Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datavetenskap
2019 | LIU-IDA/LITH-EX-A--19/037--SE

Evaluation and comparison of a RabbitMQ broker solution on Amazon Web Services and Microsoft Azure

Evaluering och jämförelse av en RabbitMQ broker-lösning på Amazon Web Services och Microsoft Azure

Sebastian Lindmark
Andreas Järvelä

Supervisor : Rouhollah Mahfouzi
Examiner : Petru Eles

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

In this thesis, a scalable, highly available and reactive RabbitMQ cluster is implemented on Amazon Web Services (AWS) and Microsoft Azure. An alternative solution was created on AWS using the CloudFormation service. These solutions are performance tested using the RabbitMQ PerfTest tool by simulating high loads with varied parameters. The test results are used to analyze the throughput and price-performance ratio for a chosen set of instances on the respective cloud platforms. How performance changes between instance family types and cloud platforms is tested and discussed. Additional conclusions are presented regarding the general performance differences in infrastructure between AWS and Microsoft Azure.

Acknowledgments

First we would like to thank Cybercom Sweden AB for the opportunity to make our master's thesis. Furthermore we would like to send a special thank you to our supervisors at Cybercom, Jesper Ahlberg and Esbjörn Blomquist, for always being engaged in our work and providing us with endless support. We also thank Petru Eles, our examinator at Linköping University for giving us feedback during meetings and for providing valuable discussions regarding the thesis.

We would also like to thank Innovation Zone, more specifically the other master's thesis students at Cybercom for a cooperative and positive atmosphere. Thank you to Christian Wångblad for all the jokes and most importantly for inspiring us to do our best.

Thank you to our friends for all the laughs and support and a special thank you to our families for the support and encouragement during our university studies.

Lastly, we would like to thank each other for many laughs and a great cooperation, not only during our master's thesis but also for the entire time at the university.

Contents

Abstract iii

Acknowledgments iv

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Aim . . . 2
- 1.3 Research questions . . . 2
- 1.4 Delimitations . . . 3

2 Background 4
- 2.1 Cloud Computing . . . 4
- 2.2 Message broker . . . 4
- 2.3 AMQP . . . 5
- 2.4 PerfTest . . . 5
- 2.5 Virtual CPU . . . 6
- 2.6 Load Balancer . . . 6
- 2.7 Virtualization . . . 7
- 2.8 Related Work . . . 7

3 Method 11
- 3.1 Broker Architecture . . . 11
- 3.2 RabbitMQ setup . . . 12
- 3.3 Automatic RabbitMQ clustering . . . 13
- 3.4 Amazon Web Services . . . 14
- 3.5 Microsoft Azure . . . 19

4 Comparison between AWS and Azure 21
- 4.1 Setup . . . 21
- 4.2 Result . . . 24
- 4.3 Discussion . . . 29

5 Comparison between AWS and network accelerated Azure **31**
5.1 Result 34
5.2 Discussion 41

6 Performance analysis of compute optimized instances **42**
6.1 Testing 42
6.2 Result 45
6.3 Discussion 47

7 Cost model analysis of RabbitMQ instances **49**
7.1 Cost modelling 49
7.2 Result 51
7.3 Discussion 53

8 General Discussion **54**

9 Conclusion **56**
9.1 Future Work 58

Bibliography **59**

List of Figures

3.1 Broker cluster of three nodes connected to a load balancer 12
3.2 Illustrates the use of Security Groups 14
3.3 RabbitMQ container copying external ip 16
3.4 The architecture of an auto scaling broker solution 17

4.1 AWS high-tier cluster performance with 150byte package size 25
4.2 Azure high-tier cluster with 150byte package size 25
4.3 AWS low-tier performance with 150byte package size 26
4.4 Azure low-tier performance with 150byte package size 26
4.5 AWS high-tier performance with 2 MB package size 27
4.6 Azure high-tier performance with 2 MB package size 27
4.7 AWS low-tier performance with 2 MB package size 28
4.8 Azure low-tier with 2 MB package size 28

5.1 Azure server architecture without network acceleration 32
5.2 Azure server architecture with network acceleration 32
5.3 Azure A4m_v2 (high-tier) performance with network acceleration disabled. 35
5.4 Azure D2_v2 (low-tier) performance with accelerated networking enabled 35
5.5 AWS high-tier cluster performance with 150 byte package size 36
5.6 Azure high-tier cluster performance with 150 byte package size 37
5.7 AWS low-tier cluster performance with 150byte package size 38
5.8 Azure low-tier cluster performance with 150byte package size 38
5.9 AWS high-tier cluster performance with 2MB byte package size 39
5.10 Azure high-tier cluster performance with 2MB package size 39
5.11 AWS low-tier cluster performance with 2MB package size. 40
5.12 Azure low-tier cluster performance with 2MB package size 40

6.1 AWS compute optimized VM performance 45
6.2 Azure compute optimized VM performance 45
6.3 AWS general purpose m5 instance family performance 46
6.4 AWS general purpose t3 instance family performance 46

List of Tables

3.1 Port access needed for a RabbitMQ server 15
3.2 Description of AWS instance family types 15
3.3 Load Balancer port forwarding 17

4.1 Tests performed using PerfTest 21
4.2 Azure specifications for the high and low tier clusters 22
4.3 AWS specifications for the high and low tier clusters 22
4.4 AWS specifications for the virtual machine used to perform tests 23
4.5 Azure specifications for the virtual machine used to perform tests 23

5.1 Specifications for the Azure high and low-tier cluster VMs 33
5.2 Specifications for the AWS high and low-tier cluster VMs 33
5.3 AWS specifications for the VM used to perform tests 34
5.4 Azure specifications for the VM used to perform tests 34

6.1 Tests performed to simulate high loads using PerfTest 43
6.2 The specification of the virtual instances running RabbitMQ 43
6.3 The specification of the general type instances on AWS 44
6.4 Measured bandwidth on Azure VMs 44

7.1 Low cost instances 49
7.2 Medium cost instances 50
7.3 High cost instances 50
7.4 Instance price-performance evaluation for 1 queue 51
7.5 Instance price-performance evaluation for 2 queues 51
7.6 Instance price-performance evaluation for 10 queues 52
7.7 Instance price-performance evaluation for 200 queues 52

1 Introduction

Cloud computing has grown rapidly to become an alternative to on-premise cluster solutions. It provides customers with computational services and the delivery of on-demand computational power. Before the rise of cloud platforms, an option was to implement your own solution. However, managing your own server cluster can be costly as it requires maintenance and knowledge of how to construct a distributed system. New cloud computing services are continuously added to support new trends in computer science, like Internet of Things (IoT) frameworks and hardware specialized for machine learning algorithms. One of the first cloud platforms was Amazon Web Services (AWS), provided by Amazon in 2006. Since then multiple tech giants like Google and Microsoft has followed the trend [1]. As a result these three now account for around 55% of the total market share [6]. Today every company can get access to immense compute power for a reasonable price without having to buy expensive hardware. Except for performance benefits, cloud computing comes with a lot of features and quality attributes regarding availability, scalability and elasticity. As network traffic grows constantly, the need for reliable servers able to withstand high traffic loads and react to traffic fluctuations is a key factor in providing a robust service.[26] These are some of the features that describe the selling points of the cloud services available today. However only a part of the complexity is abstracted by migrating your business. As mentioned earlier, the amount of services within a cloud platform are increasing and today there exists multiple ways of deploying the same solution onto the cloud.

Message brokers is a way to minimize the dependencies of an application. They are used for inter-communication between internal services in order to achieve asynchronous communication, to support loose coupling and separation of concerns. Having this type of separation allows developers to seamlessly create modular solutions. When companies develop a service, an important part is to make sure the

[1] `https://en.wikipedia.org/wiki/Cloud_computing`

system works regardless of internal failures. Some message brokers like RabbitMQ support persistent message delivery that stores messages for parts of the system that are temporarily down. [24]. This way a message persistence is achieved and once the services are back online they will receive the latest data and thereby always maintain a synchronized state. Because of message brokers taking the central role in distributing messages between modules there are two important quality attributes to consider, scalability and availability. With cloud computing emerging, moving the brokers to the cloud seems like a natural step to acquire these qualities.

This study will look at a manual RabbitMQ solution deployed on AWS and Azure to compare and evaluate their differences in performance and cost.

1.1 Motivation

The deployment approach that this paper considers is the manual deployment. For companies, using the Broker as a Service (BaaS) solution might seem like the option with the least setup time. There are, however, limitations to this solution in the form of a predefined message broker that it is not reusable over different cloud platforms. Companies want a solution that is cheap and reusable over several clouds. In addition, performance and ease of implementation are important factors. Some comparisons that have been performed today are between different brokers or the used architecture with respect to high availability and performance. There have also been comparisons of the costs of using different cloud platforms and what type of usage that is cheaper e.g. computational or bandwidth heavy [20] [16].

1.2 Aim

This thesis aims to evaluate and compare the performance of a manual deployment of RabbitMQ on AWS and Azure. A second implementation on AWS will be done using the CloudFormation service. The chosen architecture for the message broker solution is a mesh-network consisting of three nodes. Performance will be measured by messages per second (msgs/s). A testing tool called PerfTest designed for performance testing RabbitMQ servers will be used to benchmark the broker solutions. The goal is to define performance and cost differences between AWS and Azure. A price-performance model will be created to help decide which platform and instance that is best suited under certain circumstances.

1.3 Research questions

This study will answer these research questions:

1. How does the choice of cloud platform and deployment architecture affect the broker performance, in terms of throughput?

2. How does the choice of instance type affect the broker performance within and between cloud platforms, in terms of throughput?

3. How does the choice of cloud platform and respective instance type relate to the price-performance ratio?

1.4 Delimitations

This study is done in cooperation with Cybercom and based on the requirements from the company. No protocol other than AMQP 0-9-1 will be considered. RabbitMQ will be the sole message broker used. No cloud platform other than AWS and Azure will be considered. A limited set of instance types and instance family types will be evaluated. Instance family type comparisons will only be done on AWS. Performance measurement will only be performed from within the cloud environments.

2 Background

In this chapter relevant background theory and related work will be presented to give the reader relevant information and to support the aim of the study.

2.1 Cloud Computing

The introduction of cloud computing has changed the way traditional businesses designs and develop IT-systems. There are many benefits of using a cloud platform for a service such as cost savings, speed and world wide deployment. Companies do no longer need to buy large server-racks and can instead use the automatic infrastructure management systems available at cloud platforms. Infrastructure as a Service (IaaS) is an example of a cloud service provided by some cloud platforms along with Software as a service (SaaS), Platform as a Service (PaaS) and serverless. IaaS is the standard way of using a cloud computing platform, where virtual machines (VMs) can be rented and used as your own remote computer. Often VMs for specific use cases are available, ranging from low cost testing machines to high-performance computers. [30] [29]

2.2 Message broker

Message brokers are used for asynchronous message passing between clients. Messages are passed to the message broker and then distributed to the clients. Message brokers are used to integrate distributed components to provide a communication channel between them within an application or service. Message queues are an important part of message brokers. Since not all services are interested in the same data, queues can be used to separate data flows. When a client publishes data to a broker, a queue has to be specified. This way the broker can send data to all clients subscribed to a queue. Among with others, AMQP and MQTT are protocols that support message queues. MQTT is a more lightweight protocol that is designed to be used

without as much overhead. It is a good choice of protocol if the data is coming from i.e. IoT devices or mobile phones. The AMQP protocol has more overhead and supports several broker design patterns such as publish/subscribe, request/reply and producer/consumer.

2.3 AMQP

The selected message broker RabbitMQ implements AMQP as a communication protocol. For a message to be delivered to a receiver, it has to pass through an exchange which works as the message router. The exchange module passes the message to the intended queue and through the queue the message is received by the consumers. Message acknowledges are used to ensure all messages are delivered to their intended destination. This feature in AMQP makes sure message brokers can provide a fault tolerant and reliable transmission between two destinations [1]. AMQP provides two acknowledgement alternatives, either automatic or explicit acknowledgement. With automatic acknowledgement messages are considered acknowledged as soon as they are sent, whereas with explicit acknowledgment the consumer decides when to acknowledge the message e.g. when received or processed. The former alternative allows for a higher data throughput in exchange for reduced safety of delivery [8]. Communication with AMQP is done through channels where a channel can be seen as a lightweight connection. Opening multiple TCP connections will consume resources so AMQP implements a way of multiplexing channels over a single TCP connection. This way channels can simulate multiple connections using a low amount of resources. A channel ID is used in the protocol for the client and server to keep track of which channel the data belongs to. [1]

RabbitMQ

RabbitMQ is a popular open source message broker. It offers a lightweight solution that is easy to deploy anywhere and support protocols such as AMQP and MQTT. [24] RabbitMQ comes with a management tool called rabbitmqctl that can be used for configuration of an existing RabbitMQ server. A management UI is found on a running RabbitMQ server by accessing port 15672 on any browser. The UI shows various statistics such as an overview of current throughput, connected clients, queues, exchanges and more. This way the management UI can be used to either configure queues, exchanges or to analyze performance of a running RabbitMQ server to help find differences in workload. RabbitMQ also comes with automatic support for clustering. This means that using the rabbitmqctl tool nodes can join a cluster and achieve high-availability.

2.4 PerfTest

PerfTest is a Java based testing tool created by the RabbitMQ team, used to test throughput for RabbitMQ servers. The tool allows for high customizability and can simulate different kinds of network scenarios, e.g. multiple producers and queues, varying packet sizes and distributed producers. The most basic test is started by supplying an IP to the broker. Optional flags are used for customization of the tool and

the traffic to send. Throughput statistics are output during tests to the console and statistics for the finished test can later be exported to files. Metrics that are collected and measured each second during the test are send rate (msgs/s), receive rate (msgs/s) and latency (min/median/75th/95th/99th) µs. These values are based on the messages sent during a period of 1 second. A full summary is given for a finished test where an average sending and receive rate is calculated by the tool. [23]

2.5 Virtual CPU

Virtual CPUs (vCPUs) are virtual processors running in VMs. A thread on the host machine can simulate one vCPU on the VM. AWS and Azure provide different families of machines depending on the area of application, e.g. general purpose and compute optimized family. For each of the families a certain type of processor is used on the host machine. [27]

2.6 Load Balancer

A load balancer distributes traffic to multiple resources, like VM pools, containers and IP addresses. Load balancers can either be internal-facing within a virtual private cloud (VPC) or internet-facing exposed to the public. On AWS multiple external load balancers exist depending on the usage of the application. Application load balancers are optimized to handle HTTP/HTTPS traffic, whereas network load balancers are used for TCP/SSL traffic. The classic load balancer can handle both HTTP and TCP/SSL traffic [21]. Azure has one external load balancer available in two stock keeping units (SKUs), basic and standard. The standard SKU is an extended version of the basic SKU, where features such as increased backend pool size, outbound rules and availability-zone redundancy are included. No difference in performance between the two SKUs are listed. [28] The load balancer support a dynamic port mapping, where incoming traffic on port X can be forwarded to a destination IP on port Y.

There are many benefits of using a load balancer in a system. One of them is the possibility to automatically distribute workload across multiple servers within a cluster. A routing algorithm is used to decide which server to forward traffic to. The routing algorithm can either be dependent on an internal state i.e. a round-robin implementation or/and make decisions based on information provided by servers within the cluster i.e forward traffic to the server with lowest current CPU usage. An AWS network load balancer uses a "flow hash algorithm" as the routing algorithm. This algorithm is based on a hash composed of source IP, source port, destination IP, destination port and TCP sequence number [2]. The Azure load balancer uses the same hashing technique as AWS, a 5-tuple hash composed of the same components. [3]
Another benefit of using a load balancer is to provide fault tolerance and high availability to a system. The load balancer can, with health checks, determine whether to forward traffic to a server or not depending on the state of the server. A server can go offline and the system will still be responsive as traffic is routed to a healthy server. Using a load balancer in conjunction with automatic scaling is a way to make your system reactive, in terms of scaling up/down the cluster depending on the workload.

The cluster is then able to react accordingly to changes in traffic and both provide performance and cost savings during high/low workloads. [21]

2.7 Virtualization

Virtualization is the concept of creating virtualized computation environments on a single physical machine. A virtualized machine (VM) uses emulated hardware and does therefore not need dedicated physical hardware components and is instead provided with the emulated hardware it was configured to use [5]. The characteristics of VMs are the following:

Partitioning

- One host machine can run multiple VM instances running different OSs.
- Resources of the host machine are shared between the VMs [31].

Isolation

- Processes can be run separately on isolated VMs, providing fault tolerance and high availability [31].

Encapsulation

- The state of the VM is stored as a file, making them highly transferable and movable between different host machines [31].

These characteristics have during the last years enabled cloud platforms to provide VMs on demand, it is called IaaS.

Docker

Docker is an operating system-level virtualization software used for building and deploying applications locally or in cloud environments. Docker containers encapsulate applications and their dependencies in order to provide a virtualized environment independent of the host OS. This guarantees that the environment and all the dependencies for the application do not change, no matter where and what host OS the application is hosted on.

2.8 Related Work

This section will present relevant information for the reader. Theory on message brokers, cloud platforms and performance comparisons are included for this chapter. The information provided will support the aim for this thesis.

Scaling/Elasticity

Workload is difficult to estimate and therefore most companies use scalable and elastic cloud services. Gascon-Samson et al. [14] conducted a study on a software based middleware to handle scaling and elasticity for a channel-based pub/sub framework called Redis. The authors mentioned two areas of the middleware that need scaling, system-level and channel-level. At system-level the middleware could add or remove pub/sub servers depending on current server load. At channel-level the middleware could replicate a channel by distributing the same subscription to another server, decreasing the workload of that channel.
Coutinho et al. [9] conducted a survey on cloud computing and summarized the different elastic approaches used. The authors mention three methods:

- Horizontal scaling is a commonly used method. It works by adding or removing resources (i.e. servers or virtual environments) to improve performance. Once a new server has been added, it can take jobs from overloaded servers.
- Vertical scaling refers to the dynamic resource allocation of a VM. The authors mention resizing and replacement as two different methods to handle resources of VMs.
- Migration refers to the method of moving a virtual server instance to a different server to balance the workload.

In order for the system to determine when to use the different methods the authors also mention two of the most common models used:

- A reactive model uses thresholds to detect when additional resources need to be allocated. The thresholds can apply to anything related to workload.
- The proactive model is used for the same purpose as the reactive model but it detects when additional resources need to be allocated by predicting the workload. This is usually done by looking at the load-history.

Performance

Ionescu [17] conducted a research study about performance analysis between the two most popular open-source brokers, ActiveMQ and RabbitMQ. The research showed their respective advantages, and the situations where they are best applied. The research was carried out by setting up an isolated system consisting of a client, queue and a server. Scaling properties were not taken into consideration in this setup but instead the experiments focused on checking the raw performance of the two brokers. The results are of relevance as they show different use cases for each broker. ActiveMQ had a throughput of around 1.5 times more when producing data, making it a more suitable choice over RabbitMQ for applications where lots of data is fed from the system. It was, however, concluded that RabbitMQ was the best choice when feeding clients with data, being as much as 50 times more effective than ActiveMQ.

AWS EC2

Comparing different cloud platforms

The top cloud platforms all provide a default set of functionality. Therefore a problem when selecting which platform to use is to make sure the performance requirements are met. Dordevic et al. [13] compared the performance between two of the top cloud platforms, AWS and Microsoft Azure. For this performance test they used a Linux VM with similar specifications on each of the platforms. The biggest difference between the Azure and the AWS VM was the processor frequencies, running at 2.19 GHz and 1.8 GHz respectively. They did not mention any reason behind this difference or if two machines with the same specifications could have been selected instead. The performance analysis was done using the benchmark tool Phoronix Test Suite3. Results showed that Azure performed better which probably were because of the more powerful CPU. AWS did, however, perform slightly better than Azure in a disk performance test. Except for performance scores the authors concluded that AWS offers machines with better specifications for the same cost. They also noticed in the setup process that AWS provided more customizability and tuning of their VMs, making VM-optimizations available within the cloud.

Kotas et al. [20] also concluded that AWS provides more computational power for the same cost. They conducted a test where they measured the CPU-performance and bandwidth-speed for two VM-instances on AWS and Azure. A cost effectiveness score was made by combining performance measurements with cost for the respective VM-instance. They concluded that an AWS instance was a cheaper choice when running computationally heavy operations on the CPU. In contrast, an Azure instance was cheaper in terms of bandwidth usage.

There are multiple aspects when deciding which service and cloud provider to choose. Except for benchmark tests and performance, cost and price models can be at least an equally important factor. Hyseni and Ibrahimi [16] compared AWS and Google Cloud by looking at services and prices per instance. The results showed that Amazon offered a greater number of services for their customers to choose from. Conversely, Google Cloud had the lowest price per instance. There is therefore a trade-off between these two when selecting the cloud provider.

Message queuing

Jutadhamakorn et al. [19] proposed in their paper a system for a scalable and low-cost cluster using message brokers and load balancing tools to automatically account for traffic fluctuations. The server-side setup consisted of a single computer running the web server NGINX, handling the load balancing and forwarding of MQTT-requests back and forth between clients and brokers. The cluster of brokers was orchestrated by the system Docker Swarm. By using Docker Swarm, communication was provided between connected nodes within the swarm, along with functionality to dynamically add new nodes to the swarm. The latter provided scalability properties to the system as well as an entry point for load balancing using NGINX. A comparison against a system consisting of a single message broker was done to show perfor-

mance advantages, where the proposed system performed up to eight times better in context of throughput and CPU-load. No performance comparisons were done between different types of brokers or load balancing servers. The components used for the purposed system are not available on the AWS and Azure platforms. The system architecture is however of interest for this study.

Alternatives to message queuing

The standardized way of communicating over the internet is via the HTTP protocol. However, as stated by Yokotani et al. [32] this protocol is not best suited for all applications due to a significant package overhead. They concluded that a more lightweight and effective alternative is needed when dealing with small energy efficient applications like IoT-devices. In their study they made a comparison between the HTTP and MQTT protocol by comparing the overhead and payload sizes when sending the same data over the two protocols. They showed that the MQTT protocol used far less overhead over HTTP, where the effect was especially noticeable when the number of sending devices increased. This was also concluded by Wankhede et al. [4] in a similar study where they also focused on the energy efficiency of the two protocols. Results showed that MQTT was more energy efficient because of the simpler setup/handshake procedure. In this thesis, MQTT is not a suitable protocol to use since it is more focused on lightweight transactions for i.e. IoT devices, and is not optimized for speed. RabbitMQ is not compatible with the HTTP protocol, therefore it will not be considered.

Broker architecture

There are different architectures regarding the master-slave design pattern to consider when working with message brokers. Rostanski et al. [25] conducted a study on the relation between performance and availability regarding different master-slave solutions. Using a single node architecture they found that their chosen message broker RabbitMQ had an average publish rate of about 33000 msgs/s. This means that a single master queue in a single node network offers high performance. Since RabbitMQ performs well as a single node the authors tested two other options with a broker mesh-network consisting of three nodes, offering availability. The first option which offered the highest availability and a significant reduction of performance was having all nodes of the three-node network to shadow a message queue from each neighbour. This way the message broker solution had an average publish rate of about 8500 msgs/s. They considered a third solution only having one slave to each master queue, calling this the N+1 architecture. This way availability was still achieved but with higher performance. This solution had the average publish rate of about 12600 msgs/s and was motivated to be a good choice if valuing availability as well as performance.

In our experiments, a three node cluster will be created with shadowing to two neighbouring nodes to achieve a high availability solution.

3 Method

This chapter describes the method of implementing a manual broker solution on AWS and Azure along with a template based implementation using CloudFormation. This chapter also describes the method of conducting performance tests and analysing the results.

3.1 Broker Architecture

The goal for the broker architecture was to replicate something that is used in enterprise solutions today. This way a comparison could be made between cloud platforms, regarding the performance of VMs and the difficulties of setting up a more advanced architectural structure. Like any other message broker, RabbitMQ can be run on a single server without any external architectural components. However, for enterprise solutions a high availability cluster is used to make sure that the broker is always accessible.

The chosen architecture for the broker solution was the high availability cluster consisting of three nodes. This is an enterprise solution for brokers that want to achieve high availability.

Since the consumers of a message broker do not have any information of how many active nodes there are or which are heavily loaded, a load balancer was needed. Having consumers of the service connect to a load balancer allows for automatic distribution of connections to the active message brokers. The load balancer also provides an availability factor, as forwarding of data is only done to healthy and accessible nodes. Figure 3.1 presents how the desired broker architecture looks like.

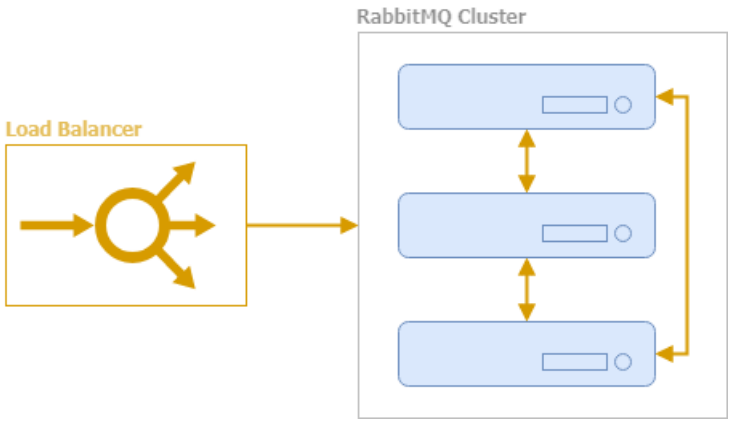


Figure 3.1: Broker cluster of three nodes connected to a load balancer

3.2 RabbitMQ setup

Before installing a RabbitMQ server, two different installation strategies were considered. The first option to manually install the RabbitMQ server would give a deeper understanding of all the dependencies that are required. Initially an Amazon Linux OS was installed on an EC2 instance. This was an arbitrarily chosen OS to be the base for a RabbitMQ server. However, installing a pure RabbitMQ server using the package manager "yum" was not possible since it required a "systemd" dependency. With further investigation a suggestion of CentOS 7 was found and came to be the operating system of choice as it comes with "systemd" pre-installed. By first installing the required packages, an Amazon Machine Image (AMI) could be created and used to start new VMs and create a RabbitMQ cluster. To avoid the risk of a cloud platform not supplying an OS with "systemd", a more general approach was taken.

The second alternative was to use Docker, that with a single CLI command deploys a RabbitMQ server. The end goal was to get a general solution for setting up a server that would work on multiple cloud platforms without any major changes. The Docker solution would work independently of the host OS, as long as it had support for the software. The Docker alternative was chosen as the method to run a RabbitMQ server.
To achieve a cluster, the nodes need to be aware of each other. One way to achieve this was to create a configuration file that consisted of all available hosts. But to allow for dynamic scaling this was not an option. The final method used for automatic clustering is described in section 3.3

3.3 Automatic RabbitMQ clustering

Algorithm 1: The pseudo-code used for automatic RabbitMQ clustering

```
1  host_ip = get_host_machine_IP()
2  start_rabbitmq_docker_container(host_ip, ports);
3  while docker container is starting do
4  |   nothing;
5  end
6  stop_node()
7  execute_docker_setup_commands();
8  execute_rabbitmq_setup_commands();
9  cluster nodes = get_host_nodes_from_loadbalancer(lb_ip);
10 for cluster nodes do
11 |   if node is online then
12 |   |   connect_slave_to_master_node(node);
13 |   |   exit for;
14 |   end
15 end
16 if cluster nodes is empty then
17 |   register_new_cluster();
18 end
19 start_node();
```

The code for automatic clustering was used by every VM in the cluster during startup. Lines #1-5 initialized the docker container running the RabbitMQ server by first retrieving the name of the host machine. The name was typically the external ip address of the machine. The name was passed to the Docker container and set as the container hostname. This enabled RabbitMQ nodes to find each other during clustering over network. Line #7 sets up the docker and RabbitMQ environments to be accessible from the internet. In this setup, the external IP was added to the /etc/hosts file and the RabbitMQ configuration file was configured for external requests.

```
$ 127.0.0.1 EXTERNAL_IP >> /etc/hosts
$ [{rabbit, [{loopback_users, []}]}]. >> /etc/rabbitmq/rabbitmq.conf
$ rm -rf /var/tmp/aws-mon
$ rabbitmqctl add_vhost /admin
$ rabbitmqctl add_user admin admin
$ rabbitmqctl set_user_tags admin administrator
$ rabbitmqctl set_permissions -p /admin admin ".*", ".*", ".*"])
$ rabbitmqctl set_policy -p /admin server-qa-cluster ".*?" .
    {"ha-mode":"2","ha-sync-mode":"automatic"}
```

Line #8 sets up authorized users, user permissions and a policy to mirror each queue to 2 other nodes.
The remaining code lines controlled the actual clustering. If a cluster already existed or not, a newly started VM either joined an existing cluster or created an entry point itself. This was made possible through the load balancer's public static DNS address where the script made a request to the available underlying cluster. Depending on

the cluster existence, the script either got the IP of the cluster or received an error back. In the latter case, the script exited and the connecting client registered itself under the load balancer for the next VM to connect to.

3.4 Amazon Web Services

AWS provides the Amazon Elastic Compute Cloud (EC2) which is where the manual deployment was hosted. The manual deployment was constructed using multiple EC2-instances, an AMI and Auto Scaling Groups in association with a load balancer.

The service Amazon CloudFormation was used to create the template based system model. A JSON-file was used to describe the system model, dependencies and required packages.

Manual Deployment

Security Group

Any instance type created on AWS has all inbound network traffic blocked by default. This is a standard security precaution to prevent the vulnerabilities of allowing all network traffic. To enable networking to your instance a Security Group is needed.

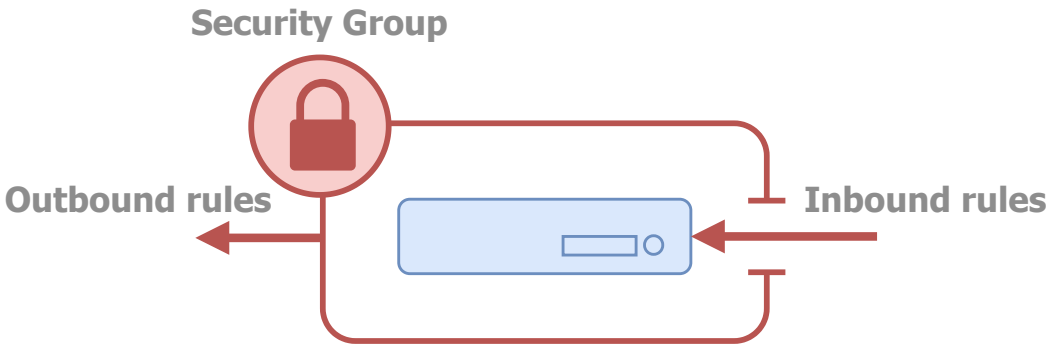


Figure 3.2: Illustrates the use of Security Groups

Figure 3.2 illustrates the usage of Security Groups. The first step of creating a virtual instance for a RabbitMQ server on AWS was to define the inbound and outbound port rules.

Protocol	Port Range	Description
TCP	4369	Used for peer discovery of other RabbitMQ nodes
TCP	5671-5672	Used by AMQP to transfer data
TCP	25672	Used for internal communication between nodes
HTTP	15672	Used to get access to the management UI

Table 3.1: Port access needed for a RabbitMQ server

As seen in table 3.1 there are four port ranges of interest. These were used to create a Security Group resource on AWS that could be assigned to all resources in the cluster.

Launching EC2 Instance

AWS provides a large variety of AMIs to be used on an EC2 instance. These consist of pre-installed packages and an operating system. The "Amazon Linux AMI 2018.03.0 (HVM)" AMI was chosen because it had Docker included by default.

After choosing an AMI the next step was to pick an instance type. AWS provides another large variety of options. Table 3.2 shows the different instance families that AWS offers. Depending on the tests performed different instance types were chosen.

Family	Description
General Purpose	Uses a balance of computation, memory and networking and is often used for more general solutions and workloads [15].
Compute Optimized	Used for higher computational performance with a higher performant host CPU. This family also offer the cheapest options with respect to computational performance [7].
Memory Optimized	Used for higher memory performance and is the cheapest option for memory intensive applications [22].

Table 3.2: Description of AWS instance family types

Custom AMI

AMIs are saved states of a VM that contain all the installed packages and files from the time it was created. A snapshot was taken of a VM and from that a custom AMI, based on the current state of the machine, was created. This enabled each newly created VM to be started up with the same state and libraries installed.

Docker container

In order for multiple RabbitMQ instances to connect to each other they must know where to connect. They do this by having a resolvable IP address as their name, which by default is the name of the host running the process. However, when running RabbitMQ inside a container, the name of the container is a randomly generated string and not a resolvable IP address. This was solved by providing an IP address as

the Docker hostname -h parameter when starting up the container. In this case, the IP address was the public IP of the host machine. This can be seen in figure 3.3. In addition to hostname the -p flag was used to forward all ports required by RabbitMQ. To enable clustering the same cookie was passed to all nodes. The following docker command line was used:

```
docker run -d -h ip-xx-xxx-xxx-xxx
-p 5672:5672 -p 15672:15672 -p 4369:4369 -p 25672:25672
--name rabbitmq-server -e RABBITMQ_ERLANG_COOKIE='cookie'
rabbitmq:3-management
```

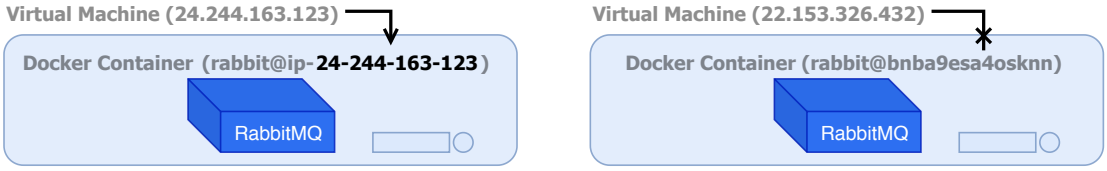


Figure 3.3: RabbitMQ container copying external ip

Launch Configuration

A Launch Configuration was used to create a template of how individual instances are defined. This had to be created since it is used together with an Auto Scaling Group resource to make a cluster. In addition to creating an instance by hand the Launch Configuration also takes input in a field called user data. User data is a custom script to be executed during the start up of new VMs. The script that was entered in user data is shown in section 3.3

Auto Scaling Group

In order to achieve a fully dynamic cluster an Auto Scaling Group was created. The minimum and maximum instance capacity was set to 1 and 3 respectively. In order to scale up or down the Auto Scaling Group is integrated with the CloudWatch API to get utilization statistics from the instances in the group. Together with this, two alarms were created to determine when to scale. The first alarm was set to add an instance if the average CPU utilization of a RabbitMQ node was more than 60% for 30 seconds. The other alarm was set to scale down the cluster if the average CPU utilization was less than 30% for 30 seconds. This solution also allowed nodes to be automatically connected to a load balancer. This way an auto scaling cluster was created that could scale depending on workload. Since high availability is important, setting the min and max capacity to 3 nodes would allow the solution to always maintain 3 nodes regardless of node failures. This was done during the testing sessions to prevent scaling from affecting test results.

Load Balancer

The classic Load Balancer was selected for use by the Auto Scaling Group. During initial testing this did however resulted in a drastic throughput decrease compared with throughput achieved without a Load Balancer. It also gave varying and unpredictable results between runs. Because of this, the Network Load Balancer was selected instead as it proved to be more reliable and more performant.

Table 3.3 shows the needed ports that were forwarded by the network load balancer. The only required port to send/receive AMQP data to/from a RabbitMQ server is 5672 and in order to reach the management UI from a browser, port 15672 was forwarded as well.

Load Balancer Protocol	Load Balancer Port	Instance Protocol	Instance Port
TCP	5672	TCP	5672
HTTP	15672	HTTP	15672

Table 3.3: Load Balancer port forwarding

CloudFormation

The CloudFormation service was used to create a reusable template of the manual deployment architecture. By creating a "Stack", the AWS CloudFormation service creates and connects all the required resources defined in a template. Since multiple clusters had to be created for each new test environment, the template enabled for easy reconfiguration of the cluster. This saved a lot of time and ensured the cluster was always deployed in the same way. CloudFormation is also used in enterprise solutions to simplify deployment. The CloudFormation was used to create an exact copy of the manual deployment.

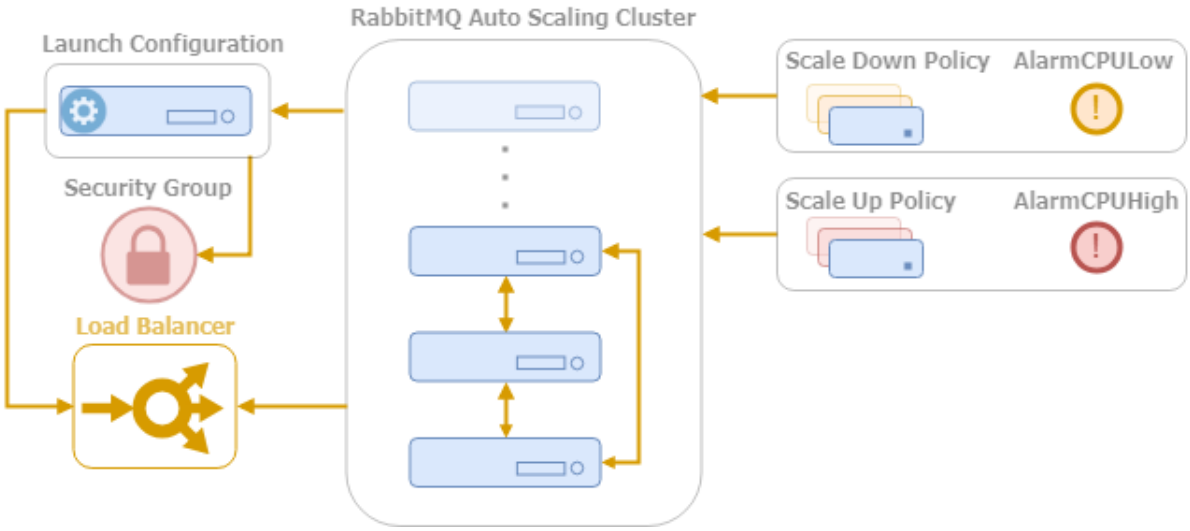


Figure 3.4: The architecture of an auto scaling broker solution

In order to get a working solution with CloudFormation the required resources were defined. Since the solution was already implemented manually the same infrastructure was assumed. Figure 3.4 shows what the manual approach architecture looked like and therefore acted as a guide when working with the CloudFormation template.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "",
    "Resources": {
        "AutoscalingCluster": { . . . },
        "ScaleUpPolicy":{ . . . },
        "AlarmCPUHigh": { . . . },
        "ScaleDownPolicy": { . . . },
        "AlarmCPULow": { . . . },
        "ElasticLoadBalancer": { . . . },
        "LaunchConfiguration": { . . . },
        "SecurityGroup": { . . .}
    }
}
```

Listing 3.1: CloudFormation template example

Listing 3.1 shows what a simple CloudFormation template looks like without the specific attributes defined for each resource. Given this initial file, together with the CloudFormation documentation provided by AWS, an Infrastructure as Code template could be created. Since AWS provides a large amount of attributes for each resource type, the manual approach was used as a guide to replicate the required components. Once the template was complete we could use that together with AWS to create a stack that binds all resources automatically for a working broker solution described in Figure 3.4.

3.5 Microsoft Azure

The Microsoft Azure platform provides the Azure Virtual Machine (AVM) service which is similar to the AWS EC2 service. The AVM service was used to host and construct the RabbitMQ cluster. A scalable pool of VMs along with a load balancer was created using the built in feature Scale Set.

Manual Deployment

Custom Azure Image

A custom image based on the CentOS 7.5 operating system was created to simplify the process of RabbitMQ clustering. Unlike the "User Data"-section available on the AWS platform, described in 3.4, the Azure platform did not implement this feature. Therefore an alternative solution involving a script scheduler called cronjob, was implemented for this platform. A file containing the script 3.3 was created and a cronjob was setup to run this script on startup of the machine. To avoid multiple executions of this script during a restart of the system, a check was placed in the beginning of the script to ensure a one time occurring setup process on each machine. An additional change to the script had to be made. For a RabbitMQ cluster to be formed, the nodes had to have a resolvable IP as their hostname. AWS solved this by automatically setting the external IP as the hostname on all of their VMs, whereas Azure gave each node an unresolvable randomized ID. This was solved programmatically by using an external API service in order to get the public IP of the VM. The response from the API was used to set the resolvable IP-address as the hostnames on the VMs. Before creating an image of the current state of the machine, a deprovisioning process had to be done. This was a necessary step in order to prepare the system for usage on multiple machines.

Scale Set

A Scale Set is a feature in Azure and was used when creating the cluster. The Scale Set handles the logic of deploying instances and scaling resources in the cluster based on policy rules. Using this feature a high and a low tier cluster was created. The instances in the cluster were configured to use the custom image defined in the previous section, along with the hardware specifications defined in table 4.2 for the respective performance version. The two clusters were deployed in the same availability zone to eliminate performance differences during testing, such as the network distance factor. Azure provides an option of deploying a VM in Low-Priority mode. This means that the VM is not guaranteed the specified performance at all times. The option of deploying instances in Low-Priority mode was thereby disabled. The Scale Set was configured with a min/max capacity and scaling policy rules based on CPU usage. Along with the creation of the Scale Set, a load balancer was created in the process. With a load balancer distributing the traffic to the cluster, high availability was achieved by ensuring that only healthy nodes receive data. Azure has two load balancing options to choose from, one optimal for web-based traffic and the other for stream-based traffic. The stream-based load balancer was chosen since it best fits the traffic type of the AMQP protocol. The basic SKU alternative was selected since the

benefits of upgrading to standard SKU would not affect the results, as described in section 2.6.
For the load balancer to distribute traffic to the cluster, necessary forwarding rules, Table 3.1, had to be created along with health probes. A health probe was defined in each of the forwarding rules to determine whether a node was healthy or not and thereby decide to forward traffic to it or not. The health probe was configured to ping RabbitMQ management port 15672 for a health diagnosis on all of the nodes.

The load balancer was, after the creation of forwarding rules and the health probes, able to forward traffic to a node as soon as the node had executed the script defined in section 3.3 and had started up the RabbitMQ management server.

Using a Scale Set all of the necessary network and interface components were created automatically, involving network interfaces, network security groups and virtual networks. To allow and receive traffic from the load balancer and from external sources additional configurations had to be made to the automatically created network security group. Rules for the ports defined in Table 3.1 were manually added along with SSH access on port 22 to allow for inbound traffic.

4 Comparison between AWS and Azure

4.1 Setup

PerfTest was chosen as the main testing tool for the broker solutions and is developed by the RabbitMQ team. PerfTest requires a single docker command to run and supports a variety of arguments to performance test broker solutions. A set of arguments were defined based on the types of tests to perform. Package sizes were narrowed down based on the requirements of Cybercom's most common IoT package sizes which is roughly 150 bytes for the average sensor package and 2 MB for an image. The test duration was set to 60 seconds for all tests.

Test ID	Producers	Consumers	Queues	Duration (s)	Message Size (bytes)
1	1	1	1	60	150
2	1	1	1	60	2000000

Table 4.1: Tests performed using PerfTest

Table 4.1 shows the tests performed. All tests were performed five times each with one minute pause between tests. All tests were performed on two different architectural solutions. The general architecture consisted of having a load balancer and three broker nodes as shown in Figure 3.1. The other architecture was to disregard the load balancer and connect to one of the nodes in the cluster. This was done on AWS and Azure.

The RabbitMQ management page displays metrics of the current allocated memory of the RabbitMQ server. This memory metric was constantly observed through the RabbitMQ management page during each test session. Signs of memory exceeding abnormal values were looked for during the tests.

Machine specifications

Table 4.2 and 4.3 show the VM specifications for the high and low-tier performance clusters on AWS and Azure. The clusters on both platforms were picked from the general purpose families and selected to match each other in terms of vCPUs and memory.

Cluster	Low-Tier	High-Tier
Azure Instance	B1s	A4m_v2
Family	General Purpose	General Purpose
vCPUs	1	4
Memory (RAM)	1 GB	32 GB
Image	CentOS-based 7.5	CentOS-based 7.5
Region	North Europe	North Europe
Availability Zone	Zones 1	Zones 1

Table 4.2: Azure specifications for the high and low tier clusters

Cluster	Low-Tier	High-Tier
AWS Instance	t2.micro	r5.xlarge
Family	General Purpose	General Purpose
vCPUs	1	4
Memory (RAM)	1 GB	32 GB
Image	Amazon Linux AMI (HVM)	Amazon Linux AMI (HVM)
Region	Ireland	Ireland
Availability Zone	eu-west-1	eu-west-1

Table 4.3: AWS specifications for the high and low tier clusters

Amazon EC2 Instance	r5.xlarge
vCPUs	4
Memory (RAM)	32 GB
Storage	EBS
Bandwidth	Up to 10 GB/s
AMI	Amazon Linux AMI 2018.03.0 (HVM)
Amazon Region	Ireland
Availability Zone	eu-west-1

Table 4.4: AWS specifications for the virtual machine used to perform tests

Azure Instance	A4m_v2
vCPUs	4
Memory (RAM)	32 GB
Bandwidth	High
Image	CentOS-based 7.5 2018.05.10 (RHEL)
Region	North Europe
Availability Zone	Zones 1

Table 4.5: Azure specifications for the virtual machine used to perform tests

To avoid false results based on geographical location or temporary variations in available bandwidth, an instance machine on each of the cloud platforms tested was running the tests, shown in Tables 4.4 and 4.5. Because of this the path between the source and destination became shorter and results would depend on the actual differences in infrastructure of the platforms. Results independent of geographical location also enabled performance comparisons between cloud platforms and their respective deployment methods. The two test machines performed the tests specified in table 4.1

4.2 Result

This section will present the results from the throughput tests made on AWS and Azure with the respective RabbitMQ cluster solution. The results will show how the two cloud platforms perform with and without a load balancer in tests that exercise varied package sizes based on typical Cybercom IoT traffic.

During the test observations, the memory allocation increased from idle state of 70MB allocation to around 80MB memory allocation. The critical threshold depended on the available memory of the host machine where the lowest identified threshold was around 140 MB memory. None of the tests performed did reach the critical threshold of allocated memory of the RabbitMQ server.

Figures 4.1 and 4.2 show performance differences of connecting through an AWS/Azure load balancer versus connecting directly to the high-tier clusters. On AWS there is a performance decrease of approximately 36% when sending traffic through the load balancer. On Azure the performance decrease is approximately 72%. When comparing differences between load balancer performance on Azure and AWS, AWS has 150% better throughput than what was achieved on Azure. The difference was even higher when comparing throughput without a load balancer, where AWS had a throughput of approximately 470% higher than Azure. This was a consequent result achieved when running the testing during several testing sessions.

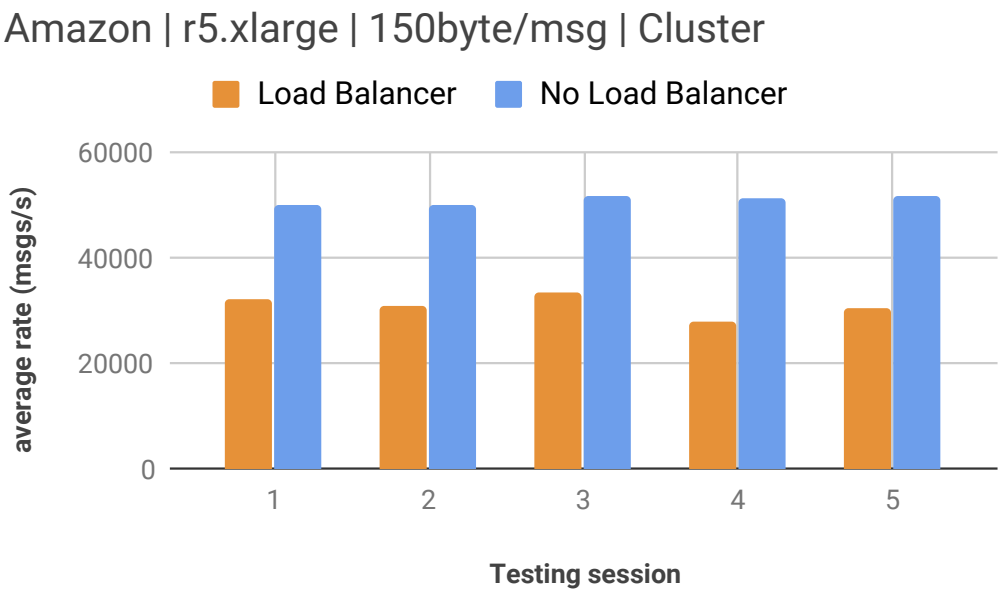


Figure 4.1: AWS high-tier cluster performance with 150byte package size

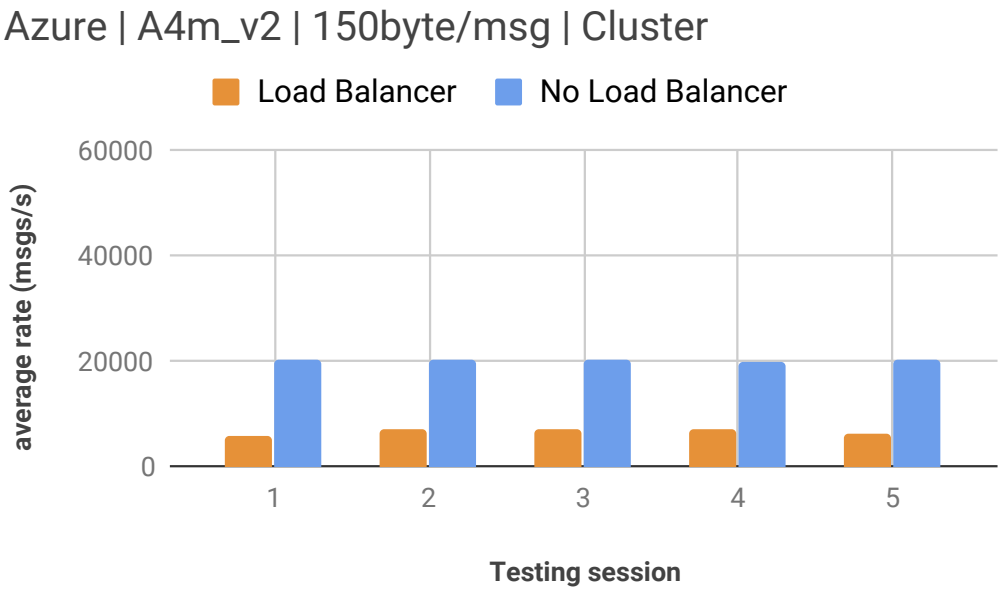


Figure 4.2: Azure high-tier cluster with 150byte package size

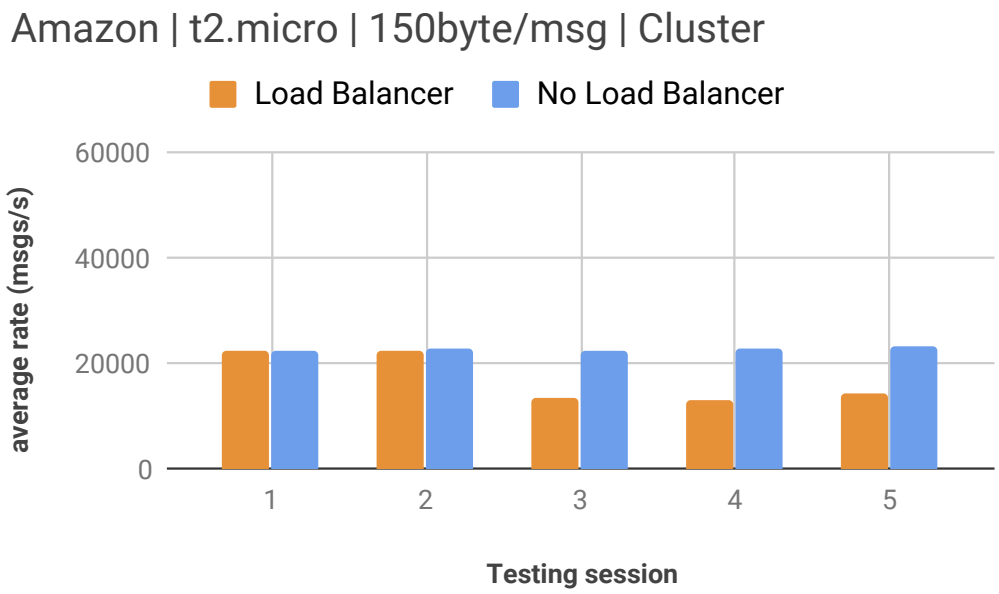


Figure 4.3: AWS low-tier performance with 150byte package size

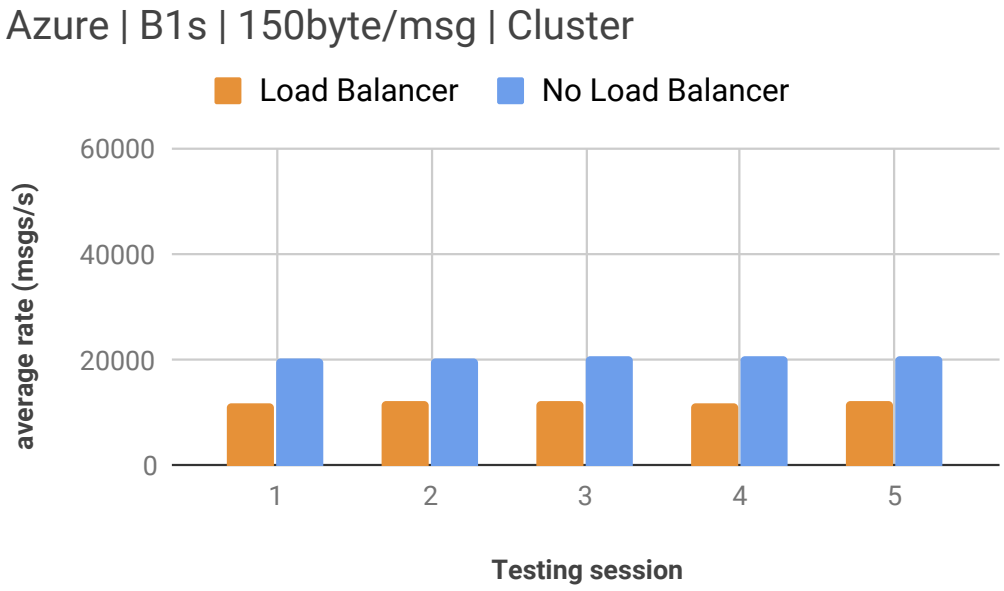


Figure 4.4: Azure low-tier performance with 150byte package size

In graph 4.3 the throughput was approximately the same during the initial testing sessions. It did however decrease for the load balanced approach during the testing sessions 3-5 as seen in the Figure.
In graph 4.4 the throughput was consistent during the test sessions with a consecutive performance decrease of around 43 % when sending traffic through the load balancer. No considerable performance difference can be seen between AWS and Azure without a load balancer. They both had an average throughput of around 21K msgs/s.

When comparing the Azure low-tier cluster with the high-tier cluster, Figures 4.2 and 4.4, the throughput was the same without a load balancer. The low-tier cluster performed better than the high-tier cluster with a load balancer.

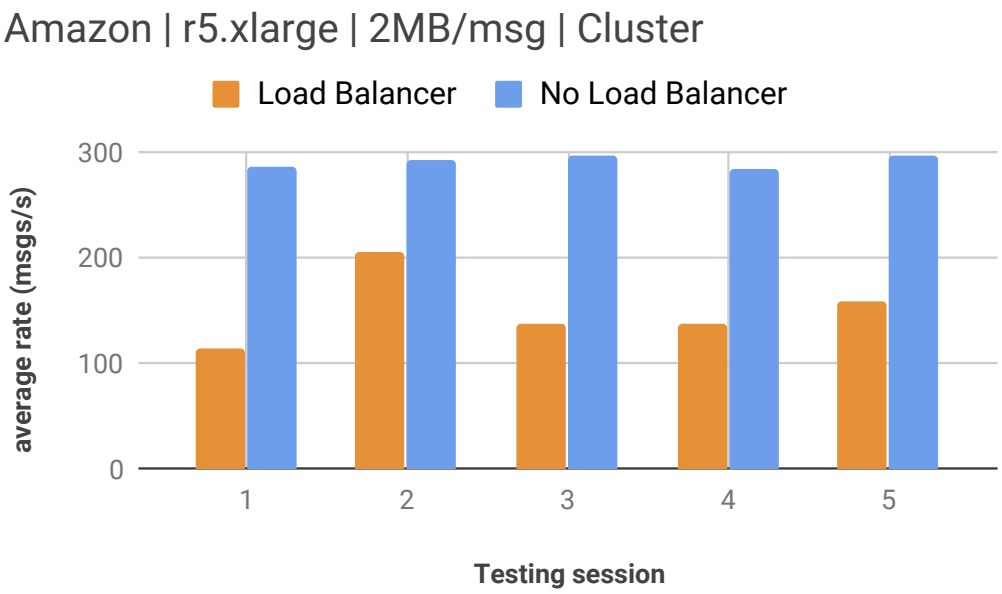


Figure 4.5: AWS high-tier performance with 2 MB package size

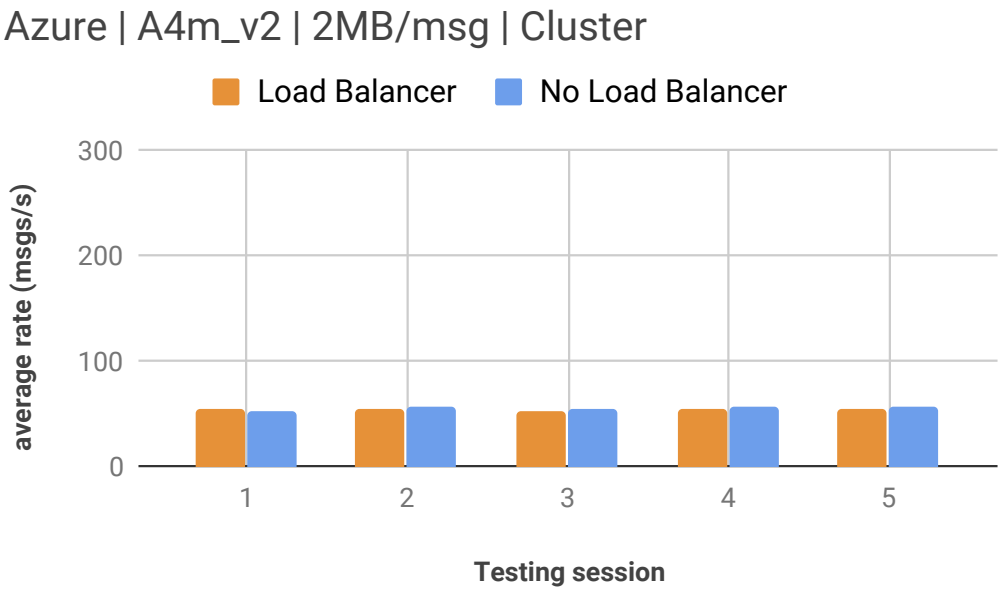


Figure 4.6: Azure high-tier performance with 2 MB package size

Figures 4.5 and 4.6 shows the high-tier results for AWS and Azure for the 2MB/message tests. On AWS there was a performance decrease of 62% when connecting

through a load balancer versus connecting directly to the cluster. On Azure no noticeable difference was seen between using a load balancer versus not using one.

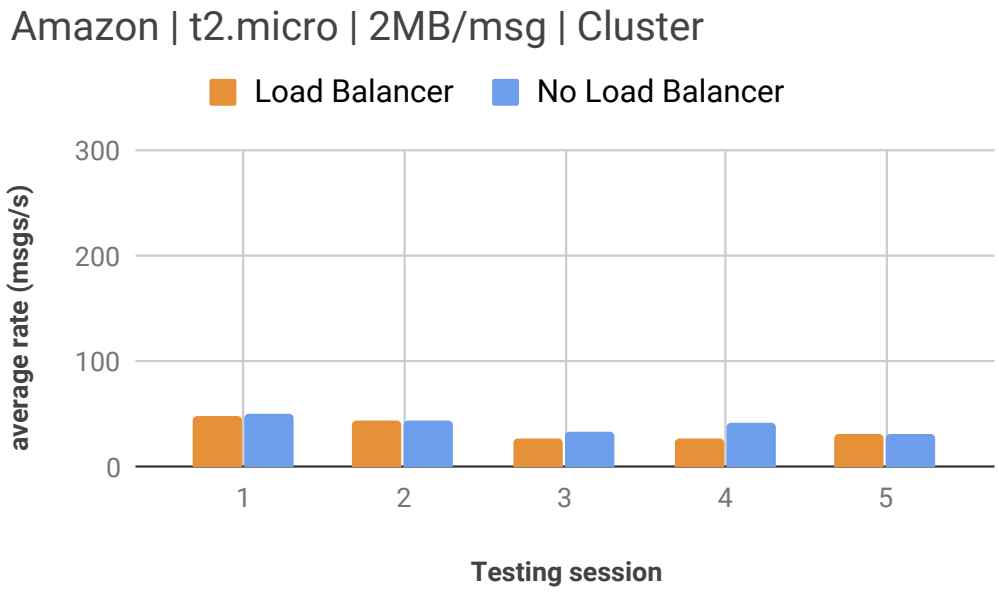


Figure 4.7: AWS low-tier performance with 2 MB package size

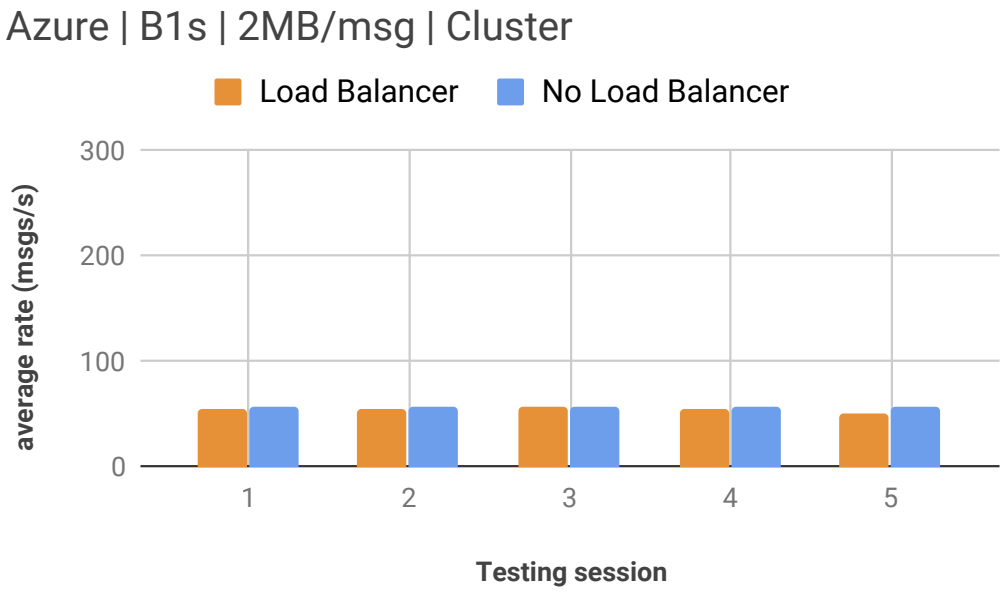


Figure 4.8: Azure low-tier with 2 MB package size

Figures 4.7 and 4.8 show the low-tier cluster results for AWS and Azure for the 2MB/message tests.

4.3 Discussion

In this section the throughput tests on AWS and Azure will be discussed.

AWS

On all throughput tests there is a consistent performance decrease when using a load balancer versus not using one on AWS. Since the load balancer should be able to handle high traffic throughput, the results are surprising. They show that choosing to include a load balancer in a system solution can in the worst case reduce the throughput by a factor of 2.52 as seen in the first testing session of Figure 4.5. The Figure also shows that the load balancer has the capability to throughput around 200 msgs/s, which equals to around 4 Gbit/s in one of the sessions. This performance decrease during the other testing sessions can be caused by several factors, e.g. current traffic load conditions or bandwidth throttling by AWS infrastructure. Since AWS specifies for the r5.xlarge machines to deliver network bandwidth of up to 10 Gbit/s, the latter factor is probably not the cause. It could relate to what Jackson et al. [18] found in their study. They concluded that performance variations could be found when running tests on an EC2 instance. The conclusions were that this was probably caused by other processes running on the same host machine and requiring computational resources.

Comparing the two 150 byte/msg tests, seen in figures 4.1 and 4.3, the results are clearly dependent on the hardware of the respective machines. The bandwidth should not be the bottleneck in the system since t2.micro and r5.xlarge have a measured base bandwidth of around 100 Mbit/s and 1.24 Gbit/s respectively. During the test sessions the amount of bits transferred is considerably lower than these bandwidth limits.

Azure

The pattern of having a decreased throughput was also observed on Azure when using a load balancer during the 150byte/msg tests. Tests show that choosing to include a load balancer on Azure can in the worst case reduce the throughput by a factor of around 4 as seen in Figure 4.2. Worth noticing is that the low-tier cluster performs just as good as the high-tier cluster in all of the non load balanced tests i.e. Figure 4.2 equals the performance of Figure 4.4 and Figure 4.6 equals performance of Figure 4.8. Since A4m_v2 has hardware and a pricing significantly better/higher than the B1s machine, the results practically mean that you might as well pay for a cheap low-tier machine since you will still get the same performance as with the high-tier machine.

The results show that the throughput reached in the 150byte/msg test is constant at around 20K msgs/s for the high and low-tier clusters. This result is about 30 Mbit/s of data flowing through the network, which should not be limited by available bandwidth of the two machines. Since the tests performed consists of sending data over a single queue, RabbitMQ handles this queue using a single CPU core. Therefore a mul-

ticore processor, such as in the A4m_v2, could theoretically handle more throughput in a multi-queue test by utilizing the multiple cores available.

Comparison between AWS and Azure

Comparing the results for the respective cloud platforms, the AWS platform outperforms Azure in all of the tests. The weaker low-tier machine is able to handle more traffic than the high-tier machine on Azure.

Based on the results observed through the RabbitMQ management page, the amount of memory on the host machine does not affect the performance of the RabbitMQ cluster. If memory would be an affecting factor for the performance, more memory would be allocated during the tests. It is therefore not worth paying for an instance with higher memory capacity if throughput is the prioritized metric. Having an instance with more memory does, however, serve another important quality, reliability. In the case of a sudden increase of messages a RabbitMQ server can only process a limited rate of messages per second and has to place unprocessed messages in memory. If the available memory is full, the RabbitMQ server will temporarily stop all inbound traffic. A limited memory on the host machine would contribute negatively to this effect and more messages would be dropped and never reach the intended destination.

5 Comparison between AWS and network accelerated Azure

This chapter will present a new setup strategy based on results and lessons learned from chapter 4

Accelerated network on Azure

Azure has an option during creation phase to activate accelerated networking. Accelerated networking bypasses virtual switches that the traffic normally has to go through. As a result, the throughput is greatly increased, latency is reduced and CPU utilization is decreased. The virtual switch, as seen in figure 5.1, handles policies and security rules for all arriving network traffic. When network acceleration is enabled, the virtual switch is bypassed and security rules are instead handled by the hardware of the receiving VM, as seen in figure 5.2. [10]

According to Azure, accelerated networking is supported on VMs with 2 or more vCPUs e.g. the DS_v2 or Fs instance families. The choice of VMs on Azure was therefore changed to instances with support for accelerated networking. The new instance specification for Azure is shown in table 5.1.

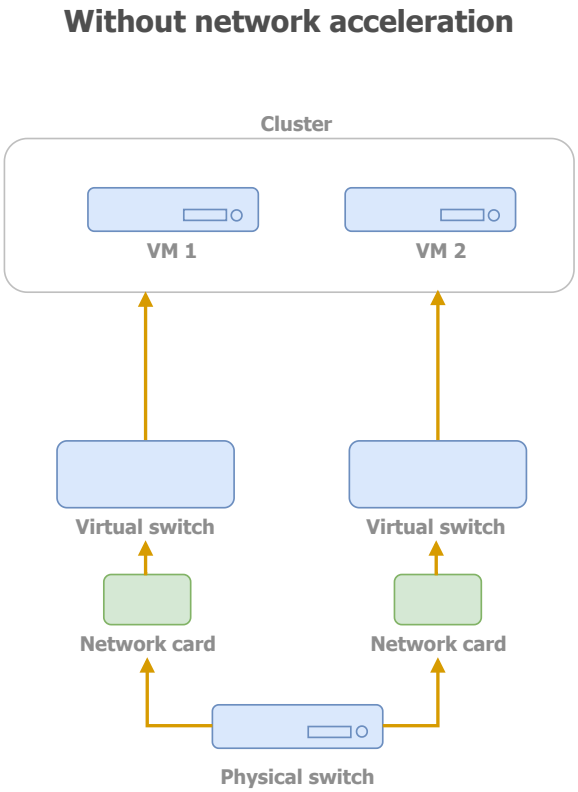


Figure 5.1: Azure server architecture without network acceleration

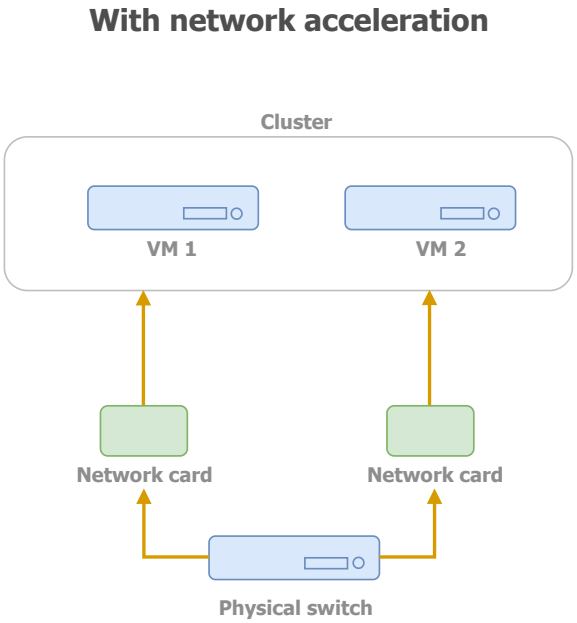


Figure 5.2: Azure server architecture with network acceleration

Instance specifications

Cluster	Low-Tier	High-Tier
Azure Instance	D2_v2	D3_v2
Family	General Purpose	General Purpose
vCPUs	2	4
Memory (RAM)	7 GB	14 GB
Image	CentOS-based 7.5	CentOS-based 7.5
Region	North Europe	North Europe
Availability Zone	Zones 1	Zones 1

Table 5.1: Specifications for the Azure high and low-tier cluster VMs

Table 5.1 shows the new specifications for the VMs on Azure. With these changes, the instances on AWS had to be changed accordingly to match the new instances as closely as possible. Table 5.2 shows the new setup used on AWS.

Cluster	Low-Tier	High-Tier
AWS Instance	m5a.large	r5.xlarge
Family	General Purpose	Memory Optimized
vCPUs	2	4
Memory (RAM)	8 GB	32 GB
Image	Amazon Linux AMI (HVM)	Amazon Linux AMI (HVM)
Region	Ireland	Ireland
Availability Zone	eu-west-1	eu-west-1

Table 5.2: Specifications for the AWS high and low-tier cluster VMs

The testing instances for this chapter were also updated to withstand the performance of the updated cluster, the specifications are shown in tables 5.3 and 5.4.

AWS Instance	c5n.4xlarge
vCPUs	16
Memory (RAM)	42 GB
Bandwidth	Up to 25 GB/s
AMI	Amazon Linux AMI (HVM)
Amazon Region	Ireland
Availability Zone	eu-west-1

Table 5.3: AWS specifications for the VM used to perform tests

Azure Instance	F16s
vCPUs	16
Memory (RAM)	32 GB
Bandwidth	Very High
Image	CentOS-based 7.5
Region	North Europe
Availability Zone	Zones 1

Table 5.4: Azure specifications for the VM used to perform tests

5.1 Result

This section will present the results from the throughput tests made on AWS and Azure with the respective RabbitMQ cluster solution. The results will show how the two cloud platforms perform with and without a load balancer in tests that exercise varied package sizes based on typical Cybercom IoT traffic. Comparisons of tests with accelerated network on and off are included to show the difference in performance it makes.

Comparing difference in accelerated network performance

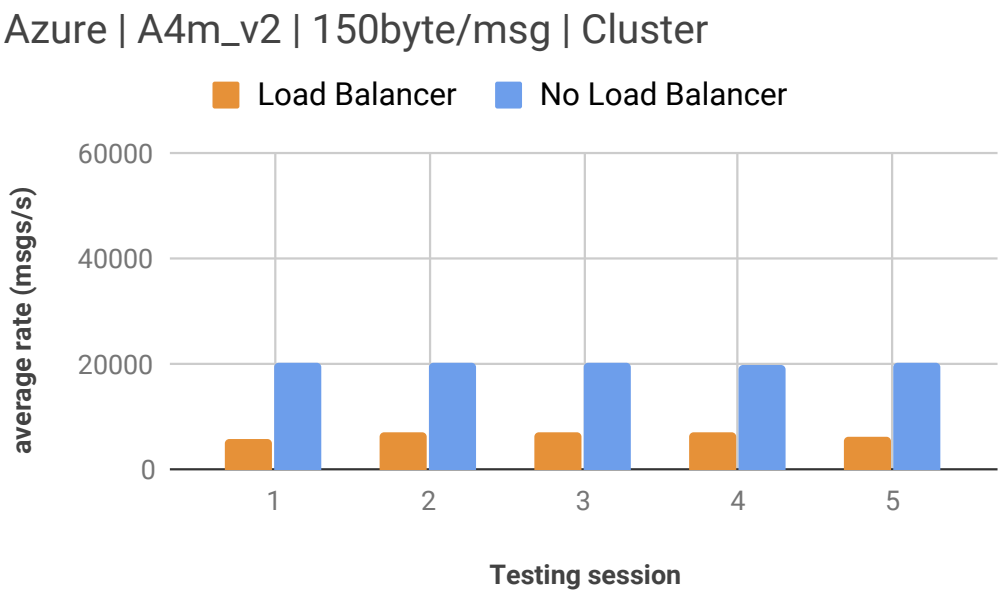


Figure 5.3: Azure A4m_v2 (high-tier) performance with network acceleration disabled.

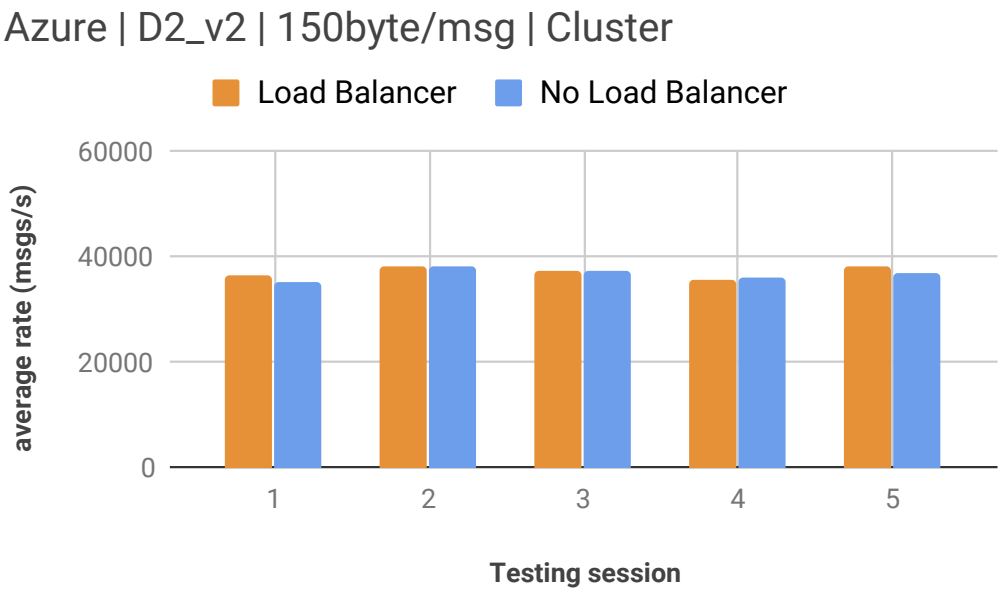


Figure 5.4: Azure D2_v2 (low-tier) performance with accelerated networking enabled

Figures 5.3 and 5.4 show the performance difference between using a VM with network acceleration disabled versus enabled on the Azure platform. Since A4m_V2 is a stronger instance than D2_v2 in terms of more vCPUs and memory, A4m_V2 should

in theory perform better and have a higher throughput. However, these results show that enabling network acceleration results in a throughput increase of 660% compared to a stronger instance with disabled network acceleration.

Comparison between platforms with network acceleration enabled

This section will present results where network acceleration is enabled on Azure. The results presented will be a fair comparison of the true throughput of the respective cloud platforms.

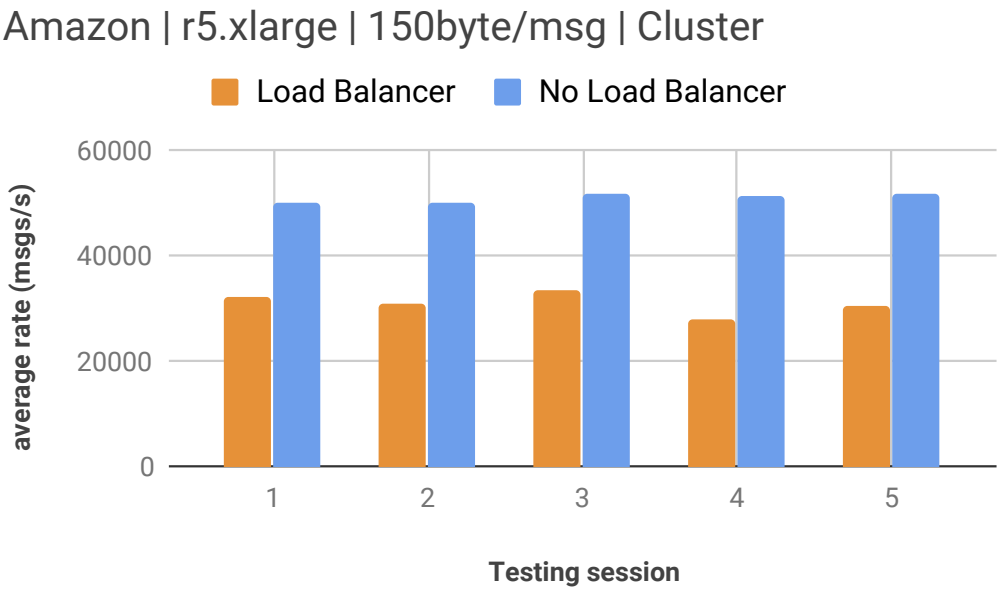


Figure 5.5: AWS high-tier cluster performance with 150 byte package size. This figure was also used in section 4

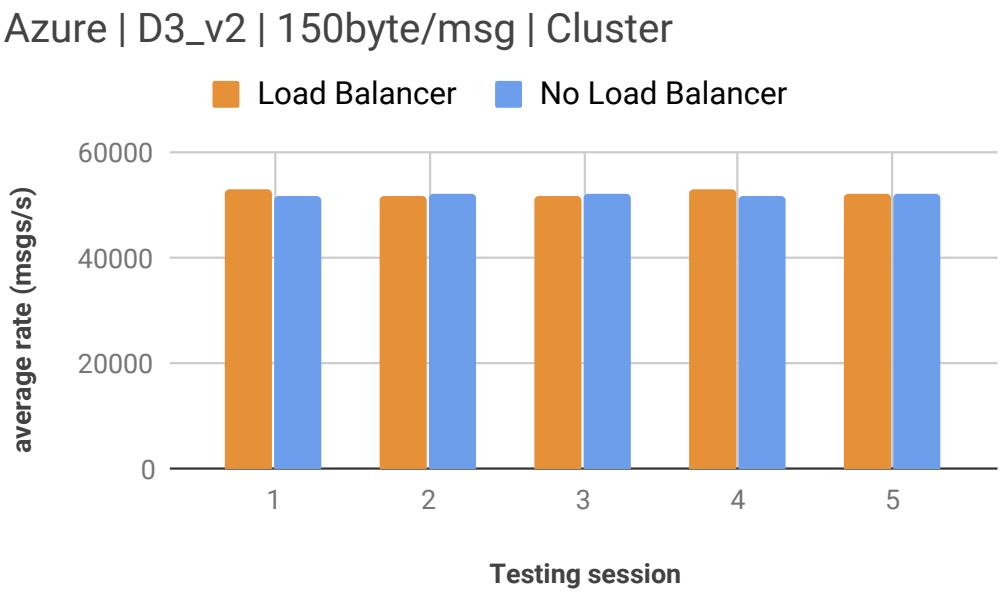


Figure 5.6: Azure high-tier cluster performance with 150 byte package size

The updated network acceleration settings for Azure resulted in an average throughput of 51878 msgs/s when bypassing the load balancer, as shown in figure 5.6. The same test performed on AWS resulted in an average throughput of 50936 msgs/sec, as shown in figure 5.5. This is an increase on Azure of 1.85% compared to the throughput achieved through AWS. When comparing the performance through the load balancer, Azure and AWS had an average throughput of 52127 msgs/s and 31891 msgs/s. Connecting through the load balancer resulted on Azure in a throughput increase of 0.48% whereas on AWS an average throughput decrease of 37% over the non load balanced tests.

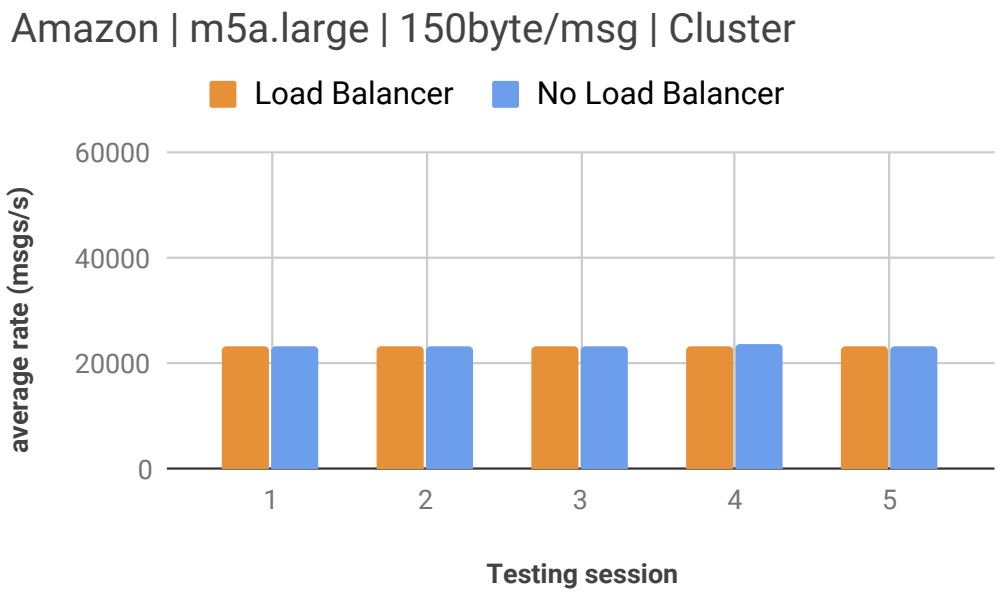


Figure 5.7: AWS low-tier cluster performance with 150byte package size. This is the same graph as in section 4 and figure 4.1

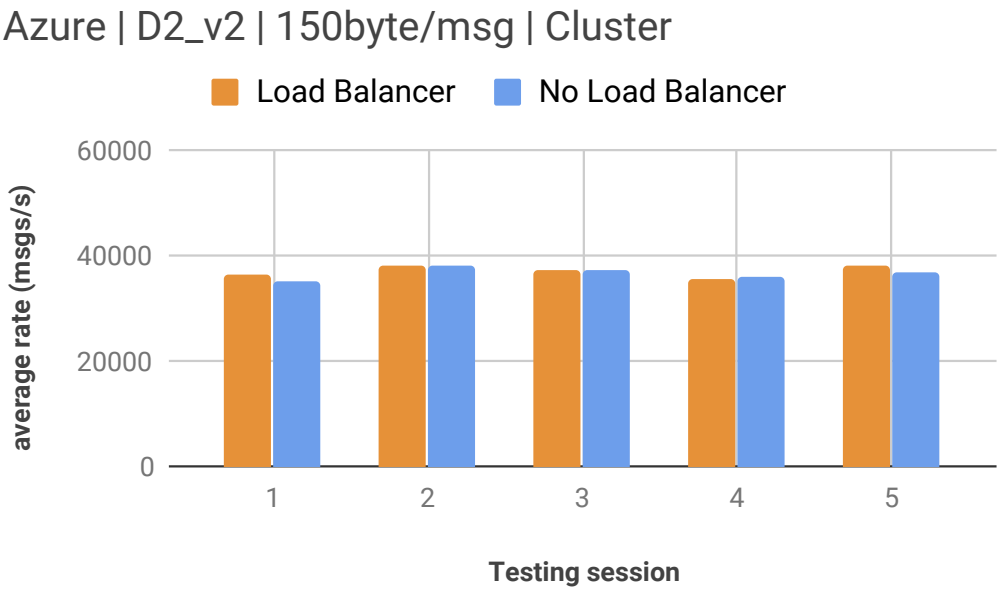


Figure 5.8: Azure low-tier cluster performance with 150byte package size

Figures 5.7 and 5.8 show the results achieved by AWS and Azure on the low-tier clusters. The throughput when bypassing the load balancer was 23235 msgs/s and 36946 msgs/s for AWS and Azure respectively. The throughput when connecting through the load balancer resulted in an average of 23213 msgs/s and 36607 msgs/s.

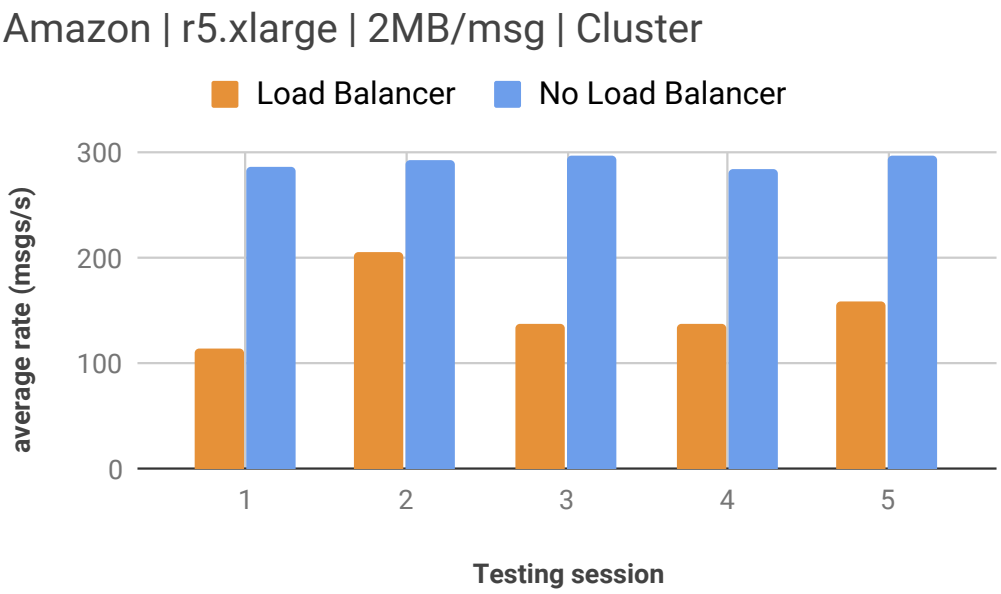


Figure 5.9: AWS high-tier cluster performance with 2MB byte package size. This figure was also used in section 4

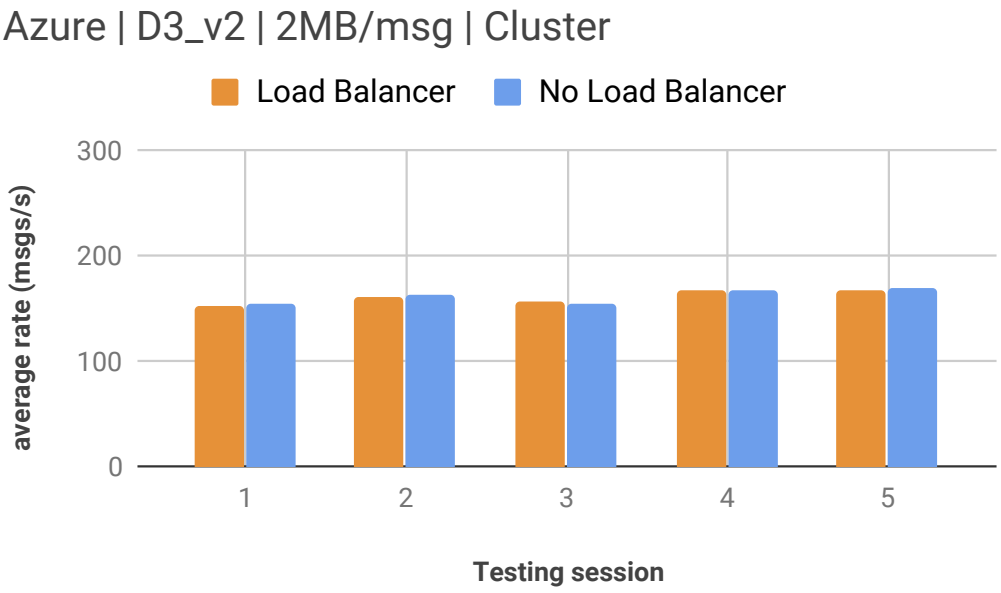


Figure 5.10: Azure high-tier cluster performance with 2MB package size

Figures 5.9 and 5.10 show the results for the high-tier 2MB/msg test. AWS had an average performance of 290 msgs/s when bypassing the load balancer. The Azure platform had on the same test a throughput of 150 msgs/s. When connecting to the load balancer, the throughput was 151 msgs/s and 150 msgs/s for AWS and Azure respectively.

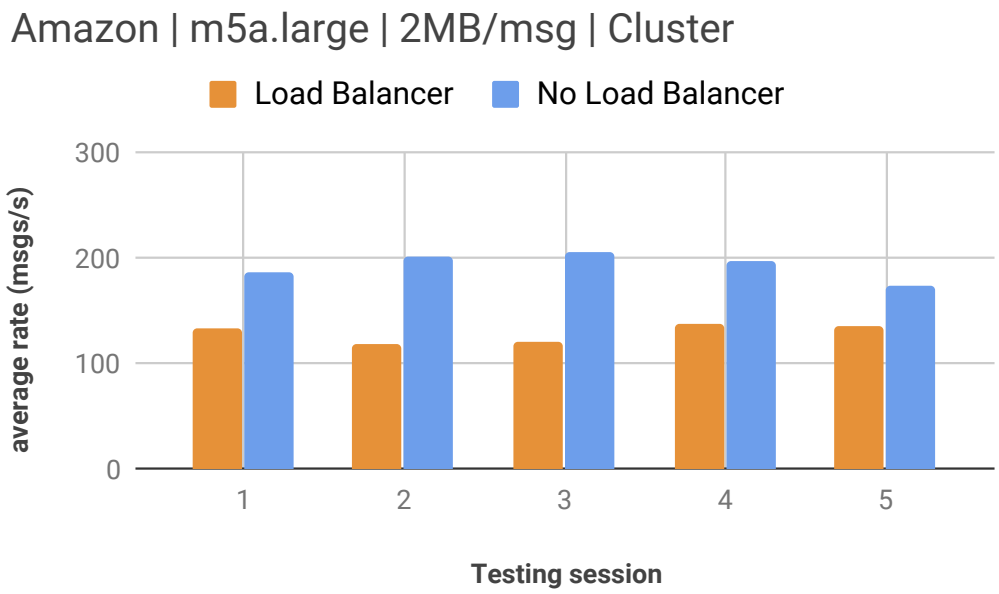


Figure 5.11: AWS low-tier cluster performance with 2MB package size.

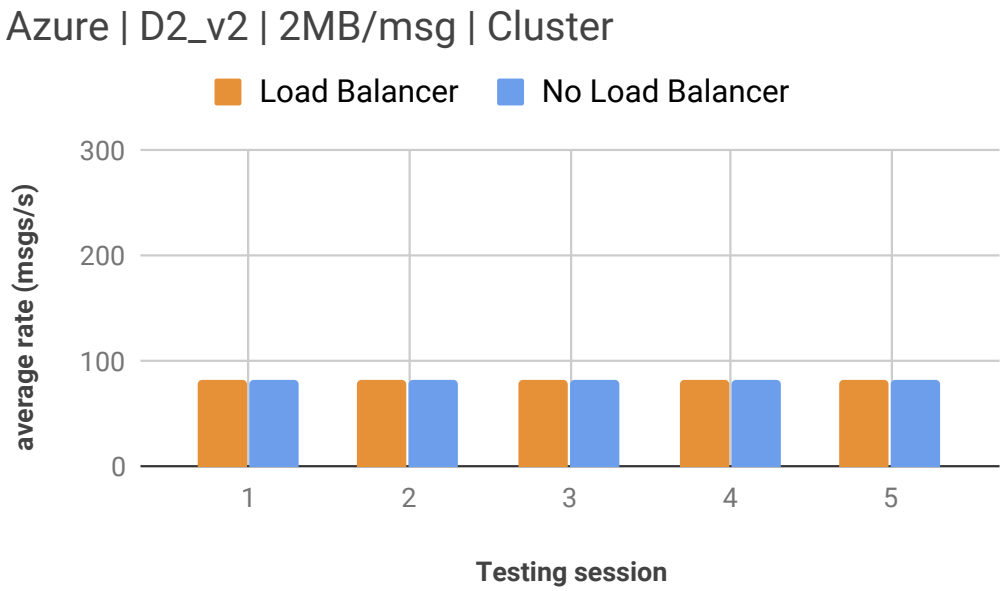


Figure 5.12: Azure low-tier cluster performance with 2MB package size

Figures 5.11 and 5.12 show the throughput results for the low-tier clusters. AWS had an average performance of 192 msgs/s when bypassing the load balancer. The Azure platform had on the same test a throughput of 81 msgs/s. When connecting to the load balancer the throughput was 134 msgs/s and 81 msgs/s for AWS and Azure respectively.

5.2 Discussion

This section will discuss the results of the fair throughput comparisons between AWS and Azure.

AWS

The load balancer of AWS had varying performance compared to the more stable non-load balanced throughput values. This was a consistent pattern throughout the tests. Reasons for this could be the additional latency related with DNS resolution or limited bandwidth from or to the load balancer. The 2MB/msg tests on the r5.xlarge instance, shown in figure 5.9, show that the bandwidth is at least 4 Gbit/s through the load balancer. Bandwidth throttling could be a potential reason for this behaviour. AWS might have a normal calculated bandwidth usage of the system. When much traffic is suddenly passed through the load balancer, the calculated bandwidth has little time to react and change the limits accordingly. In that case, a longer test of an hour would probably trigger this bandwidth limit change.

Azure

The accelerated network configuration for Azure resulted in a big throughput improvement. This enabled a more fair comparison between the two cloud platforms. Azure does not seem to suffer from the same performance degradation as AWS when connecting through the load balancer. The results between the load balanced and non load balanced solution have close to equal throughput results.

General discussion

When comparing the enabled with the disabled accelerated networking tests, the results show a clear throughput increase with accelerated networking enabled. According to the tests, having this feature enabled would for every Internet application communicating with or receiving information from other sources, be a way to speed up your internet traffic throughput up to 660%. The choice of enabling accelerated networking on all instances should therefore be obvious since the information on accelerated networking only states advantages and no disadvantages [10].

According to the 2MB/msg tests AWS performs better than Azure when handling big data packets . The low-tier cluster on AWS even performs better than the high-tier cluster on Azure for these tests. Azure does, however, perform better than AWS on the smaller packet tests. It is hard to speculate the reasons behind these results and why the two platforms differ from each other in each of the tests. What could be concluded is that AWS has a higher throughput when sending large messages, e.g. images. Azure has a higher throughput when sending small packets, e.g. sensor data.

The fact that the high-tier instance specification for AWS, as specified in table 5.2, has 19 GB of memory whilst the Azure high-tier instance only has 14 GB memory, should not affect the throughput. This was concluded in chapter 4.

6 Performance analysis of compute optimized instances

In this chapter, a performance analysis between compute optimized instances on AWS and Azure will be presented. Also an analysis of differences in performance between compute optimized and general purpose instances on AWS will be done.

6.1 Testing

Previous tests found differences in how a load balancer can affect performance for a single queue and how message size affects the performance. The next step was to look at how compute optimized instances perform for high loads over multiple queues. This was done to see how CPU affects performance and if there are any differences between cloud platform instances of the same instance family. Two additional setups were made on AWS to compare two instance types, t3 and m5, from the general type family with c5 from the compute optimized family. No load balancer was used together with the instances, to focus on pure instance performance. The same setup of testing instances was used as in chapter 5 and specified in Tables 5.3 and 5.4.

To simulate more realistic loads in a real production environment, PerfTest allows for high load tests over multiple queues.

Test ID	Producers	Consumers	Queues	Duration (s)	Message Size (bytes)
1	1	1	1	60	12
2	2	2	2	60	12
3	3	3	3	60	12
4	4	4	4	60	12
5	10	10	10	60	12
6	200	200	200	60	12

Table 6.1: Tests performed to simulate high loads using PerfTest

Table 6.1 shows the tests that were defined for simulating high loads over multiple queues. Since queues run on a single core, a stronger CPU should give higher performance.

Provider	Instance type	Family	vCPU	Memory (GB)	Bandwidth
Amazon	c5.xlarge	Compute Optimized	4	8	Up to 10 Gbit/s
Amazon	c5.2xlarge	Compute Optimized	8	8	Up to 10 Gbit/s
Amazon	c5.4xlarge	Compute Optimized	16	8	Up to 10 Gbit/s
Azure	F4s	Compute Optimized	4	8	High
Azure	F8s	Compute Optimized	8	16	High
Azure	F16s	Compute Optimized	16	32	Very high

Table 6.2: The specification of the virtual instances running RabbitMQ

Table 6.2 shows the instances used for the different broker clusters. A compute optimized instance type was chosen for both cloud platforms to get high CPU performance. Both AWS and Azure advertise these instance types to perform better with regards to CPU heavy calculations. An additional c5n.4xlarge and F16s instance was used as testing instances on the respective cloud platforms. Both testing instances were placed in the same region and availability zone as the broker cluster they tested.

As mentioned earlier, two additional instance types were chosen to test the performance differences between family types on AWS. The AWS platform was prioritized for these tests and equivalent tests on Azure were not conducted. In Table 6.3, the specifications for the instances are listed. The main difference between the m5 and t3 general purpose series is that the t3 series has a network burst technology. This means that the average bandwidth gained from a t3 instance is less than the specified bandwidth. This could mean that the bandwidth achieved from such an instance could be 2-3 Gbit/s if used for longer periods of time. The 16 vCPU t3 instance is not included since AWS does not have this option.

Provider	Instance type	Family	vCPU	Memory (GB)	Bandwidth
Amazon	m5.xlarge	General purpose	4	16	Up to 10 Gbit/s
Amazon	m5.2xlarge	General purpose	8	32	Up to 10 Gbit/s
Amazon	m5.4xlarge	General purpose	16	64	Up to 10 Gbit/s
Amazon	t3.xlarge	General purpose	4	16	Up to 5 Gbit/s
Amazon	t3.2xlarge	General purpose	8	32	Up to 5 Gbit/s

Table 6.3: The specification of the general type instances on AWS

Testing bandwidth on Azure VMs

As seen in Table 6.2, there are no actual bandwidth values for the Azure VMs. In order to make sure that the bandwidth was not a bottleneck in testing the system, bandwidth tests were performed on the instances.

The tool ntttcp, a network performance tool for Windows and Linux, was used in running a bandwidth test between the testing instance and the instances running the RabbitMQ server. The measured bandwidths for the respective Azure VM is shown in table 6.4.

Instance type	Bandwidth	Measured bandwidth
F4s	High	3 Gbit/s
F8s	High	6 Gbit/s
F16s	Very high	12 Gbit/s

Table 6.4: Measured bandwidth on Azure VMs

6.2 Result

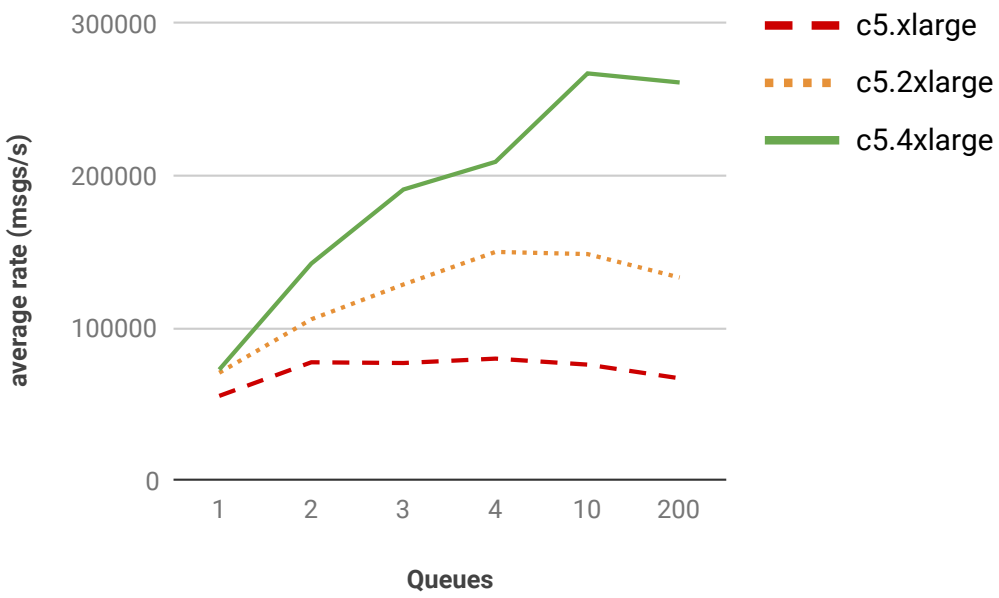


Figure 6.1: AWS compute optimized VM performance

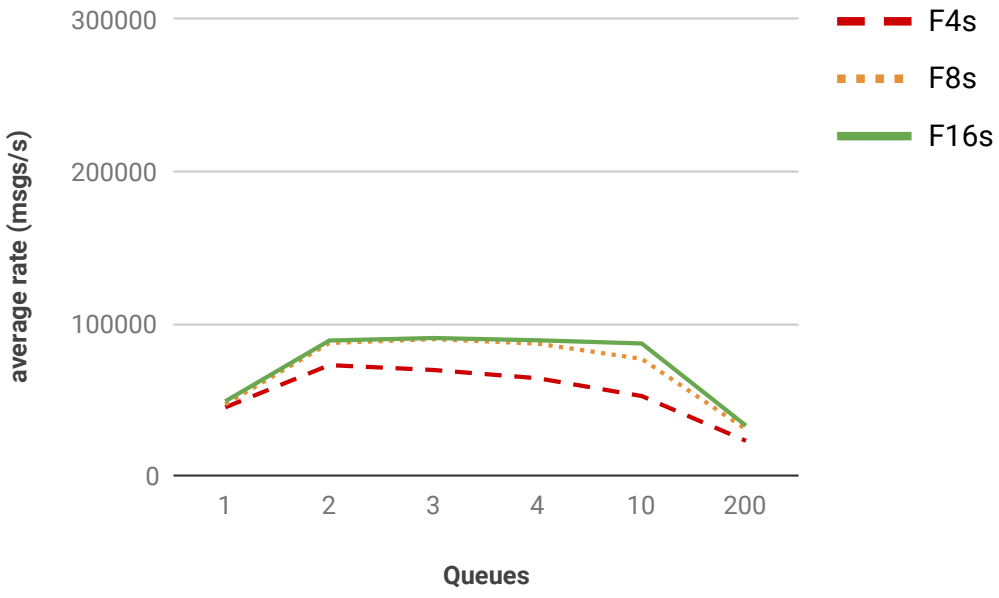


Figure 6.2: Azure compute optimized VM performance

Figures 6.1 and 6.2 show the performance of compute optimized VMs on AWS and Azure. The figures show the relation between performance and number of queues for different compute optimized instances.

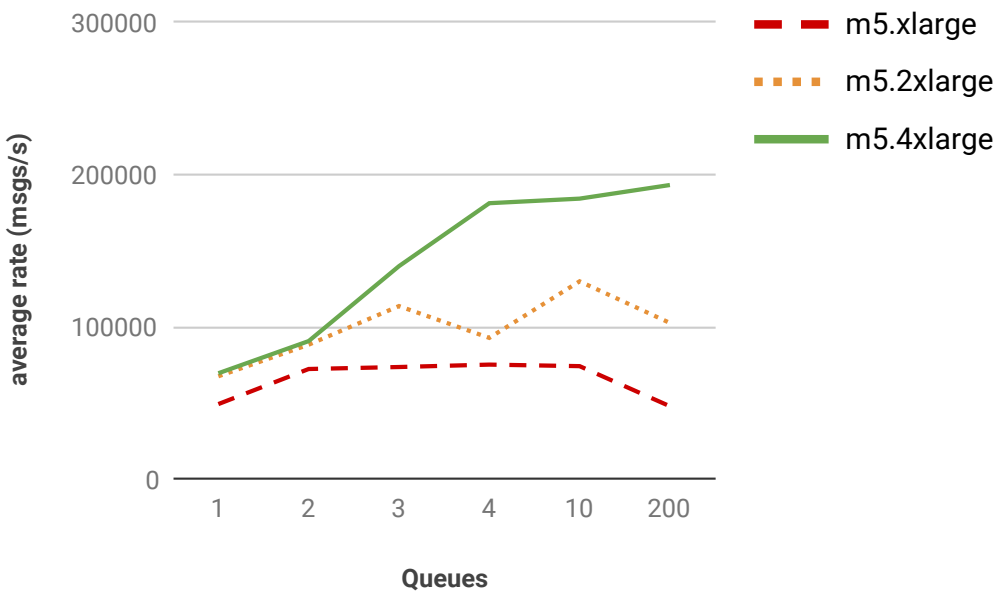


Figure 6.3: AWS general purpose m5 instance family performance

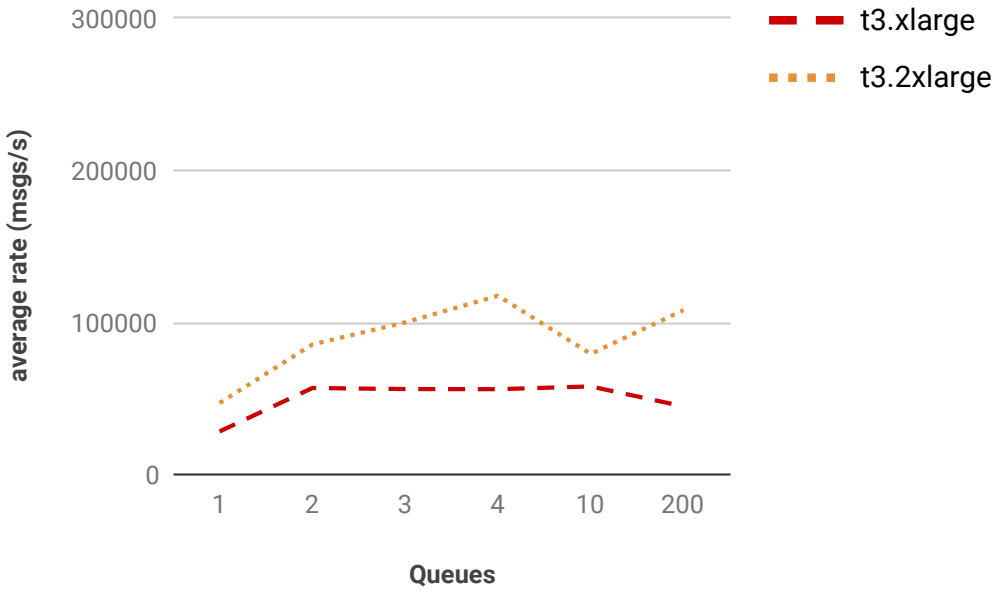


Figure 6.4: AWS general purpose t3 instance family performance

Figures 6.3 and 6.4 show the performance of general purpose VMs on the AWS cloud. The figures show the relation between performance and number of queues for different general purpose instances.

6.3 Discussion

In this section the results will be discussed and possible sources of error affecting the results.

AWS

The results shown in Figure6.1 support that having more vCPUs on AWS will give an increase of performance. With higher loads from 10-200 queues the difference is most significant. Since RabbitMQ is running one queue on a single core, this supports the hypothesis that more vCPUs should give higher performance. What can also be seen is that maximum throughput for certain hardware is reached roughly when the number of queues equals the number of vCPUs. The throughput is never increased when the number of queues exceeds the number of vCPUs. This is why the point where maximum throughput is achieved differs from instance to instance, as can be seen in Figure 6.1. The maximum throughput for the any instance type should in theory occur when the number of queues equals the number of vCPUs. For the c5.4xlarge instance the maximum value should be reached at 16 queues. As discussed and concluded in chapter 4, memory is not a key factor in performance for a RabbitMQ server. The m5.4xlarge instance has 64 GB of memory whilst the c5.4xlarge has 8 GB but still achieves a higher throughput, as seen in Figures 6.1 and 6.3. This also supports the conclusion regarding how memory impacts RabbitMQ performance.

What can be seen and analyzed from the tests done on the m5 and t3 series, Figures 6.3 and 6.4, is that the average throughput is approximately the same when comparing the t3.xlarge with m5.xlarge and t3.2xlarge with m5.2xlarge. By analyzing previous results and trends in this chapter, the performance drops seen for the m5.2xlarge and t3.2xlarge instances during 4 and 10 queue tests respectively are probably caused by temporary performance fluctuations. During the other tests, performance drops have not occurred before the number of queues has grown larger than the total number of vCPUs.

One of the bigger differences between the t3 and m5 series is that t3 has a burst feature built in which enables higher clock frequencies for short periods. By comparing the two current figures it is hard to draw conclusions whether the burst feature was active or not during the tests. This could probably be concluded if the tests were run for a longer time. The m5-series provides more reliable throughput results than the t3-series, since there is no external control of the t3-burst activation.

Azure

The result from the test done on Azure, shown in Figure 6.2, does not correlate with the AWS results presented in Figure 6.1. A maximum throughput can clearly be seen at around 90K msgs/s for the F16s and F8s instance. However, the tests for which results are shown in Figure 6.2 have network acceleration enabled and are setup in the exact same way as in Chapter 5, with the main difference that these results do not show differences in throughput. The tests has been rerun multiple times during

different times of the day to substantiate them. What differentiate this test from the ones done in chapter 5 is the utilization of multiple vCPUs. A reason why the cluster is not able to scale better past two queues could be a limitation by the Azure platform to prevent full utilization of the CPU. In Figure 6.3 the AWS instances m5.4xlarge and m5.2xlarge have almost identical throughput results for 1 and 2 queues, just like the Azure F8s and F16s instances in Figure 6.2. The difference could be that AWS is able to utilize more of the CPU power. However, there are no indications to support this except from the results, so it should be considered an unlikely reason. The F16s instance was used in these tests as the testing instance. It is the same instance as the strongest VM running the RabbitMQ server and could potentially be a cause of the low throughput results on Azure. The throughput of the F4s instance drops off earlier and should therefore not be affected by the throughput of the testing instance. The F8s and F16s instances are stronger instances and could be processing the packets faster than the testing instance is able to produce them, which could be why the two instances perform equally throughout the test. However, the AWS testing instance, which is also a 16 vCPU compute optimized instance, is in these tests able to produce up to 900% more packets than the Azure instance. The computational difference of the two testing instances should in theory not differ that much since they have equal specifications. The same testing instances were also used in chapter 5 and the Azure instance was then able to produce as much as, or even more than the AWS testing instance.

The bandwidth was manually measured and gave speeds up to 12Gbit/s, as shown in Table 6.4. All of the instances used in the test support higher throughput than what was sent. Therefore the bandwidth should not be a bottleneck in the system. Given these results a compute optimized c5n.4xlarge instance on AWS has a throughput advantage over the F16s on Azure of approximately 900% when sending traffic over 200 queues.

7 Cost model analysis of RabbitMQ instances

7.1 Cost modelling

The goal of finding the most cost efficient instances was to see if the computational family really is the cheapest and best option for RabbitMQ brokers. There is no telling how much performance all enterprise solutions need. Providing a general understanding of which instances are the most price worthy, hopefully brings something to brokering on the cloud. Both Amazon and Microsoft provide a tool that can be used to calculate cost. The Microsoft Azure Pricing Calculator and AWS Simple Monthly Calculator. By committing to instances for one or three years, you get a discount on the on-demand cost. Since the discount was the same on both platforms, the on-demand cost was picked for this analysis. In addition to the billing option, the desired usage was set to 24 hours/day. The calculated costs can be seen in Tables 7.1 - 7.3 below.

Platform	Instance type	Monthly Cost
AWS	c5.xlarge	$ 140.55
Azure	F4s	$ 164.98
AWS	m5.xlarge	$ 156.65
AWS	t3.xlarge	$ 133.52

Table 7.1: Low cost instances

Platform	Instance type	Monthly Cost
AWS	c5.2xlarge	$ 281.09
Azure	F8s	$ 329.96
AWS	m5.2xlarge	$ 313.30
AWS	t3.2xlarge	$ 267.04

Table 7.2: Medium cost instances

Platform	Instance type	Monthly Cost
AWS	c5.4xlarge	$ 562.18
Azure	F16s	$ 660.65
AWS	m5.4xlarge	$ 626.60

Table 7.3: High cost instances

Instead of only comparing costs between instance types using the cost tables, the score equation was developed. This was developed to help choose between instances based on arbitrary throughput requirements on the message broker. An instance was assumed to handle loads below the maximum capacity, derived in chapter 6. Since cost is related to the hardware specifications of an instance, the 4, 8 and 16 vCPU instances were categorized into three different price classes. This made it easier to see the differences in cost within each tier. Together with the derived costs from Tables 7.1 - 7.3 and the queue performance tests conducted in Chapter 6, the three most valuable instance options for different throughput requirements could be found. The following equation was used:

$$Score = Throughput / MonthlyCost. \tag{7.1}$$

The Throughput in equation 7.1 is an arbitrary throughput demand of a RabbitMQ cluster. The throughput was set to 10k and incremented by another 10k for each calculated score. With this, the instances with the highest scores for each required throughput were chosen. Since each instance performs differently based on the number of queues, the scores were calculated for 1, 2, 10 and 200 queues. A capacity equation was added in addition to the score, to see how much of the instance capacity that is currently being used for the chosen throughput demand.

$$Capacity = (Throughput / MaximumThroughput) * 100. \tag{7.2}$$

MaximumThroughput in equation 7.2 refers to the benchmarked highest throughput achieved by that instance. A new capacity was calculated for each of the four queue tests.

7.2 Result

The Tables 7.4 - 7.7 show which instances should be chosen for an enterprise solution based on the required throughput and number of queues. The first choice is the name of the instance that got the best price-performance ratio based on equation 7.1 followed by the calculated capacity usage from equation 7.2. Given that an instance is capable of achieving the required throughput defined in the first column of each table, the cheapest option will be first choice followed by the second and third cheapest options. If any of the chosen instances does not change between the incrementation of 10k msgs/s, a range of rates is defined.

Messages/s	First Choice	Second Choice	Third Choice
Up to 10k-20k	t3.xlarge (35% - 70%)	c5.xlarge (18% - 36%)	m5.xlarge (20% - 41%)
Up to 30k-40k	c5.xlarge (54% - 72%)	m5.xlarge (61% - 81%)	F4s (67% - 89%)
Up to 50k	c5.xlarge (90%)	c5.2xlarge (71%)	m5.2xlarge (74%)
Up to 60k	c5.2xlarge (85%)	m5.2xlarge (89%)	c5.4xlarge (83%)
Up to 70k	c5.2xlarge (99%)	c5.4xlarge (97%)	-

Table 7.4: Instance price-performance evaluation for 1 queue

Messages/s	First Choice	Second Choice	Third Choice
Up to 10k-50k	t3.xlarge (18% - 88%)	c5.xlarge (13% - 65%)	m5.xlarge (14% - 69%)
Up to 60k	c5.xlarge (78%)	m5.xlarge (83%)	F4s (86%)
Up to 70k	c5.xlarge (91%)	m5.xlarge (97%)	t3.2xlarge (82%)
Up to 80k	t3.2xlarge (93%)	c5.2xlarge (76%)	m5.2xlarge (91%)
Up to 90k	c5.2xlarge (85%)	c5.4xlarge (63%)	m5.4xlarge (99%)
Up to 100k	c5.2xlarge (95%)	c5.4xlarge (70%)	-
Up to 110k-140k	c5.4xlarge (77% - 98%)	-	-

Table 7.5: Instance price-performance evaluation for 2 queues

Messages/s	First Choice	Second Choice	Third Choice
Up to 10k-50k	t3.xlarge (17% - 86%)	c5.xlarge (13% - 66%)	m5.xlarge (13% - 67%)
Up to 60k-70k	c5.xlarge (79% - 92%)	m5.xlarge (81% - 94%)	t3.2xlarge (76% - 88%)
Up to 80k-120k	c5.2xlarge (54% - 81%)	m5.2xlarge (62% - 92%)	c5.4xlarge (30% - 45%)
Up to 130k-140k	c5.2xlarge (88% - 94%)	c5.4xlarge (49% - 52%)	m5.4xlarge (71% - 76%)
Up to 150k-180k	c5.4xlarge (56% - 67%)	m5.4xlarge (81% - 96%)	-
Up to 190k-260k	c5.4xlarge (71% - 97%)	-	-

Table 7.6: Instance price-performance evaluation for 10 queues

Messages/s	First Choice	Second Choice	Third Choice
Up to 10k-40k	t3.xlarge (22% - 89%)	c5.xlarge (15% - 60%)	m5.xlarge (21% - 84%)
Up to 50k-60k	c5.xlarge (75% - 90%)	t3.2xlarge (46% - 55%)	c5.2xlarge (38% - 45%)
Up to 70k-100k	t3.2xlarge (65% - 92%)	c5.2xlarge (53% - 75%)	m5.2xlarge (68% - 98%)
Up to 110k-130k	c5.2xlarge (83% - 98%)	c5.4xlarge (42% - 50%)	m5.4xlarge (57% - 67%)
Up to 150k-180k	c5.4xlarge (54% - 73%)	m5.4xlarge (72% - 98%)	-
Up to 190k-260k	c5.4xlarge (77% - 96%)	-	-

Table 7.7: Instance price-performance evaluation for 200 queues

Each color in Tables 7.4 - 7.7 represent different instance types. The color gets darker for higher cost instances, based on Tables 7.1 - 7.3.

7.3 Discussion

PerfTest evenly distributes messages over all queues. Queues can handle different types of data which results in varying publish rates and package sizes. There is no telling if conducting new tests with uneven loads would give the same results. Tables 7.4 - 7.7 give an overview of which instance types obtain a better price-performance ratio. The tables support what Kotas et al. [20] concluded on the computational price of the computation optimized families on Amazon. Since results concluded that RabbitMQ is dependent on CPU it is clear that the C5 family is one of the better alternatives for enterprise solutions. Although t3.xlarge seems to be the best choice for up to 40k msgs/s, the cheaper price weighs heavily even though it is only 7$ cheaper than the c5.xlarge, which for most users is a negligible difference. As soon as the throughput requirement goes above 40k, the c5.xlarge instance takes its place, being able to handle almost twice the load.

With loads lower than 10k msgs/s, the tables show that t3.xlarge is the best option without considering other smaller instances that can manage the same workloads.

8 General Discussion

This chapter will discuss the general method used and the choices made throughout the study. Also, discussion about the ethical and societal aspects are included. Discussion about the results received are included in their respective chapter.

In section 3.1, the final broker architecture was chosen. The scalable and highly available RabbitMQ cluster had the number of shadowed queues set to 2. Rostanski et al. [25] concluded that having a highly available broker cluster would drastically decrease the performance. However, for this study PerfTest was used without acknowledgement or message persistence. Without message persistence activated the broker nodes do not need to utilize their memory to store messages during run-time. It is important to note that this is the main reason for memory not being a bottleneck for each test session. Given a good enough CPU in the host machine, the brokers are capable of achieving the same consumption rate as publish rate. The only case where memory usage was a deciding factor was if the test machine managed to achieve a high enough publish rate to overwhelm the broker, then messages were stored in a queue. So does having high availability give any benefits without acknowledgement or persistence? The load balancer on AWS assigns connections to a node using the flow-hash strategy. Assuming that an IoT sensor does not require acknowledgement in a production environment, the load balancer can be used to distribute master queues over multiple nodes. This way load it is evenly distributed amongst all nodes in the cluster and the solution achieves high availability with 2 backup nodes in case of a failure.

The solution used Docker as the host software to run the RabbitMQ server. An alternative approach tested and implemented, had the RabbitMQ server run directly on the virtual machine. By creating an AMI, instances could be launched and immediately have access to necessary libraries and software installed. The advantages of this approach would be shorter start up time for nodes to join the cluster. The current

solution has to wait an arbitrary period for the docker container to start up and execute the setup commands as seen in Algorithm 1. A solution running directly on the machine would eliminate this waiting period and execute commands directly. Furthermore, what has not been tested is how the throughput is affected when all traffic is forwarded through the Docker container. Despite added start up times and unknown effects on the throughput, the Docker solution was selected since it is a more general approach to be used across cloud platforms. The Docker container allowed for faster configuration times on new cloud platforms since few additional software had to be installed. The Docker solution also allowed for cross OS support.

Work in a wider context

If message brokers would be hosted on on-premise solutions, companies using them would have to employ people with the required competence. Knowledge of e.g. network security and server hardware would be needed as well. However, when as many as 70% of companies are moving their solutions to the cloud the need for this in-house competence is decreased. [11] This might in turns result is fewer jobs available.

It is important to look at cloud computing in the context of power consumption and energy efficiency. Moving to cloud computing decreases costs for companies that are dependent on scalable servers. Thereby, this allows smaller companies to grow. With these great benefits comes the environmental impact that cloud computing has. By creating large data centers and managing servers for millions of users, the environment could be affected negatively. Therefore it is important to strive for energy efficient architectures and underlying components. Deng et al. have more specifically proposed a framework for energy efficiency improvements on virtual machines. This framework could be used to effectively lower the energy consumption and thereby reduce the environmental impact of using cloud computing. [12]

With cloud services emerging, most companies no longer have the need for physical servers. There have been many incidents with data leaking from cloud storages. It is important to note that using a service on the cloud risks the integrity of data and leave responsibility to the companies.[11]

9 Conclusion

A scalable RabbitMQ cluster solution was designed and created in order to test the performance of the protocol AMQP over two of the most popular cloud platforms. The solution was designed to be reactive, in terms of being able to scale automatically based on workload and to be highly available. Availability was accounted for by including a load balancer in the solution to automatically forward traffic only to healthy nodes in the cluster. Having an available system is for enterprises an important factor in order to provide an operable and usable service for the users. In order to improve availability further, the term reliability was also of concern for this solution. As much as enterprises want their system to be operable and online, reliability is an important quality attribute that relates to the success of intercommunication. The solution was able to account for the reliability by replicating data over multiple nodes and was in the event of a node outage able to restore lost data. The purpose of the whole system architecture and design was to replicate a system to be used in production environments. The equivalent system setups deployed on the two cloud platforms allowed for comparisons between them and a performance study to be conducted.

It is clear through the results that both the cloud platforms and deployment architectures have a big effect on the system throughput. Starting with the configured architecture of the respective cloud platform, the results show that small optimizations on the platforms can have a significant impact on the throughput. On AWS the most significant improvements came from utilizing the network load balancer instead of the classic load balancer. The classic load balancer gave both highly varying throughput results between runs and a lower average throughput than with the network load balancer. On Azure, there only exists one load balancer and was therefore not an issue on this cloud platform. What instead caused the biggest difference in performance was the feature accelerated networking. Choosing a machine becomes important since Azure does not support the accelerated networking feature on all

machines. Selecting a machine not compatible with this feature can for a network dependent service have considerable impact on the overall throughput from client to server and vice versa. What can be concluded from these results is that the a and b-series on Azure should never be used for network intensive tasks. Choosing a machine from the d or f-series will be a better choice if network performance is important. Worth mentioning is the price class difference, where the b and d-series are labeled as an economic choice for development and testing whilst d and f-series has the general and computation labels respectively.
Furthermore, overall throughput conclusions between the AWS and Azure cloud platforms can be drawn from the results and discussions done in this study. AWS has had a higher throughput in the majority of the tests done. AWS was also better at utilizing the performance of multiple virtual CPUs when running the multiqueue tests. What could be seen as a trend throughout the tests were the performance degrations when adding a load balancer on AWS. The load balancer on Azure did, however, not have a substantial affect on the throughput but had in terms a lower general throughput than what was achieved on AWS.

Throughout the study, two instance families on these cloud platforms have been studied and tested, the general and the compute optimized family. It was concluded during the study that CPU was a more important component than memory. The compute family was therefore chosen as one of the families to test. Comparisons of the tests conducted on AWS between the compute optimized family and the general purpose family shows that the compute optimized family are able to utilize the advantages of a multicore processor. Not much throughput difference can be seen when running tests on a single queue. The real benefit of using a compute optimized family over the general purpose family comes when running heavier tests over multiple queues. This can especially be seen when comparing the c5 with the m5 and t3 machine families. Between the m5 and t3 general purpose machine families there are small differences in throughput. If the burst feature was deactivated on the t3 machines they could be seen as an equal or even better alternative in terms of throughput than the m5 machines. The compute optimized family on Azure did however not show any utilization of the multicore processor in the tests performed and had a relatively low throughput compared to AWS.

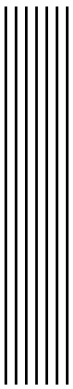
Based on the results and discussion of the price-performance ratios between the set of instances used for this thesis, a conclusion can be made on performance over cost. Because of the high performance and relatively low cost of the compute optimized instances, their price-performance ratio is high for any throughput. However, for lower throughput requirements the choice of instance type has less impact. For this study the t3.xlarge general purpose instance always had the highest score for the lower requirements. With low rates up to 20k messages/s almost any instance will work, even a t2.micro free tier instance. The compute optimized instances on Azure performed poorly on the tests in chapter 6 and therefore only the F4s instances made it to the top three choices. Even with tests being conducted twice with accelerated networking enabled, the results stayed the same. The results can help companies get an understanding of which instance types are most cost efficient for a given throughput threshold.

9.1 Future Work

There are many aspects of brokers in the cloud to further investigate.
Since PerfTest is quite limited to high loads there is an opening to develop a more customizable testing tool that simulate realistic workloads with exchanges, message persistence over multiple queues and variations in workload. This way, more realistic loads could be simulated to give a wider analysis of how RabbitMQ performs in production environments. PerfTest can also be used for a more distributed testing approach. Either PerfTest can be started as separate processes on the same machine or it can be distributed across multiple machines. The latter would give a more realistic workload since the data packets would not originate from the same source.

In addition to this study one could further investigate by including another big cloud platform like Google Cloud to see how it stands against the performance of AWS and Azure.

Since this study is focused on the difference of manually deployed solutions on the clouds, future work could be to look at the already existing broker solutions provided on the cloud from Bitnami, CloudAMQP or each cloud's respective BaaS.



Bibliography

[1] *AMQP 0-9-1 Model Explained — RabbitMQ*. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html (visited on 03/12/2019).

[2] *AWS Load balancer : Routing algorithm*. URL: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html (visited on 03/16/2019).

[3] *Azure Load balancer : Routing algorithm*. URL: https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview (visited on 03/22/2019).

[4] Bharati Wukkadada ; Kirti Wankhede ; Ramith Nambiar ; Amala Nair. "Comparison with HTTP and MQTT In Internet of Things (IoT)". In: *Conference on Inventive Research in Computing Applications (ICIRCA)* (July 2018).

[5] Erick M. Halter Chris Wolf. *Virtualization - From the Desktop to the Enterprise*. Apress, 2005, pp. 1–5.

[6] *Cloud Market Q3 Snapshot: Azure Is Fastest, But AWS Is Biggest – AWSInsider*. URL: https://awsinsider.net/articles/2018/11/01/azure-fastest-but-aws-biggest.aspx (visited on 02/10/2019).

[7] *Compute Optimized Instances - Amazon Elastic Compute Cloud*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html (visited on 02/23/2019).

[8] *Consumer Acknowledgements and Publisher Confirms — RabbitMQ*. URL: https://www.rabbitmq.com/confirms.html (visited on 03/05/2019).

[9] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. "Elasticity in cloud computing: a survey". en. In: *annals of telecommunications - annales des télécommunications* 70.7 (Aug. 2015).

[10] *Create a computer with accelerated network | Microsoft Docs*. URL: https://docs.microsoft.com/sv-se/azure/virtual-network/create-vm-accelerated-networking-cli (visited on 03/28/2019).

[11] Deepak R Bharadwaj, Anamika Bhattacharya, and Manivannan Chakkaravarthy. "Cloud Threat Defense – A Threat Protection and Security Compliance Solution". In: *2018 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)* (Nov. 2018).

[12] Dongyan Deng, Kejing He, and Yanhua Chen. "Dynamic virtual machine consolidation for improving energy efficiency in cloud data centers". In: *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)* (Aug. 2016).

[13] Borislav S. Dordevic, Slobodan P. Jovanovic, and Valentina V. Timcenko. "Cloud Computing in Amazon and Microsoft Azure platforms: Performance and service comparison". In: *2014 22nd Telecommunications Forum Telfor (TELFOR)* (Nov. 2014).

[14] Kemme Gascon-Samson Garcia and Kienzle. "Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud". In: *2015 IEEE 35th International Conference on Distributed Computing Systems* (July 2015).

[15] *General Purpose Instances - Amazon Elastic Compute Cloud*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html (visited on 04/10/2019).

[16] Lindita Nebiu Hyseni and Aferdita Ibrahimi. "Comparison of the cloud computing platforms provided by Amazon and Google". In: *2017 Computing Conference* (July 2017).

[17] V. M. Ionescu. "The analysis of the performance of RabbitMQ and ActiveMQ". In: *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)* (Sept. 2015).

[18] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud". In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science* (Nov. 2010).

[19] P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga, and D. Kobayashi. "A scalable and low-cost MQTT broker clustering system". In: *2017 2nd International Conference on Information Technology (INCIT)* (Nov. 2017).

[20] Charlotte Kotas, Thomas Naughton, and Neena Imam. "A comparison of Amazon Web Services and Microsoft Azure cloud platforms for high performance computing". In: *2018 IEEE International Conference on Consumer Electronics (ICCE)* (Jan. 2018).

[21] *Load Balancer Types - Amazon Elastic Container Service*. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html (visited on 03/15/2019).

[22] *Memory Optimized Instances - Amazon Elastic Compute Cloud*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/memory-optimized-instances.html (visited on 05/10/2019).

[23] *RabbitMQ PerfTest*. URL: https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/ (visited on 02/08/2019).

[24] *Reliability Guide — RabbitMQ*. URL: https://www.rabbitmq.com/reliability.html (visited on 02/02/2019).

[25] Maciej Rostanski, Krzysztof Grochla, and Aleksander Seman. "Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ". In: *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems* (Oct. 2014).

[26] Mohammed A. Saifullah and Dr. M. A. Maluk Mohammed. "Scalable Load Balancing using Enhanced Server Health Monitoring and Admission Control". In: *5 IEEE International Conference on Engineering and Technology* (Sept. 2015).

[27] *vCPU - Amazon Elastic Compute Cloud*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html (visited on 03/06/2019).

[28] *What is Azure Load Balancer? | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview (visited on 02/14/2019).

[29] *What is cloud computing? A beginner's guide | Microsoft Azure*. URL: https://azure.microsoft.com/en-in/overview/what-is-cloud-computing/ (visited on 02/02/2019).

[30] *What is IaaS? Infrastructure as a Service | Microsoft Azure*. URL: https://azure.microsoft.com/en-in/overview/what-is-iaas/ (visited on 02/04/2019).

[31] *What is virtualization technology & virtual machine? | VMware*. URL: https://www.vmware.com/solutions/virtualization.html (visited on 04/22/2019).

[32] Tetsuya Yokotani and Yuya Sasaki. "Comparison with HTTP and MQTT on required network resources for IoT". In: *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)* (Sept. 2016).