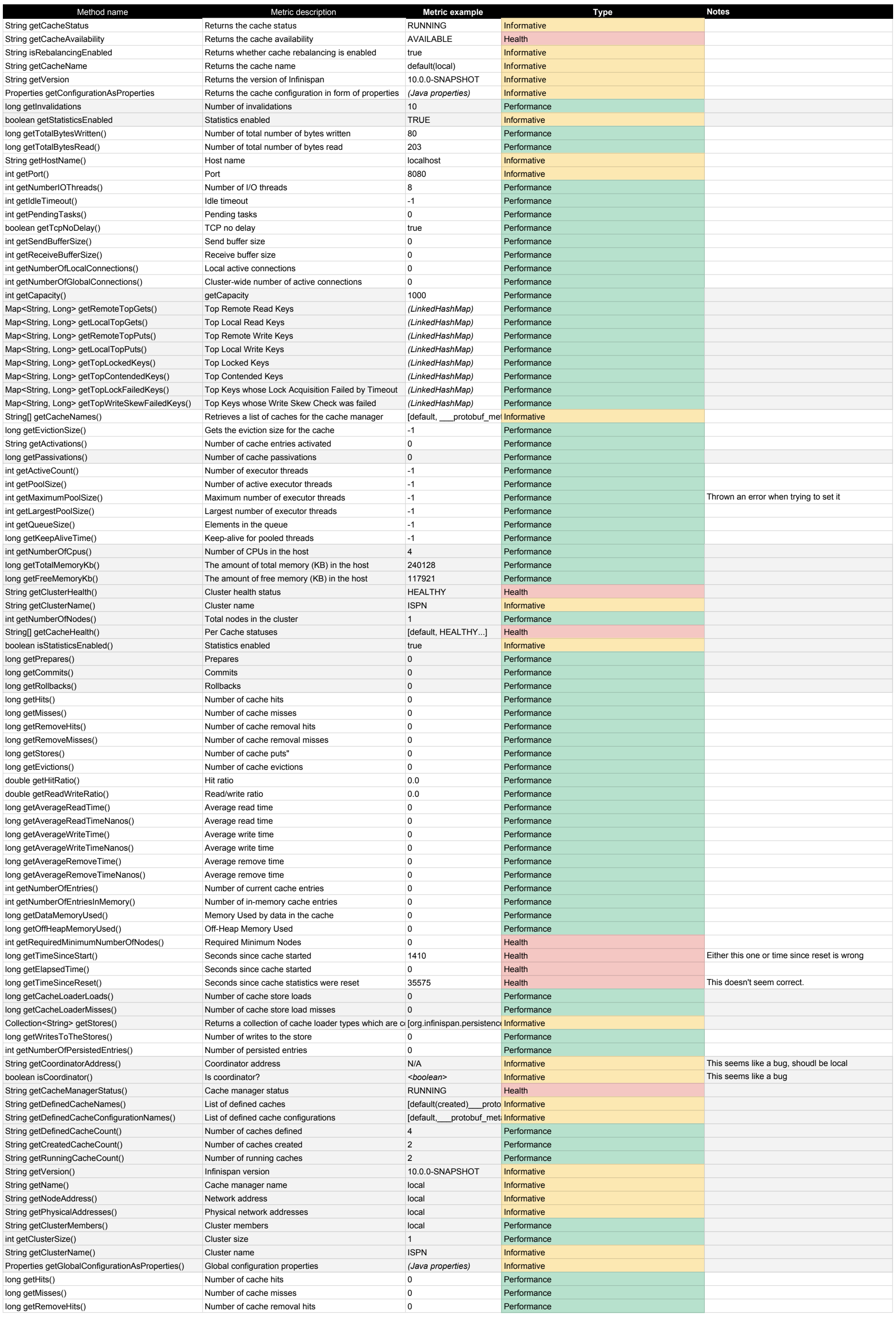
- Multi-tenancy - designing the system in such a way, so that it can be used by multiple users without sharing their data.
- MSE - Mean Square Error.
- OpenShift - a public cloud offering from Red Hat.
- RSE - Residual Standard Error.
- TLS - Transport Layer Security - a common name for encryption protocol.
- TLS/ALPN - Application Level Protocol Negotiation extension for TLS.
- TLS/SNI - Service Name Indication extension for TLS.
- VM - A Virtual Machine.
- Xen - A virtualization hypervisor.

Chapter 10

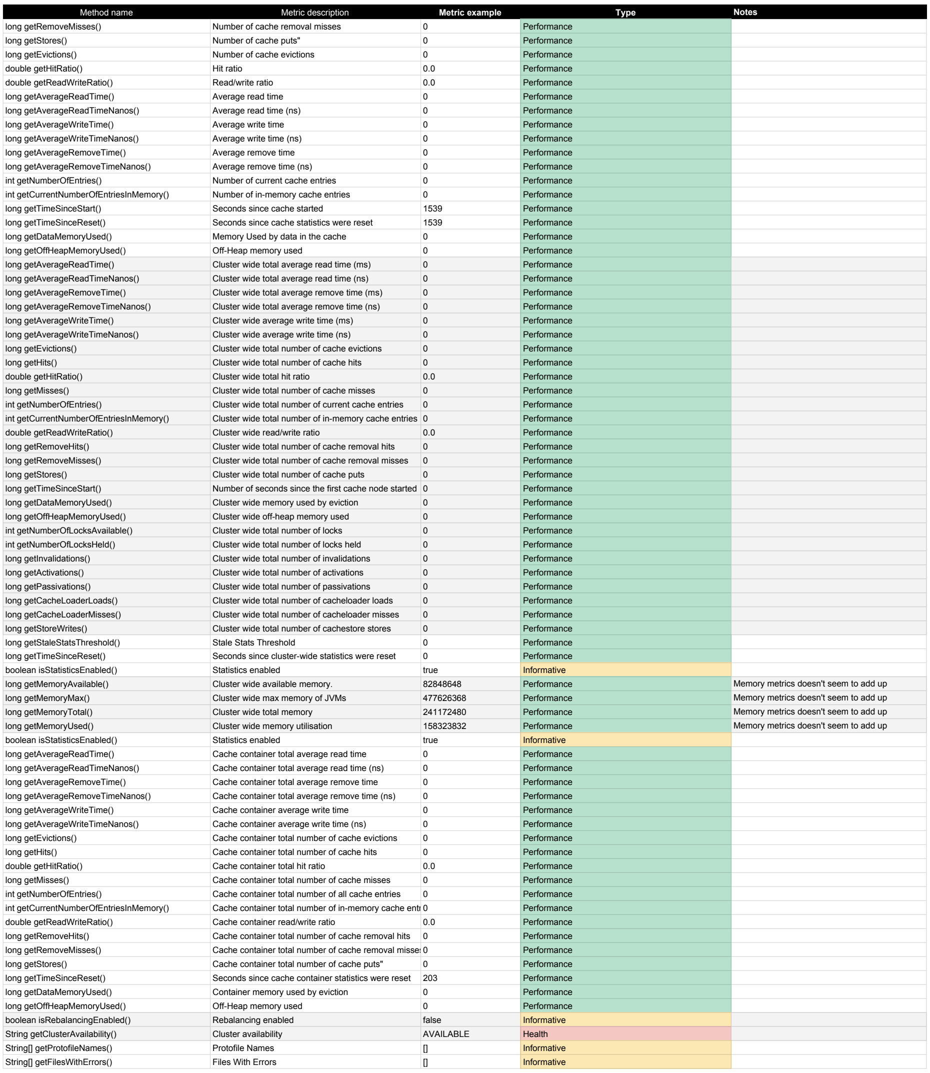
Appendix

This paragraph contains additional materials used for investigating and implementing solutions proposed in this thesis.

10.1 Infinispan metrics

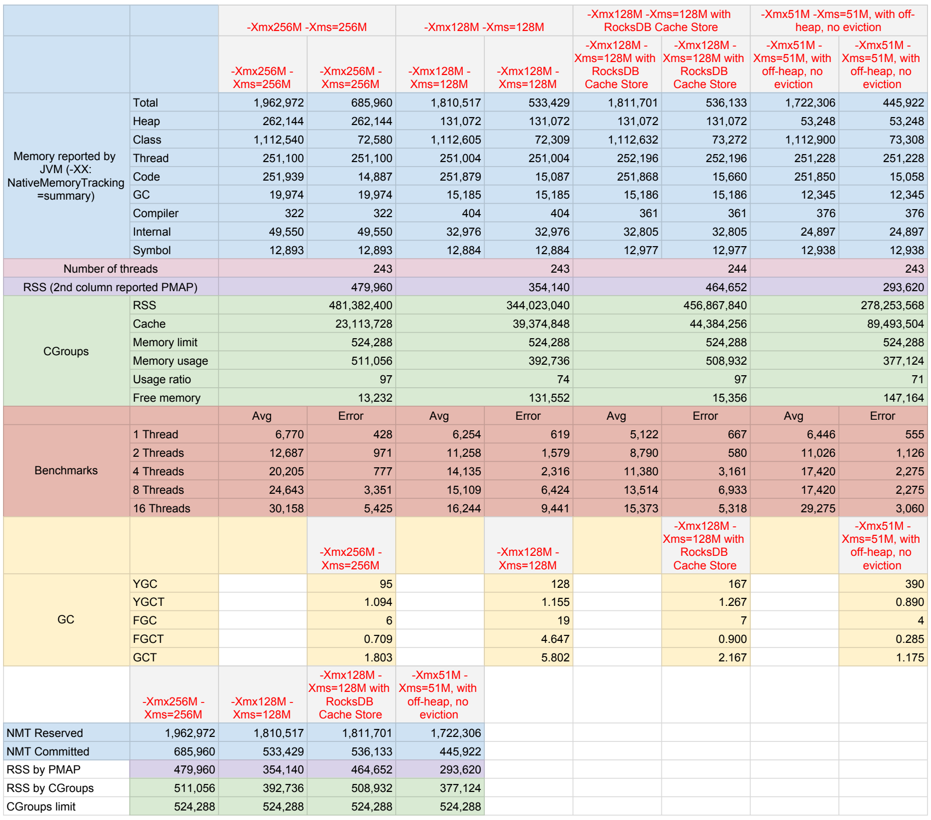


Method name	Metric description	Metric example	Type	Notes
String getCacheStatus	Returns the cache status	RUNNING	Informative	
String getCacheAvailability	Returns the cache availability	AVAILABLE	Health	
String isRebalancingEnabled	Returns whether cache rebalancing is enabled	true	Informative	
String getCacheName	Returns the cache name	default(local)	Informative	
String getVersion	Returns the version of Infinispan	10.0.0-SNAPSHOT	Informative	
Properties getConfigurationAsProperties	Returns the cache configuration in form of properties	*(Java properties)*	Informative	
long getInvalidations	Number of invalidations	10	Performance	
boolean getStatisticsEnabled	Statistics enabled	TRUE	Informative	
long getTotalBytesWritten()	Number of total number of bytes written	80	Performance	
long getTotalBytesRead()	Number of total number of bytes read	203	Performance	
String getHostName()	Host name	localhost	Informative	
int getPort()	Port	8080	Informative	
int getNumberIOThreads()	Number of I/O threads	8	Performance	
int getIdleTimeout()	Idle timeout	-1	Performance	
int getPendingTasks()	Pending tasks	0	Performance	
boolean getTcpNoDelay()	TCP no delay	true	Performance	
int getSendBufferSize()	Send buffer size	0	Performance	
int getReceiveBufferSize()	Receive buffer size	0	Performance	
int getNumberOfLocalConnections()	Local active connections	0	Performance	
int getNumberOfGlobalConnections()	Cluster-wide number of active connections	0	Performance	
int getCapacity()	getCapacity	1000	Performance	
Map<String, Long> getRemoteTopGets()	Top Remote Read Keys	*(LinkedHashMap)*	Performance	
Map<String, Long> getLocalTopGets()	Top Local Read Keys	*(LinkedHashMap)*	Performance	
Map<String, Long> getRemoteTopPuts()	Top Remote Write Keys	*(LinkedHashMap)*	Performance	
Map<String, Long> getLocalTopPuts()	Top Local Write Keys	*(LinkedHashMap)*	Performance	
Map<String, Long> getTopLockedKeys()	Top Locked Keys	*(LinkedHashMap)*	Performance	
Map<String, Long> getTopContendedKeys()	Top Contended Keys	*(LinkedHashMap)*	Performance	
Map<String, Long> getTopLockFailedKeys()	Top Keys whose Lock Acquisition Failed by Timeout	*(LinkedHashMap)*	Performance	
Map<String, Long> getTopWriteSkewFailedKeys()	Top Keys whose Write Skew Check was failed	*(LinkedHashMap)*	Performance	
String[] getCacheNames()	Retrieves a list of caches for the cache manager	[default, ___protobuf_met	Informative	
long getEvictionSize()	Gets the eviction size for the cache	-1	Performance	
String getActivations()	Number of cache entries activated	0	Performance	
long getPassivations()	Number of cache passivations	0	Performance	
int getActiveCount()	Number of executor threads	-1	Performance	
int getPoolSize()	Number of active executor threads	-1	Performance	
int getMaximumPoolSize()	Maximum number of executor threads	-1	Performance	Thrown an error when trying to set it
int getLargestPoolSize()	Largest number of executor threads	-1	Performance	
int getQueueSize()	Elements in the queue	-1	Performance	
long getKeepAliveTime()	Keep-alive for pooled threads	-1	Performance	
int getNumberOfCpus()	Number of CPUs in the host	4	Performance	
long getTotalMemoryKb()	The amount of total memory (KB) in the host	240128	Performance	
long getFreeMemoryKb()	The amount of free memory (KB) in the host	117921	Performance	
String getClusterHealth()	Cluster health status	HEALTHY	Health	
String getClusterName()	Cluster name	ISPN	Informative	
int getNumberOfNodes()	Total nodes in the cluster	1	Performance	
String[] getCacheHealth()	Per Cache statuses	[default, HEALTHY...]	Health	
boolean isStatisticsEnabled()	Statistics enabled	true	Informative	
long getPrepares()	Prepares	0	Performance	
long getCommits()	Commits	0	Performance	
long getRollbacks()	Rollbacks	0	Performance	
long getHits()	Number of cache hits	0	Performance	
long getMisses()	Number of cache misses	0	Performance	
long getRemoveHits()	Number of cache removal hits	0	Performance	
long getRemoveMisses()	Number of cache removal misses	0	Performance	
long getStores()	Number of cache puts"	0	Performance	
long getEvictions()	Number of cache evictions	0	Performance	
double getHitRatio()	Hit ratio	0.0	Performance	
double getReadWriteRatio()	Read/write ratio	0.0	Performance	
long getAverageReadTime()	Average read time	0	Performance	
long getAverageReadTimeNanos()	Average read time	0	Performance	
long getAverageWriteTime()	Average write time	0	Performance	
long getAverageWriteTimeNanos()	Average write time	0	Performance	
long getAverageRemoveTime()	Average remove time	0	Performance	
long getAverageRemoveTimeNanos()	Average remove time	0	Performance	
int getNumberOfEntries()	Number of current cache entries	0	Performance	
int getNumberOfEntriesInMemory()	Number of in-memory cache entries	0	Performance	
long getDataMemoryUsed()	Memory Used by data in the cache	0	Performance	
long getOffHeapMemoryUsed()	Off-Heap Memory Used	0	Performance	
int getRequiredMinimumNumberOfNodes()	Required Minimum Nodes	0	Health	
long getTimeSinceStart()	Seconds since cache started	1410	Health	Either this one or time since reset is wrong
long getElapsedTime()	Seconds since cache started	0	Health	
long getTimeSinceReset()	Seconds since cache statistics were reset	35575	Health	This doesn't seem correct.
long getCacheLoaderLoads()	Number of cache store loads	0	Performance	
long getCacheLoaderMisses()	Number of cache store load misses	0	Performance	
Collection<String> getStores()	Returns a collection of cache loader types which are c	[org.infinispan.persistence	Informative	
long getWritesToTheStores()	Number of writes to the store	0	Performance	
int getNumberOfPersistedEntries()	Number of persisted entries	0	Performance	
String getCoordinatorAddress()	Coordinator address	N/A	Informative	This seems like a bug, shoudl be local
boolean isCoordinator()	Is coordinator?	*<boolean>*	Informative	This seems like a bug
String getCacheManagerStatus()	Cache manager status	RUNNING	Health	
String getDefinedCacheNames()	List of defined caches	[default(created)___proto	Informative	
String getDefinedCacheConfigurationNames()	List of defined cache configurations	[default,___protobuf_meta	Informative	
String getDefinedCacheCount()	Number of caches defined	4	Performance	
String getCreatedCacheCount()	Number of caches created	2	Performance	
String getRunningCacheCount()	Number of running caches	2	Performance	
String getVersion()	Infinispan version	10.0.0-SNAPSHOT	Informative	
String getName()	Cache manager name	local	Informative	
String getNodeAddress()	Network address	local	Informative	
String getPhysicalAddresses()	Physical network addresses	local	Informative	
String getClusterMembers()	Cluster members	local	Performance	
int getClusterSize()	Cluster size	1	Performance	
String getClusterName()	Cluster name	ISPN	Informative	
Properties getGlobalConfigurationAsProperties()	Global configuration properties	*(Java properties)*	Informative	
long getHits()	Number of cache hits	0	Performance	
long getMisses()	Number of cache misses	0	Performance	
long getRemoveHits()	Number of cache removal hits	0	Performance	



Method name	Metric description	Metric example	Type	Notes
long getRemoveMisses()	Number of cache removal misses	0	Performance	
long getStores()	Number of cache puts"	0	Performance	
long getEvictions()	Number of cache evictions	0	Performance	
double getHitRatio()	Hit ratio	0.0	Performance	
double getReadWriteRatio()	Read/write ratio	0.0	Performance	
long getAverageReadTime()	Average read time	0	Performance	
long getAverageReadTimeNanos()	Average read time (ns)	0	Performance	
long getAverageWriteTime()	Average write time	0	Performance	
long getAverageWriteTimeNanos()	Average write time (ns)	0	Performance	
long getAverageRemoveTime()	Average remove time	0	Performance	
long getAverageRemoveTimeNanos()	Average remove time (ns)	0	Performance	
int getNumberOfEntries()	Number of current cache entries	0	Performance	
int getCurrentNumberOfEntriesInMemory()	Number of in-memory cache entries	0	Performance	
long getTimeSinceStart()	Seconds since cache started	1539	Performance	
long getTimeSinceReset()	Seconds since cache statistics were reset	1539	Performance	
long getDataMemoryUsed()	Memory Used by data in the cache	0	Performance	
long getOffHeapMemoryUsed()	Off-Heap memory used	0	Performance	
long getAverageReadTime()	Cluster wide total average read time (ms)	0	Performance	
long getAverageReadTimeNanos()	Cluster wide total average read time (ns)	0	Performance	
long getAverageRemoveTime()	Cluster wide total average remove time (ms)	0	Performance	
long getAverageRemoveTimeNanos()	Cluster wide total average remove time (ns)	0	Performance	
long getAverageWriteTime()	Cluster wide average write time (ms)	0	Performance	
long getAverageWriteTimeNanos()	Cluster wide average write time (ns)	0	Performance	
long getEvictions()	Cluster wide total number of cache evictions	0	Performance	
long getHits()	Cluster wide total number of cache hits	0	Performance	
double getHitRatio()	Cluster wide total hit ratio	0.0	Performance	
long getMisses()	Cluster wide total number of cache misses	0	Performance	
int getNumberOfEntries()	Cluster wide total number of current cache entries	0	Performance	
int getCurrentNumberOfEntriesInMemory()	Cluster wide total number of in-memory cache entries	0	Performance	
double getReadWriteRatio()	Cluster wide read/write ratio	0.0	Performance	
long getRemoveHits()	Cluster wide total number of cache removal hits	0	Performance	
long getRemoveMisses()	Cluster wide total number of cache removal misses	0	Performance	
long getStores()	Cluster wide total number of cache puts	0	Performance	
long getTimeSinceStart()	Number of seconds since the first cache node started	0	Performance	
long getDataMemoryUsed()	Cluster wide memory used by eviction	0	Performance	
long getOffHeapMemoryUsed()	Cluster wide off-heap memory used	0	Performance	
int getNumberOfLocksAvailable()	Cluster wide total number of locks	0	Performance	
int getNumberOfLocksHeld()	Cluster wide total number of locks held	0	Performance	
long getInvalidations()	Cluster wide total number of invalidations	0	Performance	
long getActivations()	Cluster wide total number of activations	0	Performance	
long getPassivations()	Cluster wide total number of passivations	0	Performance	
long getCacheLoaderLoads()	Cluster wide total number of cacheloader loads	0	Performance	
long getCacheLoaderMisses()	Cluster wide total number of cacheloader misses	0	Performance	
long getStoreWrites()	Cluster wide total number of cachestore stores	0	Performance	
long getStaleStatsThreshold()	Stale Stats Threshold	0	Performance	
long getTimeSinceReset()	Seconds since cluster-wide statistics were reset	0	Performance	
boolean isStatisticsEnabled()	Statistics enabled	true	Informative	
long getMemoryAvailable()	Cluster wide available memory.	82848648	Performance	Memory metrics doesn't seem to add up
long getMemoryMax()	Cluster wide max memory of JVMs	477626368	Performance	Memory metrics doesn't seem to add up
long getMemoryTotal()	Cluster wide total memory	241172480	Performance	Memory metrics doesn't seem to add up
long getMemoryUsed()	Cluster wide memory utilisation	158323832	Performance	Memory metrics doesn't seem to add up
boolean isStatisticsEnabled()	Statistics enabled	true	Informative	
long getAverageReadTime()	Cache container total average read time	0	Performance	
long getAverageReadTimeNanos()	Cache container total average read time (ns)	0	Performance	
long getAverageRemoveTime()	Cache container total average remove time	0	Performance	
long getAverageRemoveTimeNanos()	Cache container total average remove time (ns)	0	Performance	
long getAverageWriteTime()	Cache container average write time	0	Performance	
long getAverageWriteTimeNanos()	Cache container average write time (ns)	0	Performance	
long getEvictions()	Cache container total number of cache evictions	0	Performance	
long getHits()	Cache container total number of cache hits	0	Performance	
double getHitRatio()	Cache container total hit ratio	0.0	Performance	
long getMisses()	Cache container total number of cache misses	0	Performance	
int getNumberOfEntries()	Cache container total number of all cache entries	0	Performance	
int getCurrentNumberOfEntriesInMemory()	Cache container total number of in-memory cache entr	0	Performance	
double getReadWriteRatio()	Cache container read/write ratio	0.0	Performance	
long getRemoveHits()	Cache container total number of cache removal hits	0	Performance	
long getRemoveMisses()	Cache container total number of cache removal misse	0	Performance	
long getStores()	Cache container total number of cache puts"	0	Performance	
long getTimeSinceReset()	Seconds since cache container statistics were reset	203	Performance	
long getDataMemoryUsed()	Container memory used by eviction	0	Performance	
long getOffHeapMemoryUsed()	Off-Heap memory used	0	Performance	
boolean isRebalancingEnabled()	Rebalancing enabled	false	Informative	
String getClusterAvailability()	Cluster availability	AVAILABLE	Health	
String[] getProtofileNames()	Protofile Names	[]	Informative	
String[] getFilesWithErrors()	Files With Errors	[]	Informative	

10.2 Infinispan 9 memory usage



		-Xmx256M -Xms=256M		-Xmx128M -Xms=128M		-Xmx128M -Xms=128M with RocksDB Cache Store		-Xmx51M -Xms=51M, with off-heap, no eviction	
		-Xmx256M -Xms=256M	-Xmx256M -Xms=256M	-Xmx128M -Xms=128M	-Xmx128M -Xms=128M	-Xmx128M -Xms=128M with RocksDB Cache Store	-Xmx128M -Xms=128M with RocksDB Cache Store	-Xmx51M -Xms=51M, with off-heap, no eviction	-Xmx51M -Xms=51M, with off-heap, no eviction
Memory reported by JVM (-XX: NativeMemoryTracking =summary)	Total	1,962,972	685,960	1,810,517	533,429	1,811,701	536,133	1,722,306	445,922
	Heap	262,144	262,144	131,072	131,072	131,072	131,072	53,248	53,248
	Class	1,112,540	72,580	1,112,605	72,309	1,112,632	73,272	1,112,900	73,308
	Thread	251,100	251,100	251,004	251,004	252,196	252,196	251,228	251,228
	Code	251,939	14,887	251,879	15,087	251,868	15,660	251,850	15,058
	GC	19,974	19,974	15,185	15,185	15,186	15,186	12,345	12,345
	Compiler	322	322	404	404	361	361	376	376
	Internal	49,550	49,550	32,976	32,976	32,805	32,805	24,897	24,897
	Symbol	12,893	12,893	12,884	12,884	12,977	12,977	12,938	12,938
Number of threads		243		243		244		243	
RSS (2nd column reported PMAP)		479,960		354,140		464,652		293,620	
CGroups	RSS	481,382,400		344,023,040		456,867,840		278,253,568	
	Cache	23,113,728		39,374,848		44,384,256		89,493,504	
	Memory limit	524,288		524,288		524,288		524,288	
	Memory usage	511,056		392,736		508,932		377,124	
	Usage ratio	97		74		97		71	
	Free memory	13,232		131,552		15,356		147,164	
		Avg	Error	Avg	Error	Avg	Error	Avg	Error
Benchmarks	1 Thread	6,770	428	6,254	619	5,122	667	6,446	555
	2 Threads	12,687	971	11,258	1,579	8,790	580	11,026	1,126
	4 Threads	20,205	777	14,135	2,316	11,380	3,161	17,420	2,275
	8 Threads	24,643	3,351	15,109	6,424	13,514	6,933	17,420	2,275
	16 Threads	30,158	5,425	16,244	9,441	15,373	5,318	29,275	3,060
			-Xmx256M -Xms=256M		-Xmx128M -Xms=128M		-Xmx128M -Xms=128M with RocksDB Cache Store		-Xmx51M -Xms=51M, with off-heap, no eviction
GC	YGC		95		128		167		390
	YGCT		1.094		1.155		1.267		0.890
	FGC		6		19		7		4
	FGCT		0.709		4.647		0.900		0.285
	GCT		1.803		5.802		2.167		1.175
	-Xmx256M -Xms=256M	-Xmx128M -Xms=128M	-Xmx128M -Xms=128M with RocksDB Cache Store	-Xmx51M -Xms=51M, with off-heap, no eviction					
NMT Reserved	1,962,972	1,810,517	1,811,701	1,722,306					
NMT Committed	685,960	533,429	536,133	445,922					
RSS by PMAP	479,960	354,140	464,652	293,620					
RSS by CGroups	511,056	392,736	508,932	377,124					
CGroups limit	524,288	524,288	524,288	524,288					

List of Figures

List of Tables

Bibliography

[1] M. Brock A. Goscinski. “A technology to expose a cluster as a service in a cloud”. In: *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107* 107 (2010), pp. 3–12. URL: http://dl.acm.org/citation.cfm?id=1862295.

[2] *A modern hybrid cloud platform for innovation: Containers on Cloud with Openshift on OpenStack*. URL: https://www.redhat.com/ja/blog/modern-hybrid-cloud-platform-innovation-containers-cloud-openshift-openstack (visited on 01/24/2020).

[3] F. Pop A. Sfrent. “Asymptotic scheduling for many task computing in Big Data platforms”. In: *Information Sciences* 319.Supplement C (2015). Energy Efficient Data, Services and Memory Management in Big Data Information Systems, pp. 71–91. ISSN: 0020-0255. DOI: 10.1016/j.ins.2015.03.053. URL: http://www.sciencedirect.com/science/article/pii/S0020025515002182.

[4] C. Şerbănescu F. Pop A. Sîrbu C. Pop. “Predicting provisioning and booting times in a Metal-as-a-service system”. In: *Future Generation Computer Systems* 72.Supplement C (2017), pp. 180–192. ISSN: 0167-739X. DOI: 10.1016/j.future.2016.07.001. URL: http://www.sciencedirect.com/science/article/pii/S0167739X1630231X.

[5] A. Casari A. Zheng. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. 1st. O’Reilly Media, Inc., 2018. ISBN: 1491953241.

[6] *Additional Kubernetes controllers from Fabric8 you can use with your microservice*. URL: https://blog.fabric8.io/additional-kubernetes-controllers-from-fabric8-you-can-use-with-your-microservice-3126a2c4c132 (visited on 05/26/2017).

[7] *Adrian Cockcroft's Blog: Ops, DevOps and PaaS (NoOps) at Netflix*. URL: http://perfcap.blogspot.com/2012/03/ops-devops-and-noops-at-netflix.html (visited on 12/10/2019).

[8] *Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta*. URL: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/ (visited on 08/20/2015).

[9] *Apache Mesos*. URL: http://mesos.apache.org/ (visited on 01/31/2020).

[10] *AppArmor - Ubuntu Wiki*. URL: https://wiki.ubuntu.com/AppArmor (visited on 01/30/2020).

[11] *AWS pricing*. 2018. URL: https://aws.amazon.com/elasticloadbalancing/pricing/ (visited on 07/31/2018).

[12] D. Oppenheimer E. Brewer-J. Wilkes B. Burns B. Grant. "Borg, Omega, and Kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57. ISSN: 00010782. DOI: 10.1145/2890784. URL: http://dl.acm.org/citation.cfm?doid=2930840.2890784.

[13] *Best Practices for Becoming an Exceptional Postgres DBA*. URL: https://www.slideshare.net/EnterpriseDB/dba-best-practices-webinar-slides-final (visited on 12/13/2019).

[14] M. Bishop. "Hypertext transfer protocol version 3 (HTTP/3)". In: *Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-20* (2019).

[15] J.A. Breuker B.J. Wielinga A.Th. Schreiber. "KADS: a modelling approach to knowledge engineering". In: *Knowledge Acquisition* 4.1 (1992), pp. 5–53. ISSN: 1042-8143. DOI: 10.1016/1042-8143(92)90013-Q. URL: https://www.sciencedirect.com/science/article/pii/104281439290013Q.

[16] P. Chaganti. *Xen Virtualization*. Packt Publishing Ltd, 2007, p. 149. ISBN: 1847192491. URL: https://books.google.com/books?id=kRJUkF9DaxIC{\&}pgis=1.

[17] Sameer Singh Chauhan et al. "Brokering in Interconnected Cloud Computing Environments: A Survey". In: *J. Parallel Distrib. Comput.* (2018). DOI: 10.1016/j.jpdc.2018.08.001. URL: https://doi.org/10.1016/j.jpdc.2018.08.001.

[18] *Cincinnati for OpenShift*. URL: https://github.com/openshift/cincinnati (visited on 01/31/2020).

[19] *Cloud Native Computing Foundation*. URL: https://www.cncf.io/ (visited on 01/31/2020).

[20] *Cloud pricing comparison: AWS vs. Microsoft Azure vs. Google Cloud vs. IBM Cloud*. URL: https://www.infoworld.com/article/3237566/cloud-pricing-comparison-aws-vs-azure-vs-google-vs-ibm.html (visited on 02/28/2020).

[21] *Custom Resources - Kubernetes*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (visited on 12/11/2019).

[22] I. Leontiadis Y. Grunenberger-M. Mellia M. Munafò K. Papagiannaki P. Steenkiste D. Naylor A. Finamore. "The Cost of the 'S' in HTTPS". In: (2014). DOI: 10.1145/2674005.2674991. URL: http://dx.doi.org/10.1145/2674005.2674991..

[23] G.J. Gordon B. Zhang D. Van Aken A. Pavlo. "Automatic Database Management System Tuning Through Large-scale Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1009–1024. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064029. URL: http://doi.acm.org/10.1145/3035918.3064029.

[24] P. Dangeti. *Statistics for machine learning : build supervised, unsupervised, and reinforcement learning models using both Python and R*. ISBN: 9781788295758. URL: https://learning.oreilly.com/library/view/statistics-for-machine/9781788295758/.

[25] *Docker - Swarm Mode*. URL: https://docs.docker.com/engine/swarm/ (visited on 01/31/2020).

[26] *Drools Project*. URL: https://www.drools.org/ (visited on 03/04/2020).

[27] M. Morisio E. Çano. *Hybrid recommender systems: A systematic literature review*. 2017. DOI: 10.3233/IDA-163209. arXiv: 1901.03888.

[28] *Edge Triggered Vs Level Triggered interrupts | Linux kernel*. URL: http://venkateshabbarapu.blogspot.com/2013/03/edge-triggered-vs-level-triggered.html (visited on 12/10/2019).

[29] *Elastic Load Balancing - Cloud Network Load Balancer*. URL: https://aws.amazon.com/elasticloadbalancing/ (visited on 05/29/2017).

[30] *Factor Analisys Scikit-learn*. URL: scikit-learn.org/stable/modules/generated/sklearn.decomposition.FactorAnalysis.html (visited on 09/12/2019).

[31] *FeedHenry - Accelerate mobile app development*. URL: http://feedhenry.org/ (visited on 07/12/2019).

[32] M. Jampani G. Kakulapati A. Lakshman A. Pilchin S. Sivasubramanian P. Vosshall W. Vogels G. Decandia D. Hastorun. “Dynamo: Amazon’s Highly Available Key-value Store”. In: (2007).

[33] T. Hastie R. Tibshirani G. James D. Witten. *An introduction to Statistical Learning*. 2000. ISBN: 978-1-4614-7137-0. DOI: 10.1007/978-1-4614-7138-7. arXiv: arXiv:1011.1669v3.

[34] B. Claise J. Quittek G. Sadasivan JN. Brownlee. *Transport Layer Security (TLS) Extensions: Extension Definitions*. URL: https://tools.ietf.org/html/rfc6066.

[35] *Gartner Says Worldwide Public Cloud Services Market to Grow 18 Percent in 2017*. URL: http://www.gartner.com/newsroom/id/3616417 (visited on 05/05/2017).

[36] *Google Cloud Load Balancing - High Performance, Global & Scalable | Google Cloud Platform*. URL: https://cloud.google.com/load-balancing/ (visited on 05/29/2017).

[37] *Google Cloud Platform Blog: Containers, VMs, Kubernetes and VMware*. URL: https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html (visited on 05/29/2017).

[38] *Google Cloud Platform Blog: Enter the Andromeda zone - Google Cloud Platform’s latest networking stack*. URL: https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html (visited on 03/07/2019).

[39] *Google Cloud Platform Blog: Enter the Andromeda zone - Google Cloud Platform’s latest networking stack*. URL: https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html (visited on 05/29/2017).

[40] *Google Cloud pricing*. 2018. URL: https://cloud.google.com/pricing/ (visited on 07/31/2018).

[41] *Google Transparency Repport*. URL: https://transparencyreport.google.com/https/overview?hl=en (visited on 02/27/2020).

[42] *Graal VM - High-performance polyglot VM*. URL: https://www.graalvm.org/ (visited on 02/28/2020).

[43] *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer*. URL: http://www.haproxy.org/ (visited on 05/29/2017).

[44] H.A. Güvenir I. Uysal. "An overview of regression techniques for knowledge discovery". In: *Knowledge Eng. Review* 14.4 (1999), pp. 319–340. URL: http://journals.cambridge.org/action/displayAbstract?aid=36077.

[45] *Implement router http/2 support ticket*. 2018. URL: https://trello.com/c/qzvlzuyx/27-3-implement-router-http-2-support-terminating-at-the-router-router (visited on 07/31/2018).

[46] *Istio*. URL: https://istio.io/ (visited on 01/31/2020).

[47] K. Nova J. Garrison. *Cloud native infrastructure : patterns for scalable infrastructure and applications in a dynamic environment*. ISBN: 9781491984307. URL: https://learning.oreilly.com/library/view/cloud-native-infrastructure/9781491984291/.

[48] M. Thomson J. Iyengar. "Quic: A udp-based multiplexed and secure transport". In: *Internet Engineering Task Force, Internet-Draft draftietf-quic-transport-17* (2018).

[49] M.J. Realff J.H. Lee J. Shin. "Machine learning: Overview of the recent progresses and implications for the process systems engineering field". In: *Computers & Chemical Engineering* 114 (2018), pp. 111–121.

[50] M.J. Realff J.H. Lee J.Shin. "Machine learning: Overview of the recent progresses and implications for the process systems engineering field". In: *Computers & Chemical Engineering* 114 (2018), pp. 111–121. DOI: 10.1016/j.compchemeng.2017.10.008. URL: https://doi.org/10.1016/j.compchemeng.2017.10.008.

[51] M. Jain E. Katz-Bassett R. Govindan K. Zarifis M. Holland. "Modeling HTTP/2 Speed from HTTP/1 Traces". In: 2016, pp. 233–247. DOI: 10.1007/978-3-319-30505-9_18. URL: http://link.springer.com/10.1007/978-3-319-30505-9{_}18.

[52] E.A. Kemp. “Problems in expert systems development”. In: *Proceedings 1993 The First New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*. IEEE Comput. Soc. Press, pp. 166–167. ISBN: 0-8186-4260-2. DOI: 10.1109/ANNES.1993.323053. URL: http://ieeexplore.ieee.org/document/323053/.

[53] J. Kingston. “Pragmatic KADS: a methodological approach to a small knowledge-based systems project”. In: *Expert Systems* 9.4 (1992), pp. 171–180. ISSN: 0266-4720. DOI: 10.1111/j.1468-0394.1992.tb00399.x. URL: http://doi.wiley.com/10.1111/j.1468-0394.1992.tb00399.x.

[54] R.M.E. Salas J.B. Ewen-R. Allen S.V. Sarma K.M. Gunnarsdottir C.E. Gamaldo. “A novel sleep stage scoring system: combining expert-based rules with a decision tree classifier”. In: *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE. 2018, pp. 3240–3243.

[55] *Kubernetes - Production-Grade Container Orchestration*. 2017. URL: https://kubernetes.io/ (visited on 05/04/2017).

[56] S. Łaskawiec. “The Evolution of Java Based Software Architectures”. In: *Columbia International Publishing Journal of Cloud Computing Research* 2.1 (2016), pp. 1–17. URL: http://paper.uscip.us/jccr/JCCR.2016.1001.pdf.

[57] *Linux Programmer’s Manual - Capabilities*. URL: http://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 01/30/2020).

[58] *Linux Programmer’s Manual - CGroups*. URL: http://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 01/30/2020).

[59] *Linux Programmer’s Manual - Namespaces*. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 01/30/2020).

[60] T. Jirsik P. Celeda M. Husák M. Cermak. “Network-based HTTPS Client Identification Using SSL/TLS Fingerprinting”. In: Aug. 2015. DOI: 10.1109/ARES.2015.35.

[61] P. Clements M. Shaw. “The golden age of software architecture”. In: *IEEE Software* 23.2 (2006). ISSN: 0740-7459. DOI: 10.1109/MS.2006.58.

[62] J. Bartos W. Dyczka-K. Królikowska M. Wasilewski W. Krysicki. *Rachunek prawdopodobieństwa i statystyka matematyczna w zadaniach cz. 2*. PWN, 1999. ISBN: 83-01-11384-7.

[63] A. Brandon V. Muntes-Mulero D. Carrera M. Zasadzinski M. Sole. “Next Stop ’NoOps’: Enabling Cross-System Diagnostics Through Graph-Based Composition of Logs and Metrics”. In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC*. 2018. ISBN: 9781538683194. DOI: `10.1109/CLUSTER.2018.00039`. arXiv: `1809.07687`.

[64] R.I. Tutueanu V. Cristea-J. Kołodziej M.A. Vasile F. Pop. “Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing”. In: *Future Generation Computer Systems* 51.Supplement C (2015). Special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems, pp. 61–71. ISSN: 0167-739X. DOI: `10.1016/j.future.2014.11.019`. URL: `http://www.sciencedirect.com/science/article/pii/S0167739X14002532`.

[65] Marko Lukša. *Kubernetes in Action*. Manning Publications, 2017. ISBN: 9781617293726.

[66] *Minikube Project*. URL: `https://github.com/kubernetes/minikube` (visited on 02/24/2020).

[67] P. Miziuła. *Machine Learning training*. 2018.

[68] A. Goichot I. Chrisment M.M. Shbair T. Cholez. “Efficiently bypassing SNI-based HTTPS filtering”. In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*. 2015. ISBN: 9783901882760. DOI: `10.1109/INM.2015.7140423`.

[69] *Monthly Configuring the server topics - Keycloak*. URL: `https://keycloak.discourse.group/c/configuring-the-server/6/l/top/monthly` (visited on 12/13/2019).

[70] F. Pop V. Cristea N. Bessis S. Sotiriadis. “Using a novel message-exchanging optimization (MEO) model to reduce energy consumption in distributed systems”. In: *Simulation Modelling Practice and Theory* 39.Supplement C (2013). S.I.Energy efficiency in grids and clouds, pp. 104–120. ISSN: 1569-190X. DOI: `10.1016/j.simpat.2013.02.003`. URL: `http://www.sciencedirect.com/science/article/pii/S1569190X13000191`.

[71] M. Wolfthal N. Maurer. *Netty in action*. ISBN: 9781617291470. URL: `https://www.manning.com/books/netty-in-action`.

[72] *Name-based Virtual Host Support - Apache HTTP Server Version 2.2.* URL: https://httpd.apache.org/docs/2.2/vhosts/name-based.html (visited on 05/21/2019).

[73] *Netty web page.* 2018. URL: https://netty.io/ (visited on 08/10/2018).

[74] *nginx.* URL: https://nginx.org/en/ (visited on 05/29/2017).

[75] *Open vSwitch: QinQ Performance - Red Hat Developer Blog.* URL: https://developers.redhat.com/blog/2017/06/27/open-vswitch-qinq-performance/ (visited on 05/21/2019).

[76] *OpenDataHub · OpenDataHub.* URL: https://opendatahub.io/ (visited on 12/13/2019).

[77] *OpenJDK: jmh.* URL: https://openjdk.java.net/projects/code-tools/jmh/ (visited on 03/08/2019).

[78] *OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes.* URL: https://www.openshift.com/ (visited on 05/04/2017).

[79] *OpenShift SDN - Additional Concepts | Architecture | OpenShift Enterprise 3.0.* URL: https://docs.openshift.com/enterprise/3.0/architecture/additional{_}concepts/sdn.html (visited on 03/07/2019).

[80] *OpenSSL 1.0.2 release notes.* 2018. URL: https://www.openssl.org/news/openssl-1.0.2-notes.html (visited on 08/10/2018).

[81] *operator-framework/operator-sdk: SDK for building Kubernetes applications. Provides high level APIs, useful abstractions, and project scaffolding.* URL: https://github.com/operator-framework/operator-sdk (visited on 12/10/2019).

[82] S. Perera. *Introduction to Anomaly Detection: Concepts and Techniques | My views of the World and Systems.* 2015. URL: https://iwringer.wordpress.com/2015/11/17/anomaly-detection-concepts-and-techniques/ (visited on 08/02/2019).

[83] C. Pinhanez. “Machine Teaching by Domain Experts: Towards More Humane, Inclusive, and Intelligent Machine Learning Systems”. In: *arXiv preprint arXiv:1908.08931* (2019).

[84] *Principal component analysis.* URL: https://en.wikipedia.org/wiki/Principal_component_analysis (visited on 02/14/2020).

[85] *Prometheus - Monitoring system and time series database*. URL: https://prometheus.io/ (visited on 03/04/2020).

[86] *Quarkus - Supersonic Subatomic Java*. URL: https://quarkus.io/ (visited on 02/28/2020).

[87] S. Paul R. Jain. *Network virtualization and software defined networking for cloud computing: A survey*. 2013. DOI: 10.1109/MCOM.2013.6658648. arXiv: 1509.07675.

[88] W. Browne R. Urbanowicz. *Introducing Rule-Based Machine Learning: A Practical Guide*. Tech. rep. 2015. URL: ryanurbanowicz.com/wp-content/uploads/2016/09/Urbanowicz{_}Browne{_}2015{_}Introducing-Rule-Based-Machine-Learning-A-Practical-Guide-GECCO15-CRC-Copy.pdf.

[89] *RAC - Reliable Asynchronous Clustering Design*. URL: https://github.com/infinispan/infinispan-designs/blob/master/RAC:-Reliable-Asynchronous-Clustering.asciidoc (visited on 03/04/2020).

[90] A. Randal. *The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers*. Tech. rep. arXiv: 1904.12226v1.

[91] *ReleaseNotes/Austin - OpenStack*. URL: https://wiki.openstack.org/wiki/ReleaseNotes/Austin (visited on 08/20/2015).

[92] J. Reschke R.T. Fielding. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. 2014. DOI: 10.17487/RFC7230. URL: https://rfc-editor.org/rfc/rfc7230.txt.

[93] R.N. Taylor R.T. Fielding. *Principled Design of the Modern Web Architecture*. Tech. rep. 2. 2000, pp. 115–150. URL: https://www.ics.uci.edu/{~}taylor/documents/2002-REST-TOIT.pdf.

[94] M. Khmakhem S. Amamou Z. Trifa. “Data protection in cloud computing: A Survey of the State-of-Art”. In: *Procedia Computer Science* 159 (2019). Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019, pp. 155–161. ISSN: 1877-0509. DOI: 10.1016/j.procs.2019.09.170. URL: http://www.sciencedirect.com/science/article/pii/S1877050919313493.

[95] C. Bird R. Deline-H. Gall E. Kamar N. Nagappan B. Nushi T. Zimmermann S. Amershi A. Begel. *Software Engineering for Machine Learning: A Case Study*. Tech. rep. URL: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/.

[96] A. Langley E. Stephan S. Friedl A. Popov. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*. RFC 7301. 2014. DOI: 10.17487/RFC7230. URL: https://rfc-editor.org/rfc/rfc7301.txt.

[97] M. Choraś S. Łaskawiec. "Considering service name indication for multi-tenancy routing in cloud environments". In: *Advances in Intelligent Systems and Computing*. Vol. 525. 2017, pp. 271–278. ISBN: 9783319472737. DOI: 10.1007/978-3-319-47274-4_33.

[98] M. Choraś S. Łaskawiec. "New Solutions for exposing Clustered Applications deployed in the cloud". In: (2018).

[99] Y. Chen H. de Saxce I. Oprescu. "Is HTTP/2 really faster than HTTP/1.1?" In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2015, pp. 293–299. ISBN: 978-1-4673-7131-5. DOI: 10.1109/INFCOMW.2015.7179400. URL: http://ieeexplore.ieee.org/document/7179400/.

[100] P. Pintelas S.B. Kotsiantis I. Zaharakis. "Supervised machine learning: A review of classification techniques". In: *Emerging artificial intelligence applications in computer engineering* 160 (2007), pp. 3–24.

[101] M. Sciabarra. *LEARNING APACHE OPENWHISK : developing open serverless solutions*. O'REILLY MEDIA, INC, USA, 2019. ISBN: 9781492046165. URL: https://learning.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/.

[102] *SELinux Project Wiki*. URL: https://selinuxproject.org/page/Main_Page (visited on 01/30/2020).

[103] *sklearn.tree.DecisionTreeClassifier — scikit-learn 0.22 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html (visited on 12/11/2019).

[104] *Spanning JGroups Kubernetes-based clusters across Google and Amazon clouds*. URL: http://belaban.blogspot.com/2019/12/spanning-jgroups-kubernetes-based.html (visited on 03/04/2020).

[105] StackRox. *The State of Container and Kubernetes Security*. Tech. rep. 2020.

[106] *Submariner project*. URL: https://github.com/submariner-io/submariner (visited on 01/31/2020).

[107] *Swarm v. Fleet v. Kubernetes v. Mesos - Swarm v. Fleet v. Kubernetes v. Mesos [Book]*. URL: https://www.oreilly.com/library/view/swarm-v-fleet/9781492028819/ch01.html (visited on 01/31/2020).

[108] *Swarm v. Fleet v. Kubernetes v. Mesos - Swarm v. Fleet v. Kubernetes v. Mesos [Book]*. URL: https://trends.google.com/trends/explore?q=Kubernetes,Docker%20swarm,mesos (visited on 01/31/2020).

[109] R. Szumski. *Top Kubernetes Operators advancing across the Operator Capability Model*. URL: https://blog.openshift.com/top-kubernetes-operators-advancing-across-the-operator-capability-model/ (visited on 02/24/2020).

[110] M. Fischetti T. Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Paw Prints, 2008, p. 272. ISBN: 1439500363. URL: https://books.google.com/books?id=Unp4PwAACAAJ{\&}pgis=1.

[111] B. Schiele T. Hunh. "Unsupervised Discovery of Structure in Activity Data Using Multiple Eigenspaces". In: *Location- and Context-Awareness*. Ed. by Mike Hazas, John Krumm, and Thomas Strang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, 151–167. ISBN: 978-3-540-34151-2.

[112] P. Nauduri T. Laszewski. *Migrating to the cloud : Oracle client/server modernization*. Syngress, 2012. ISBN: 9781597496476.

[113] R.L. Rivest C. Stein T.H. Cormen C.E. Leiserson. *Introduction to Algorithms Second Edition*. The MIT Press, Cambridge, Massachusetts London, England, 2001. ISBN: 0-262-03293-7. URL: http://web.karabuk.edu.tr/hakankutucu/CME222/MIT[1].Press.Introduction.to.Algorithms.2nd.Edition.eBook-TLFeBOOK.pdf.

[114] *The netfilter.org project web page*. 2018. URL: https://www.netfilter.org/ (visited on 07/28/2018).

[115] *The Open Market Internet Index 1995*. URL: http://www.treese.org/intindex/95-11.htm (visited on 08/20/2015).

[116] *TLS Handshake Protocol - Windows applications | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/windows/desktop/SecAuthN/tls-handshake-protocol (visited on 03/08/2019).

[117] *topic-3-decision-trees-and-knn*. URL: https://mlcourse.ai/articles/topic3-dt-knn/ (visited on 12/09/2019).

[118] *Tree Boosting With XGBoost — Why Does XGBoost Win "Every" Machine Learning Competition?* URL: https://medium.com/syncedreview/tree-boosting-with-xgboost-why-does-xgboost-win-every-machine-learning-competition-ca8034c0b283 (visited on 12/09/2019).

[119] P.S. Yu V. Cardellini M. Colajanni. *Dynamic load balancing on web-server systems*. 1999. DOI: 10.1109/4236.769420.

[120] A. K. Md. Ehsanes. Saleh V. K. Rohatgi. *An introduction to probability and statistics*. ISBN: 9781118799642. URL: https://learning.oreilly.com/library/view/an-introduction-to/9781118799642/.

[121] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17.

[122] R. Rajamony J. Rubio W. Felter A. Ferreira. "An updated performance comparison of virtual machines and Linux containers". In: *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software*. 2015. ISBN: 9781479919567. DOI: 10.1109/ISPASS.2015.7095802.

[123] Y. Leung W.K. Wong Z. Guo. *Optimizing decision making in the apparel supply chain using artificial intelligence (AI) : from production to retail*. Woodhead Publishing Ltd, 2013, p. 231. ISBN: 9780857097842. URL: https://learning.oreilly.com/library/view/optimizing-decision-making/9780857097798/.

[124] *XGBoost Documentation — xgboost 1.0.0-SNAPSHOT documentation*. URL: https://xgboost.readthedocs.io/en/latest/ (visited on 12/09/2019).

[125] S.J. Fenves Y. Reich. "The potential of machine learning techniques for expert systems". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 3.3 (1989), pp. 175–193. ISSN: 0890-0604. DOI: 10.1017/S0890060400001219. URL: https://www.cambridge.org/core/product/identifier/S0890060400001219/type/journal{_}article.

[126] *Zero-downtime Deployment in Kubernetes with Jenkins*. URL: `https://kubernetes.io/blog/2018/04/30/zero-downtime-deployment-kubernetes-jenkins/` (visited on 03/04/2020).

Abstract

Effective solutions for high performance communication in the cloud

Typical modern application architectures are based on three basic pillars - a datastore, stateless services and web pages. Deploying high-performance, stateful systems, such as distributed caches or data grids has always been challenging. Such applications require additional maintenance tasks such as monitoring system capacity, configuring backups or testing disaster recovery. The author proposed four solutions that help with addressing a few of the problems by lowering the server memory footprint by enabling multi-tenancy, improving client-server communication performance by using custom binary protocols along with enabling client side load balancing, and finally, introducing an expert system to automatically spot common application configuration mistakes. All the proposed solutions help to increase the overall system throughput and find configuration mistakes in an automated fashion.

Keywords: *Cloud, Kubernetes, Data Grid*

Streszczenie

Efektywne rozwiązania dla wysokowydajnej komunikacji w chmurach

Wiele nowoczesnych systemów informatycznych opartych jest o trzy podstawowe komponenty - bazę danych, bezstanowe komponenty realizujące logikę biznesową oraz stronę internetową, która jest interfejsem użytkownika. Komponenty chmur opartych o kontenery wspierają ten model wytwarzania systemów. Istnieje również grupa aplikacji wymykających się wcześniej wspomnianemu podejściu - są to systemy przechowujące dane, w tym systemy typu "Data Grid". Systemy te wymagają wykonywania dodatkowych czynności przy utrzymaniu systemu, takich, jak monitorowanie obciążenia systemu, konfiguracja wykonywania kopii zapasowych. Autor zaproponował cztery rozwiązania, które rozwiązują część z wcześniej wymienionych problemów poprzez: obniżenie ilości konsumowanej pamięci przez serwer systemu "Data Grid" (wykorzystując obsługę wielu aplikacji klienckich), zwiększając przepustowość pomiędzy aplikacją kliencką, a serwerem wykorzystując binarne protokoły komuikacyjne oraz technikę "client side load balancing" oraz przedstawiając system ekspercki automatycznie znajdujący typowe błędy konfiguracyjne aplikacji. Wszystkie proponowane rozwiązania pomagają zwiększyć przepustowość syste mu oraz pomagają zidentyfikować w sposób automatyczny błędy w konfiguracji.

Słowa kluczowe: *Chmura, Chmura obliczeniowa, Kubernetes, Data Grid*