POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Latency Analysis in Cloud Native Environment

Supervisor **Candidate**

Prof. Fulvio RISSO **Riccardo MARCHI**

April 2021

"Don't be afraid to give up the good to go for the great."
John D. Rockefeller

Abstract

In the recent years, latency is continuously assuming more and more relevance due to the demanding applications being developed. In fact, with the evolution of Industry 4.0, the arrival of autonomous driving, the increase in the videoconferencing and the growth of cloud gaming platforms, the end-to-end delay is becoming every day more a strict constraint than a marginal aspect. While it is crucial that each application provides a high-level user experience, safety-critical environments are even more demanding, as missing the requirements can have catastrophic effects.

Nevertheless, besides propagation delays strictly depending on the physical distance between the two endpoints, it is necessary to consider that the underlying infrastructure, as well as higher level communication protocols, are also factors influencing network performance. Indeed, the complexity of nowadays adopted technologies (i.e. orchestration platforms such as Kubernetes, as well as network middle-boxes like load balancers and firewalls) could lead to unexpected drawbacks and even if edge computing employment is promising to reduce propagation time, this distributed paradigm may not guarantee an equal improvement in the performance perceived by applications. Is application-level latency reflecting the network-level one or is the difference in their behaviors remarkable? This thesis focuses on the analysis of what are the possible drawbacks of the cloud native environment, taking a better look at those aspects and components that could impact the performance of the end-to-end communication of a micro-service application.

At first, a tool simulating a client/server application has been designed in order to gather different metrics about the communication between the two endpoints; then it has been deployed in many environment setups generating a diverse set of output data. These results showed that some scenarios worked as expected but there were some interesting behaviors that disclosed significant delay. Overall, the analysis revealed that there are not absolute problems, but depending on the specific situations some low-level tuning is typically required to better leverage the available bandwidth and achieve low latency. At the same time, small problems (e.g. a malfunctioning link corrupting some packets) turned out to have catastrophic effects, drastically decreasing the network metrics perceived by the final applications. Based on each specific scenario, the outcome has been carefully observed and some evaluations and possible solutions to the discovered problems are presented.

Table of Contents

List of Figures

Acronyms

API

Application Programming Interface

BDP

Bandwidth-Delay Product

E2E

End-to-End

ECMP

Equal-Cost Multi-Path

ICMP

Internet Control Message Protocol

ISP

Internet Service Provider

IXP

Internet eXchange Point

JSON

JavaScript Object Notation

K8S

Kubernetes

MSS

Maximum Segment Size

NIC

Network Interface Card

PoliTo

Politecnico di Torino

QoS

Quality of Service

REST

REpresentational State Transfer

RTT

Round-Trip Time

SSL

Secure Sockets Layer

TCP

Transmission Control Protocol

TLS

Transport Layer Security

UDP

User Datagram Protocol

Chapter 1

Introduction

In recent years, companies working in every kind of field are looking into the enhancement of their IT infrastructure along with their business growth, in order to exploit new technologies in the market and improve productivity as much as possible. This is due to the desire of managing all working processes easily, quickly and also keeping the expenses at the minimum.

Nowadays the most used solution to solve this problem infrastructure-wise is the adoption of a Cloud Native environment, which substantially impacts the efficiency of the companies allowing them to handle on premise machines as a unique huge amount of resources. However, as this technology brings many positive aspects, there are also negative sides such as in terms of additional overhead, that could make a difference in the way a company can take advantage of it.

One typical aspect which needs great attention regards the performance of latency-sensitive applications: the development of information and communication technology for healthcare, industrial processes, transport services, or entertainment applications are all use cases where there is the necessity to keep the end-to-end delay limited to about few milliseconds, so that they provide high availability and their potential can be properly exploited. To better explain the problem, it would be impossible to operate in a tele-surgery with no immediate feedback, a small movement beyond the necessary might kill the patient; in a cloud gaming scenario, input lag and no quick visual response would make the gameplay uncontrollable; a round-trip time over 200ms in a videoconference results in an annoying delay in the communication between participants [1]. Moreover, with the growing trend of offloading tasks from an access device to edge computing resources it is mandatory

to maintain a certain connection stability [2]. The overall latency-wise performance is a key factor for the provider to earn high revenues, therefore it cannot be overlooked.

As a further motivating example, during an application setup in the Netgroup data center at Politecnico di Torino, it has been detected a surprisingly high amount of latency and jitter, questioning the benefits of cloud computing when dealing with high bit-rate and relevant message size. The application was an image recognition service where the client periodically sends the captured frame and the server responds with a small payload indicating the position of possible human faces inside it. The message exchange is done with REST APIs and the latency perceived was very unstable with peaks of over 100ms, with the client deployed in an on-premise server in Turin and the server in a Kubernetes cluster in GCP Zurich region (the network ping is around 10ms). This problems led to the analysis done in this thesis work, expanding the investigation on a larger scale.

1.1 Goal of the thesis

The goal of this thesis is to analyse what are the possible drawbacks in terms of communication latency of the Cloud Native environment, giving an overview on the problems of the technologies involved and looking for those aspects that could impact the performance of the end-to-end communication between client and server of a microservice architecture application, especially network-wise.

By default, there are many problems affecting the latency: different network operators involved, QoS prioritization, transmission time, routing time and traffic load are most common factors, but with the new technologies involved there could be extra delay introduced by these components, that handle the packet much more than before. It is interesting to understand if the overhead added from these new elements is negligible or in some case could cause catastrophic performance deterioration.

The analysis is done by defining many different environment setups which represent real case scenarios and by using a client-server program developed and sharpened during the thesis work acting as the deployment of an application. This configurable tool registers the latency of the communication on many levels and keeps track of the parameters of the connection and then uses the gathered data to generate some graphs that are immediately available for consultation by the user.

1.2 Structure of the work

The thesis is structured as follows:

- **Chapter 2** gives an overview of the technologies involved in the discussion, giving an introduction of both the TCP main traits and the Cloud Native environment, active parts of the analysis;
- **Chapter 3** presents the state of the art around the purpose of the analysis, showing the problems known until now and the related works published in literature;
- **Chapter 4** aims at explaining what we are looking for in the analysis and thoroughly describes the tool developed to gather the data we are interested in;
- **Chapter 5** illustrates the environments designed to do the measurements and analyses the results collected also highlighting the peculiarities that stood out;
- **Chapter 6** closes the dissertation with final comments and proposals on what to focus on in the future.

Chapter 2

Background

In this chapter the technologies involved in the analysis are described, making an exhaustive overview of how they work and highlighting their distinctive traits that make them so much employed in nowadays information systems.

The first technology to be presented is TCP, the main transport layer protocol used in the Internet, with a better look at its connection management and congestion control algorithm, whose behaviors could play major roles in the E2E communication performance.

Later the Cloud Native environment is introduced, showing the microservice architectural pattern with particular emphasis on why it is so important and useful and presenting the platform employed for the deployment of containerized services, Kubernetes.

2.1 Transmission Control Protocol

Nowadays, Internet, or more generally any TCP/IP network, makes two distinct transport-layer protocols available to the upper layer: the first one is UDP (User Datagram Protocol), which offers an unreliable and connectionless service, while the second is TCP (Transmission Control Protocol), that provides a reliable, connection-oriented, full-duplex and in-sequence communication channel to the invoking application above. Even though both protocols are equally important, only the latest is the one taken into account in this analysis, since it is the most suitable when it comes to a reliable data transfer.

The primary purpose of the TCP is to provide a reliable connection service

between pairs of processes, therefore in order to do so on top of a less dependable internet communication system, facilities are required in the following areas:

- **Basic Data Transfer**: TCP is able to transfer a continuous stream of octets in each direction between its users by packaging some number of octets into segments for transmission through the Internet system. In general, TCP decides when to block and forward data at its own convenience.

- **Reliability**: TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP peer. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

- **Flow Control**: TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

- **Multiplexing**: To allow for many processes within a single Host to use TCP communication facilities simultaneously, TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection.

- **Connections**: The reliability and flow control mechanisms described above require that TCP initializes and maintains certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection. Each connection is uniquely specified by a pair of sockets identifying its two sides.

This chapter is based on [3],[4] and [5].

2.1.1 TCP Connection Management

The most important aspect that sustains the reliability of the TCP is the fact that it is connection-oriented, so the establishment of the connection is the sine qua non for the communication between one application process and another. In order to do that, they must send some preliminary segments to each other to exchange the parameters of the ensuing data transfer, using a procedure called *Three-way Handshake*.

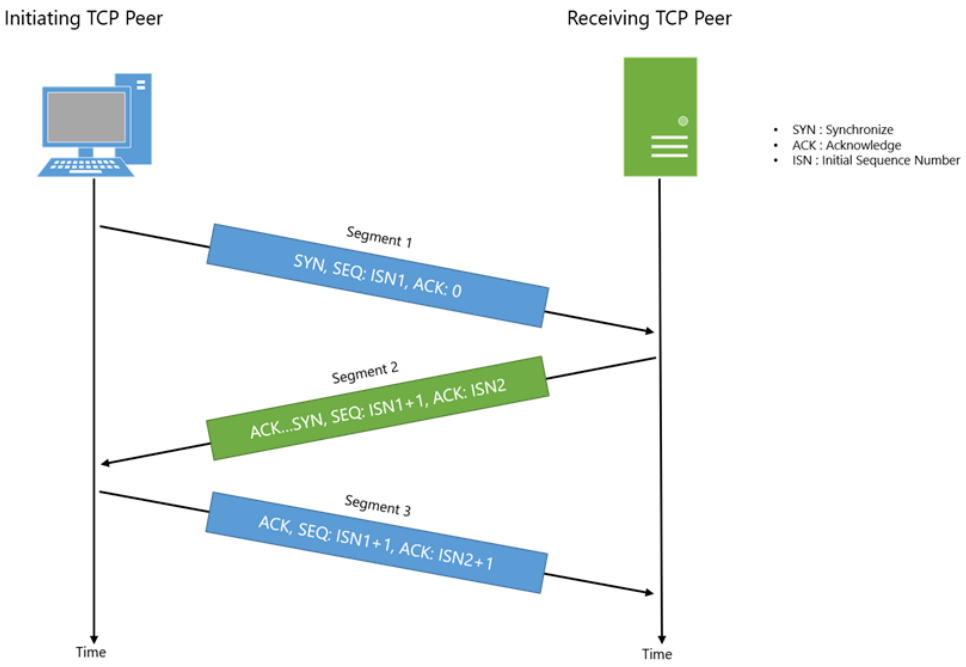


Figure 2.1: TCP Three-way Handshake

Figure 2.1 depicts the three segments of the connection establishment between two peers without involving the intermediate network elements, because they do not maintain any TCP state. This topic is important because it can significantly add to perceived delays, especially when more than one TCP session are involved. Supposing a client wants to send data to a server, the first three steps to initiate a TCP connection are:

1. *Step 1*: The client-side TCP process first sends through its socket a special TCP segment to the server-side TCP socket, without containing any application-layer data. This segment is called **TCP SYN**, since in its header the SYN flag is set to 1, indicating the intention to establish a connection. In addition,

the client randomly chooses a client_initial_sequence_number and puts it in the sequence number field of this segment's header. Then the packet is forwarded to the server within an IP datagram.

2. *Step 2*: Once the IP datagram containing the TCP SYN segment arrives at the server host (assuming it does arrive), the server recognises the TCP SYN segment and allocates the TCP buffers and variables for the connection, responding to the client with a **TCP SYNACK** segment indicating that the connection to the server is granted. Again no application-layer data is present and the SYN flag is set to 1, the header acknowledgement field is set to client_initial_sequence_number + 1 and the server chooses its own random server_initial_sequence_number putting this value in the sequence number field.

3. *Step 3* : The client receives the SYNACK segment and allocates buffers and variables itself. Then it sends the server another segment acknowledging the connection-granted message with a simple **TCP ACK**. The SYN flag is set to 0 and the payload may contain application-layer data.

Eventually the connection is established and the client and the server hosts can send segments containing data to each other.

At a certain point, either of the two hosts can end the connection, by sending the **TCP FIN** segment with the FIN flag set to 1 to the other peer, that will acknowledge it with an ACK segment. Then the same procedure will be executed the other way around, concluding with the connection shutdown and the deallocation of the resources in both hosts.

2.1.2 TCP Congestion Control

The idea behind the congestion control algorithm implemented by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion, so that when there is no congestion one can increase the send rate and decrease it if otherwise. So there are three questions to answer about this subject:

- How to limit the send rate?
- How to perceive congestion?
- Which mechanism to put in place to change the send rate?

In order to address the first aspect, TCP keeps track of an important variable, the **congestion window** (denoted as cwnd), that imposes a constraint on the rate at which a TCP sender can send traffic into the network. This value is used to limit the amount of unacknowledged data at the sender and therefore indirectly limit the sender's rate. By adjusting it, the sender can then tune the rate at which sends data into its connection, so that the bandwidth usage is optimized.

While addressing the second question, it is important to first point out that a *loss event* is represented by the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. When there is a lot of congestion in the network, the buffers of the nodes along the path are full and start to drop datagrams containing segments of the communication, resulting in the loss event just described above. Every loss event is an indication of congestion. Otherwise, if there are no such events and the acknowledgements are successfully received by the sender, it will start to increase its congestion window size at the same rate he receives these ACKs, determining how fast the data transmission improves its speed. Because TCP use acknowledgements to trigger its increase in congestion window size, this protocol is said to be **self-clocking**.

Taking into account the third point, the approach of the TCP to the problem follows three principles:

- *A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost*: as explained before, a timeout or three duplicate ACKs indicate a loss event, therefore congestion;
- *An acknowledgement segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment*: if everything proceeds well, it means that the network is not congested and the congestion window size can be increased;
- *Bandwidth probing*: TCP's strategy for adjusting the transmission rate is to increase its send rate until a loss event occurs, at which point it will be decreased. Once backed off a bit, it will retry to probe again to see if the congestion onset rate has changed.

This completes the overview of the TCP congestion control. Now the **TCP congestion control algorithm** will be presented in its three major phases, also depicted in Figure 2.2 inside the algorithm finite state machine: *slow start*, *congestion*

avoidance and *fast recovery*.

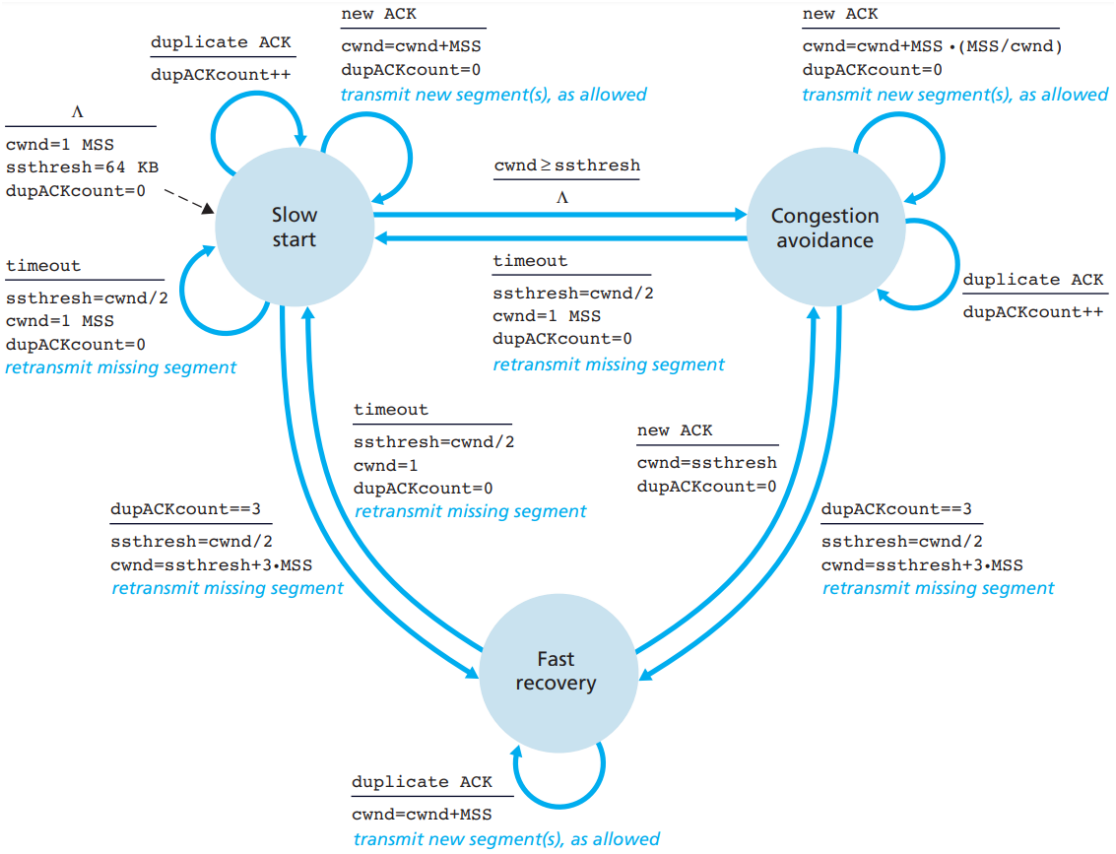


Figure 2.2: TCP Congestion Control Algorithm

Slow Start

When a TCP connection begins, the value of the congestion window is usually initialized to a small value of 1 MSS (Maximum Segment Size), sending just one segment per RTT (Round-Trip Time). Since at that rate the bandwidth is surely underutilized, the TCP sender would like to quickly find out the best cwnd size in order to improve as fast as possible. In the **slow-start** state, the congestion window is increased by 1 MSS every time a transmitted segment is first acknowledged. This process results in a doubling of the send rate every RTT, thus it starts slow but grows exponentially during this phase.

This exponential growth will stop in case of a loss event. If this event is a timeout, the TCP sender will set the cwnd back to 1 MSS and begins the slow

start process anew, but it will also set the *slow start threshold* variable, known as ssthresh, to cwnd/2, that is half of the value of the congestion window value when congestion was last detected. If the loss event is represented by the detection of three duplicate ACKs, the TCP will perform a fast retransmit and will enter the *fast recovery* state discussed below. The last way in which the exponential growth will stop is when the cwnd reaches or surpasses the ssthresh: since this last value is half the value of the congestion window when the congestion was last detected, it might be reckless to keep doubling it once again, risking to fall in the same state previously encountered. Once surpassed the slow start threshold, the slow start ends and the TCP transitions into congestion avoidance mode.

Congestion Avoidance

The congestion avoidance state has the goal to prevent another loss event in the imminent time. In fact, when the TCP reaches this state, it has just left the slow start due to the surpassing of the slow start threshold. Being the congestion window more or less half the value it was when the congestion last occurred, it is important to keep in mind that doubling its value again would mean recreating the previous risky situation. So rather than doing that, when in congestion avoidance, TCP proceeds more conservative increasing the cwnd by just one MSS every RTT.

The linear increase of the window stops like in the slow start, that is with a loss event. With a timeout it behaves the same way as the previous state: ssthresh is set to half the congestion window value at congestion occurrence and cwnd is set to 1 MSS. In the case the loss event is represented by a triple duplicate ACK, since the sender is still delivering packets to the receiver, it should behave in a more delicate way than with a timeout: the cwnd is halved and then increased 3 MSS (for good measure to account for the triple duplicate ACKs received) and again the slow start threshold is set as half the value of the congestion window when the loss event occurred. Once completed these steps, the TCP transitions in the fast recovery state.

Fast Recovery

Before introducing this behavior, it is important to notice that this is not mandatory but strongly recommended.

When TCP is in fast recovery, the congestion window is increased by 1 MSS for every duplicate ACK received for the missing segment that caused the transition

to this state and waits for the ACK to arrive: if eventually it is received, TCP transitions back to congestion avoidance, otherwise it goes back to the slow start state.

High-Bandwidth Congestion Control Algorithm: TCP CUBIC

There are many congestion control algorithm developed through the years, that implements a different behavior for this particular state. **TCP Tahoe** is one of the first ones and does not include any fast recovery phase, but then evolved in a newer version called **TCP Reno** which incorporated it. Nowadays the most used is **TCP CUBIC**, adopted by Linux kernels as default congestion control algorithm since version 2.6.18.

This version of the TCP congestion control algorithm is an extension to the current TCP standards. The particular distinctions that allows to improve scalability and stability under fast and long distance networks is the adoption of a cubic function instead of a linear one to increase the sender congestion window. This algorithm wants to contrast the low utilization of the connection after a congestion event in a network with a large bandwidth-delay product (BDP) [6], that is the equivalent to the maximum amount of data on the network circuit at any given time, i.e. data that has been transmitted but not yet acknowledged.

TCP CUBIC follows four design principles:

1. For better network utilization and stability, CUBIC uses both the concave and convex profiles of a cubic function to increase the congestion window size, instead of using just a convex function.

2. To be TCP-friendly, CUBIC is designed to behave like Standard TCP in networks with short RTTs and small bandwidth where Standard TCP performs well.

3. For RTT-fairness, CUBIC is designed to achieve linear bandwidth sharing among flows with different RTTs.

4. CUBIC appropriately sets its multiplicative window decrease factor in order to balance between the scalability and convergence speed.

As showed in Figure 2.3, when a packet loss occurs, the window shrinks but immediately starts to grow quickly until it reaches the cwnd size of the latest congestion occurrence. Here its growth almost becomes zero, but considering that

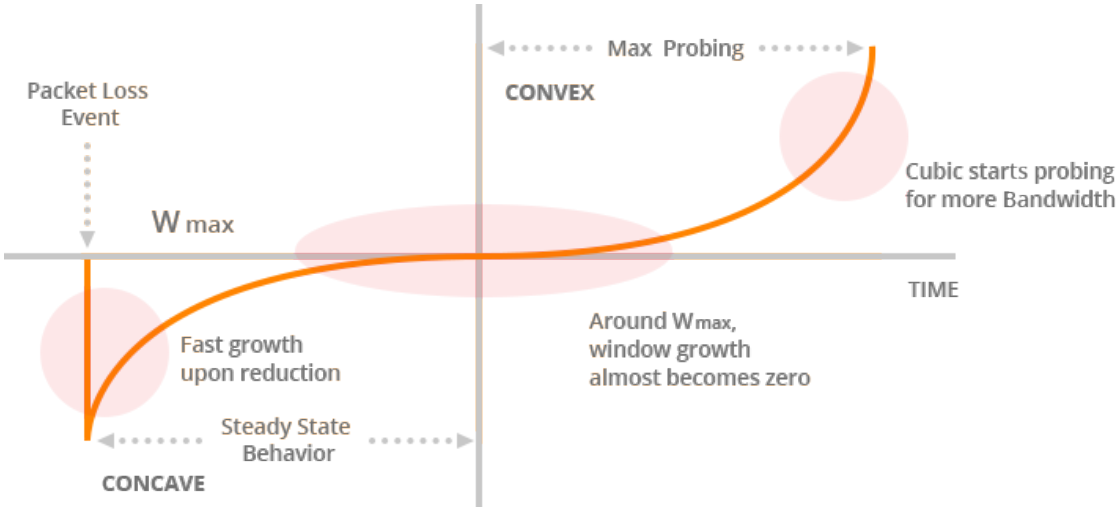


Figure 2.3: TCP CUBIC function

the internet bandwidth is always changing, after some time almost sleeping, the TCP slowly starts to probe for more bandwidth until the next loss event occurs.

TCP CUBIC is used as default congestion control algorithm in nowadays Linux systems since it is designed to be less aggressive and fairer to Standard TCP in bandwidth usage than the previous algorithms while maintaining their strengths such as stability, window scalability, and RTT fairness.

2.1.3 Transport Layer Security

TCP is a reliable communication channel, but this does not mean that it allows secure data exchange. In fact, it does not provide three important properties: *confidentiality*, *data integrity* and *peers authentication*.

Secure Sockets Layer (SSL) is a protocol relying on TCP which provides these features to connections, then evolved in the later standardized version 3 called **Transport Layer Security (TLS)**.

Nowadays TLS is broadly employed and even though it technically resides in the application layer, it is seen as a transport protocol that provides security services to TCP default characteristics. For this particular reason, TCP handshake is not enough and a TLS channel establishment procedure is needed too.

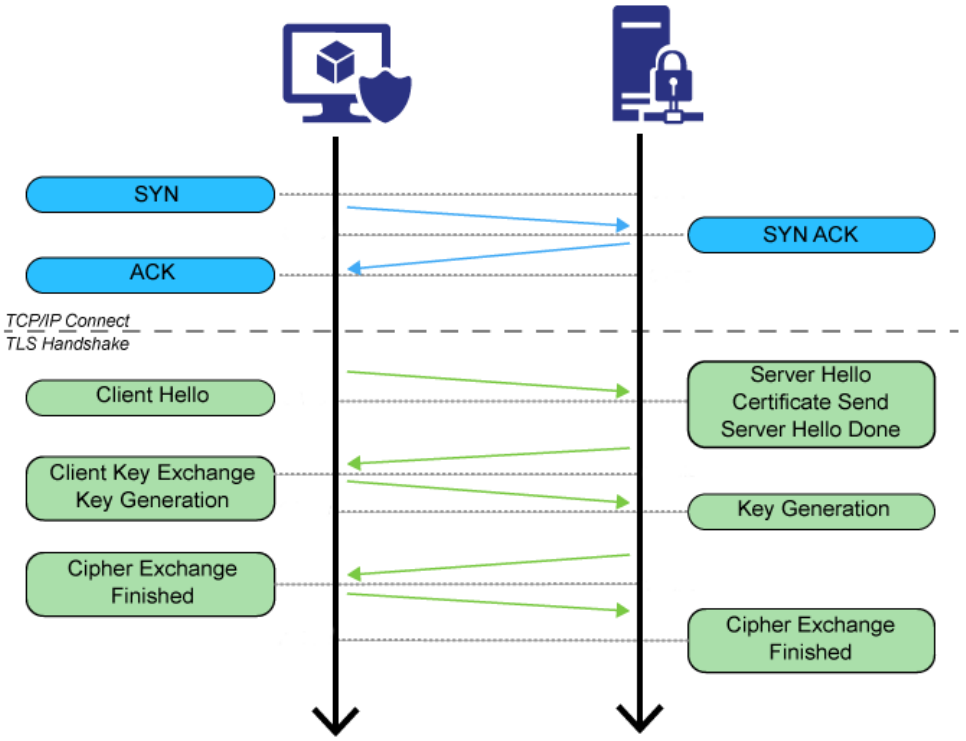


Figure 2.4: SSL/TLS Handshake

TLS Channel Establishment Phases

Figure 2.4 shows the steps to create a secure channel between a client and a server. Once established the TCP connection, the *TLS handshake* takes place:

1. *Negotiation phase*: The client specifies its highest TLS protocol version it supports along with a client nonce. The server chooses the the version and sends back a message containing the server nonce, the cipher suite and its certificate. Once verified the certificate, the client generates the Pre-Master Secret (PMS) and sends it to the server using the public key of the server. Both peers generate the Master Secret from the PMS and use it to provide the keys to the employed cryptographic algorithms.

2. *Client and server* ***ChangeCipherSpec*** *phase*: The client and the server send each other an authenticated and encrypted *Finished* message to verify the positive outcome of the handshake.

3. *Application phase*: The handshake is complete and the application protocol is enabled. Now the two peers can exchange data in a secure way.

This procedure, especially with older version of the protocol, includes further latency in the whole picture of data exchange and could be a huge factor, especially when the RTT between the peers is significant.

2.2 Cloud Native Computing

Cloud native computing is an approach in software development that utilizes cloud computing to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Technologies such as containers, microservices, serverless functions and immutable infrastructure, deployed via declarative code are common elements of this architectural style. This approach uses an open source software stack to deploy applications as microservices, packaging each part into its own container, and dynamically orchestrating those containers to optimize resource utilization.

Before the advent of the cloud, the computing and storing resources of a company were handled via virtual machines, due to their flexible use of resources keeping executing environment well isolated. Then, nowadays cloud providers started to provide their unused resources to those who needed on demand computing, generating new incomes.

Depending on the different requests of the clients, the provider can offer four delivery models:

- **Hardware as a Service (HaaS)**: the operator provides physical hardware and facilities while the client is responsible to handle everything else. It is rarely used, unless a high level of security is required. It can be also called *On-premise Environment*.

- **Infrastructure as a Service (IaaS)**: the operator provides the infrastructure with networking, computing and storage, while the client is able to handle the virtual environment created as he wants with maximum flexibility.

- **Platform as a Service (PaaS)**: the operator provides a set of services that can be exploited by the client in order to run its proprietary software, interfacing with them through public APIs.

- **Software as a Service (SaaS)**: the operator provides complete applications to users, so that he does not need to write a single line of code or have the burden of maintenance and support.

Figure 2.5 represents the responsibilities of provider and customer depending on the delivery model.

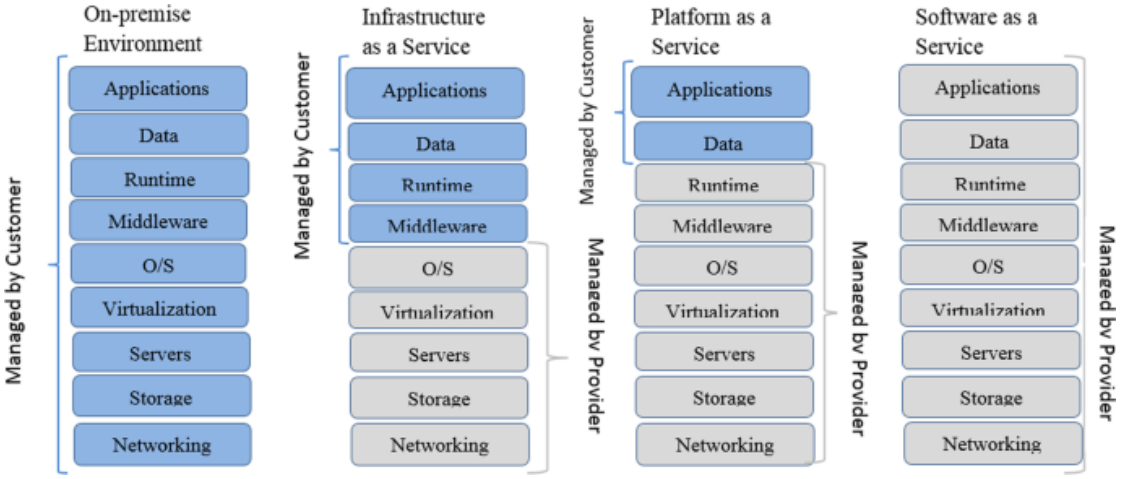


Figure 2.5: Cloud delivery models

Cloud computing has many advantages, whose impact changes depending on the model employed:

- *Elasticity*: the services needed can be provided on-demand very quickly and scaled up or down in an easy way;
- *Availability*: providers guarantee business continuity, accessible anytime from anywhere;
- *Separation of concerns*: users do not need to know anything about the underlying infrastructure or handle infrastructure-level details, therefore they can focus on their business and no particular technical knowledge is required;
- *Instant software updates*: toolkit software or entire applications automatically updated by provider;
- *Unlimited resources*: user can asks for more resources as he likes;
- *Increased data reliability*: provider offers data replication and redundancy;

- *Geo-localized services*: user can selects the region where to deploy its software as he prefers;

- *Economic considerations*: user pays as needed and the provider is able to spend less with resource pooling.

Of course, besides all the positive sides it brings, the new paradigm has also some limitations. Without Internet connection, the user cannot access his application or data and the interaction with the cloud could be problematic in case of a narrow network bandwidth. Therefore, it is crucial to keep in mind that this approach is not always appropriate for low-latency, safety-critical or aggressive data transmission applications. Moreover, some delivery models have small drawbacks: PaaS has not standard API, so applications are not portable across cloud-provides platforms, while SaaS could have less features than what we find in OS-based similar tools. Last but not least, the user loses the control over the generated data, since the storage of the data is not physically owned by him.

In order to address the first problematic of the cloud computing approach, an extension of it emerged, called **Edge/Fog Computing**, moving the computation and storage resources at the edge of the network with the intent to reduce latency for constraints demanding applications.

2.2.1 Microservices Architecture

Up until few years ago, the applications developed were following the monolithic model, that is a unique package containing all the services needed for its execution handling multiple related tasks. As imaginable, given their broad scope, these programs tend to have huge code bases, hence making a small change in a single function can require compiling and testing the entire platform, which goes against the agile approach today's developers favor. Therefore it was important to find a way to separate the separable and pull out a set of small, loosely coupled components that today take the name **Microservices**.

Microservices are both an architecture and an approach of writing software. With them, applications are broken down into their smallest components, independent from each other. Instead of the traditional, monolithic principle where everything is built into a single piece, microservices are all separated and work together to accomplish the same tasks. Each of these components, or processes, is a microservice. This approach to software development values granularity, being lightweight, and

the ability to share similar process across multiple apps. It is a major component of optimizing application development towards a cloud-native model.

The advantages of microservices are:

- *Resilience*: even if a microservice breaks in any way, it does not harm the entire application and can be immediately replaced with a new or an already present instance of the same service, guaranteeing zero downtime;
- *Scalability*: it is possible to create multiple instances of a microservice and place them in different machines, scaling the replicas depending on the traffic towards that service;
- *Agility*: the loading of a microservice instance is faster than the virtual machine bootstrap time, therefore it is possible to quickly start multiple microservices, in different versions, replace them with newer version and share an instance with many clients, all without significant effort;
- *Flexibility*: as long as they can interface with each other in a standard way, usually through REST APIs, each service can be developed using a different programming language, so that it is easier to select the most suitable one for a certain task.

As someone could expect, this approach has also drawbacks: with the separation of the services, debugging is very hard, because a single request will contact more than one microservice. For the same reason, also observability is difficult to achieve. Moreover, the decoupling of the services in containers results in a more complex architecture and an increased risk of security threats. Finally, due to the technologies involved in the management of these services and the high probability that a client request will require to contact more than one of them, this will cause, of course, a degradation in end-to-end delay, which should be addressed to preserve a high-level user experience and meet latency-critical applications constraints.

2.2.2 Kubernetes

Early on, organizations ran applications on physical servers. There was no way to define resource boundaries in this environment, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a

result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

As a solution, virtualization was introduced, allowing to run multiple Virtual Machines (VMs) on a single physical server. Virtualization keeps applications isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another one. Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines. Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware. Therefore the overhead of this solution is still remarkable and the management of each instance lifecycle and configuration is hard to handle in huge scale.

Eventually, Microservices application are deployed in containers, a form of operating system virtualization, but more lightweight, portable and with significantly less overhead. Of course containers are a good way to bundle and run your applications, but in a production environment it is necessary to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another one needs to start.

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications, that facilitates both declarative configuration and automation. Its clusters can span hosts across on-premise, public, private, or hybrid clouds. For this reason, Kubernetes is an ideal platform for hosting cloud-native applications that require rapid scaling. [7]

This orchestrator is API driven and follows the *Infrastructure as a Code* philosophy, letting the tenant to describe the logic with declarative programming without the definition of the control flow. In fact all the resources inside a K8s cluster are identified with a YAML file. The approach of this platform is Control Loop oriented, since it continuously checks the state of each resource under its control and verifies if it matches the specification provided, otherwise it makes the object converge to the desired status.

Kubernetes Components

Figure 2.6 represents the architecture of a Kubernetes platform deployment.

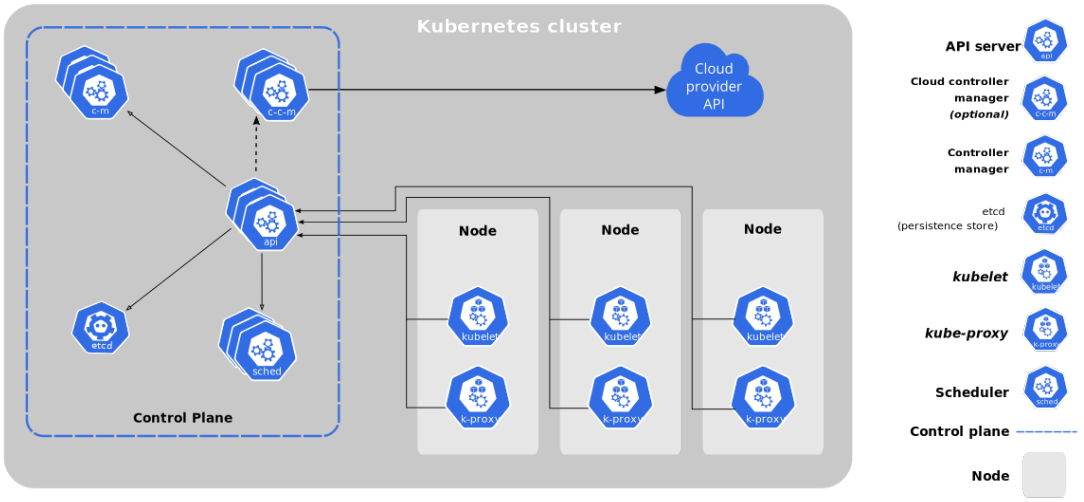


Figure 2.6: Kubernetes architecture.

When you deploy Kubernetes, you get a cluster. A Kubernetes cluster consists of a set of worker machines, each one called *Node*, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host one or more *Pods* that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

The control plane's components make global decisions about the cluster, as well as detecting and responding to cluster events. Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine that is called *master node*, and do not run user containers on it.

Components of the control plane are:

- *kube-apiserver*: the API server exposes the Kubernetes API and represents the front end for the Kubernetes control plane. kube-apiserver is designed to scale horizontally – that is, it scales by deploying more instances. It is possible to run several instances of kube-apiserver and balance traffic between those instances;

- *etcd*: it is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data;
- *kube-scheduler*: it is the control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on;
- *kube-controller-manager*: it is the control plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process;
- *cloud-controller-manager*: it is the control plane component that embeds cloud-specific control logic to interact with the underlying cloud providers.

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Node components are described below:

- *kubelet*: it is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. kubelet doesn't manage containers which were not created by Kubernetes;
- *kube-proxy*: it is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster;
- *Container runtime*: container runtime is the software that is responsible for running containers.

Kubernetes API

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another. The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes. Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API. However, you can also access the API directly using REST calls.

To make it easier to eliminate fields or restructure resource representations, Kubernetes supports multiple API versions, each at a different API path. Versioning is done at the API level rather than at the resource or field level to ensure that the API presents a clear, consistent view of system resources and behavior, and to enable controlling access to end-of-life and/or experimental APIs. To make it easier to evolve and to extend its API, Kubernetes implements API groups that can be enabled or disabled. API resources are distinguished by their API group, resource type, namespace and name. The API server handles the conversion between API versions transparently: all the different versions are actually representations of the same persisted data. The API server may serve the same underlying data through multiple API versions.

Any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, Kubernetes has designed the Kubernetes API to continuously change and grow. The Kubernetes project aims to not break compatibility with existing clients, and to maintain that compatibility for a length of time so that other projects have an opportunity to adapt. In general, new API resources and new resource fields can be added often and frequently. Elimination of resources or fields requires following the API deprecation policy, to avoid any misbehavior with the existing users. [8]

Deployment and Service Resources

When deploying an application to a Kubernetes cluster, the main K8s object that handles the practical part of creating the instances is the **Deployment** resource. It is a higher abstraction of a group of different resources that manage the creation of the desired application. With this element it is possible to define particular features the application needs but also aspects the tenants wants to control. In fact, for example, here we define the application itself, its input parameters and the *ReplicaSet* requested, that is the number of pods with that application that we want to run in parallel.

In case of a change in the deployment image field, that could be represented by an upgrade to a newer version of the application, Kubernetes puts in place the so called *Rolling Update*, that is updating Pods instances with new ones in an incremental way: at first it makes sure that the new instances are up and running and eventually it stops the older ones. By doing so, a zero downtime is assured, guaranteeing the total availability of the involved application.

But it is really important that the connection to these pods of the inbound traffic is orchestrated the best way possible, indicating a single entity that represents all the replicas of that application. The **Service** resource is an abstract way to expose an application running on a set of Pods as a network service. With Kubernetes it is not necessary to modify the application itself to use an unfamiliar service discovery mechanism, because Kubernetes gives Pods their own IP addresses and a single DNS name/virtual IP address for a set of Pods, and can load-balance across them. Figure 2.7 represents the service relationship with the deployment of a set of four replicas of an application.

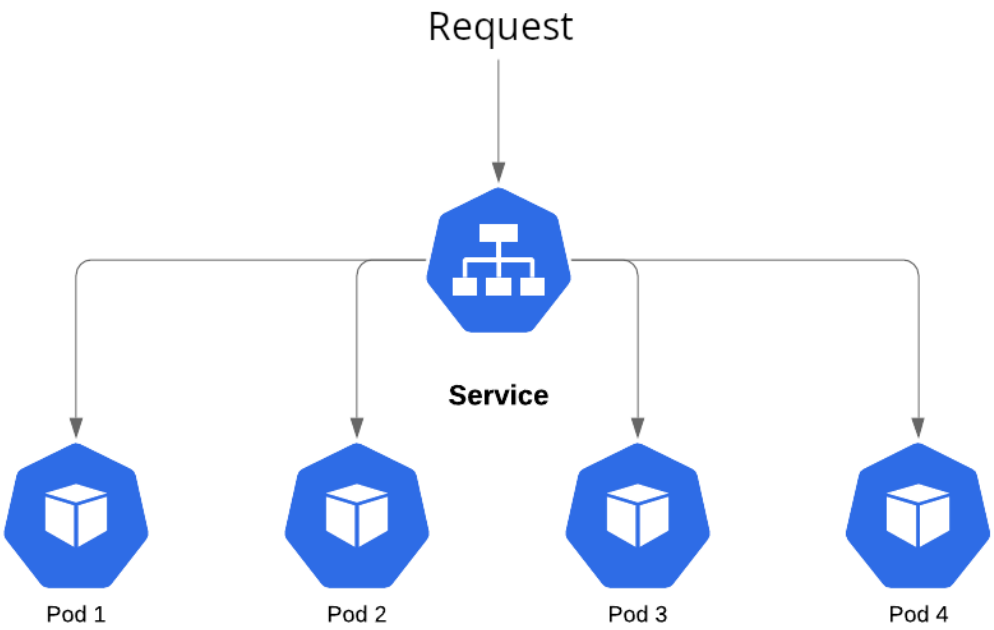


Figure 2.7: Service balancing requests on four replicas of the application.

Since the level and the way the service is exposed can vary, a Service resource can be of different type:

- ClusterIP (default): it exposes the Service on a cluster-internal IP, making the Service only reachable from within the cluster.
- NodePort: exposes the Service on each Node's IP at a static port. It automatically creates a ClusterIP Service, towards which the NodePort Service routes. It is possible to contact the NodePort Service, from outside the cluster, by using <NodeIP>:<NodePort> as address.

- LoadBalancer: exposes the Service externally using a cloud provider's load balancer, automatically creating NodePort and ClusterIP Services towards which it routes.

- ExternalName: maps the Service to the contents of the externalName field (e.g. foo.bar.example.com) by returning a CNAME record with its value, without setting up proxying of any kind.

Ingress Routing

The service types previously described have their strengths but there are also some limitations, especially when the service exposed needs to be accessed from outside the cluster. In order to address these problems, application developers looked for an alternative, creating a new level of abstraction.

The **Ingress** object exposes HTTP and HTTPS routes from outside the cluster to services within the cluster, controlling the traffic routing with rules defined on the Ingress resource. An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. In order to make it work, the name of an Ingress object must be a valid DNS subdomain name, at which it is possible to add a custom name so that it identifies the designated Ingress rules.

Comparing the Service resource to the Ingress one, we distinguish them because the first one is a transport-layer load balancer, while the second one is an application-layer load balancer. Moreover, for this particular reason, the adoption of an Ingress Controller, which also acts as a tunnel termination, is also a more secure way to handle external requests.

But in order to leverage the benefits brought by the Ingress resources, a new component outside of Kubernetes is needed to handle the routing and the tunnel termination: this component is called **Ingress Controller**.

As it is not automatically deployed once the K8s cluster is created, it has to be manually installed in Pods inside the cluster itself. When it comes to acting on requests for ingress resources, the ingress controller addresses inbound requests and produces the necessary routing specifications in line with the specific technology at hand. In common use cases, various ingress controllers are installed within a Kubernetes cluster, where they can be selected and deployed to address each request as appropriate. It is important to underline that different Ingress controllers

support different annotations, that must be indicated inside the YAML file of the Ingress resource.

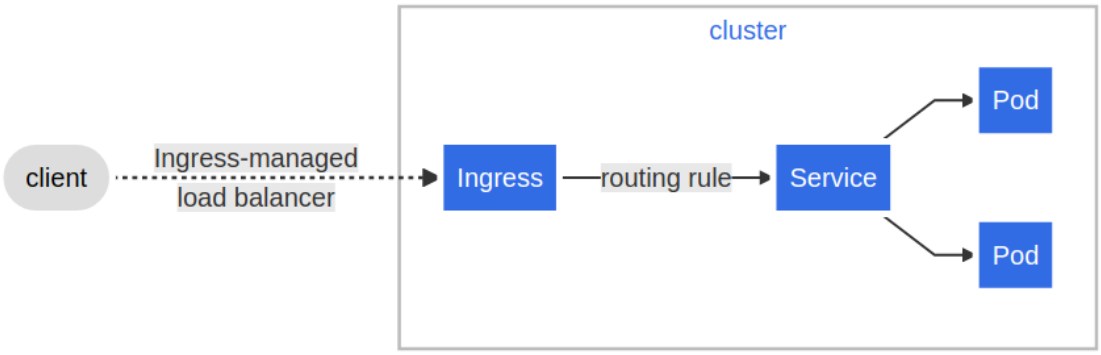


Figure 2.8: Kubernetes Ingress traffic path.

Figure 2.8 shows the path of a request when directed towards an Ingress rule. The Ingress controller receives the request from the client, analyses the address and matching it with its rules finds out the service it has to contact. Then it acts as reverse proxy, so now it makes the same request to the designated service that will forward it to one of that application's pods. Once the controller receives the response, it sends that back to the client.

Chapter 3

State of the Art

In this chapter the current situation regarding the latency problem will be presented, describing why it is so important and reporting what previous researches discovered about this problem by pointing out particular behaviors. Eventually a discussion about the related works is introduced.

3.1 Why Latency?

The evolution of technology during the last few years caused the creation of a lot of new applications in a diverse set of fields, like automotive, gaming, industry process automation and virtual reality, bringing along new and higher requirements. Especially, with the outbreak of Cloud computing, many of these applications started leveraging it, in order to facilitate deployment, management and scaling. But by doing so, even if their requirements obviously do not change, it is clear that the new components introduced could represent a sort of obstacle in the connection between the client and the server, and of course the distance between the endpoints is not negligible. Therefore, even though bandwidth improvements across the backbone and in access networks are important, the monitoring of the latency is a crucial point too.

There is a great interest in technologies that will enable proximal communication links as well as global networks to operate with very low-latency while ensuring high reliability. If we take for example a video call, it is fundamental to keep the end-to-end delay below 200 milliseconds without any jitter (i.e. packet delay variation), because otherwise it would be almost impossible for the people to

interact with each other [9]. Moreover, in case of a safety critical application, it is obvious that an increased latency would cause catastrophic events, like an accident for a vehicle implementing autonomous driving. If we want to create other opportunities to generate revenue, like medical (e.g. tele-surgery) or virtual reality applications, it is fundamental to have the end-to-end delay between the endpoints limited to few milliseconds, even in long-distance networks, so the next step is to find a way to lower and keep this factor under control. Figure 3.1 shows some use-case requirements in the nowadays IT world: in the lower left corner, there are the applications with less requirements, while going up the latency has to decrease and going right more bandwidth is needed. At the top right corner there are the most demanding applications.

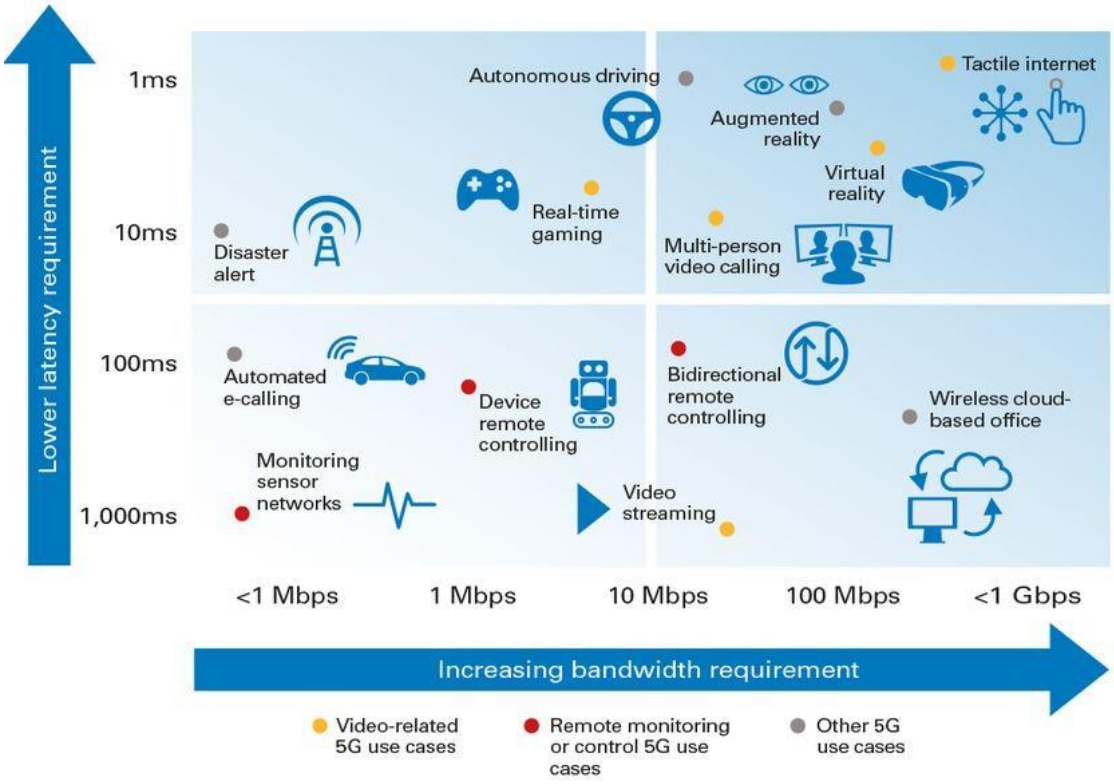


Figure 3.1: Applications bandwidth and latency requirements.

Lately, in order to reduce the propagation delay, companies are moving the computation to the edge implementing the so called *Edge/Fog Computing*, but it is not always enough, because it does not provide a solution to the latency

introduced by the different technologies and the many components involved in the communication.

3.2 TCP Bufferbloat

Almost all applications in nowadays Internet use TCP as transport-layer protocol because of its reliability. As explained before, it uses some mechanisms to guarantee that packets are correctly received by the other peer, detecting a loss event with a timeout or a triple duplicated ACK. But in case of congested network it is possible that this behavior could also cause problems deteriorating even more the effective end to end delay, especially when the buffers of the nodes along the path are particularly big.

This phenomenon is called **Bufferbloat** [10], which causes high latency in packet-switched networks due to excessive buffering of packets. Moreover, it can also produce jitter, as well as reduce the overall network throughput. When a router or switch is configured to use excessively large buffers, even very high-speed networks can become practically unusable for many interactive applications like voice over IP (VoIP), online gaming, and even ordinary web surfing. Some communications equipment manufacturers designed unnecessary large buffers into some of their network products, in which bufferbloat occurs when a network link becomes congested, causing packets to become queued for long periods in these oversized buffers. In a first-in first-out queuing system, overly large buffers result in longer queues and higher latency, and while the aim is to improve network throughput, the result is the opposite behavior, causing a worsening in response time for all TCP flows going through that node. Of course it is not possible to avoid using buffers or use very small ones, because the drawbacks would be even worse with an excessive amount of loss events, therefore is necessary to find a solution. One of the solution proposed requires the implementation of *Active Queue Management (AQM)*, that is the employment of techniques that should signal TCP to slow down when queues grow too much, but unfortunately they are not widely adopted. The Bufferbloat project developed the *CoDel* algorithm to avoid this behavior. It does not need any parameters and controls delays dynamically adapting to link rates, but there are still some misbehaviors and it is not suitable for all network circumstances [11].

Given that this issue affects the most important transport-layer protocol performance, it is one of the most significant problems in nowadays internet, very diffused and still without any definitive and unique solution.

3.3 Related Work

Being the Cloud Native environment continuously evolving and massively adopted only in the last few years, even though its benefits are many, there is not a total comprehension of the overhead introduced by this technology and what are the most useful considerations to keep the latency in the communications low enough to avoid application malfunctions.

During a research regarding coverage, performance and energy consumption of 5G technology, while comparing different TCP congestion control algorithms, it has been discovered a strange behavior, an anomaly on the throughput. In fact, comparing BBR and CUBIC, it has been found out that BBR's congestion window never shrinks during the whole execution, whereas CUBIC's one never reaches a decent throughput due to frequent multiplicative decrease [12]. The paper hints to severe packet losses, but in this thesis it is presented that it is due to the effect of idle connections, that cause the CUBIC congestion control algorithm to go into the slow start phase.

In a paper analysing all protocol stack layers to understand how to reach low-latency networking [1], there is an interesting presentation on how the internet communication world evolved since the circuit switching technology and what are the next revolutions in order to significantly improve the performance. The standard requirement of 200/250ms of end-to-end delay for human interactions is not enough anymore: there are very important job sectors, like industrial automation, medical applications, augmented reality or smart transports, that require very much lower requirements. Even though the overall performance is acceptable in some cases, the objective of the new revolution is to reach under the millisecond delays, that is the key for the most demanding and revenue generating services. The reported factors that affect the end-to-end latency making it unpredictable are:

- Different network operators or entities are involved in the message exchange, therefore by not having a single point of control is hard to predict the end-to-end latency;
- To facilitate link sharing between multiple traffic classes, modern access and

core networks provide differential treatments of packets, and implement quality-of-service (QoS)-dependent packet handling, prioritizing a flow rather than another one;

- The propagation time, obviously, cannot be ignored;
- Every time a packet goes through a node, there is a handling process so that it is redirected towards the right destination, therefore routing time is introduced;
- The traffic load in the path of a packet is the most unpredictable aspect of the latency analysis.

An evaluation of cloud and edge computing showed interesting results. The analysis was performed with the standard ping tool and involved the most important cloud providers (Azure excluded due to the disabled ICMP) used as cloud resources and Akamai servers employed as edge resources instead. Since the providers were located in the same region, the results were very similar for the cloud solution. On the other hand, they discovered that 55%/82% of the users can reach the edge in less than 10/20ms while only about 20%/50% for the cloud. Moreover, it is interesting to notice that edge servers offered lower latency to 92% of the users (even though in some cases cloud latency was lower than the edge one) [2].

A study on the deployment in the edge, showed that end-to-end latency does not improve significantly even when most application services are deployed in the edge location. Using applications designed with frontend, backend and database, they found out that the performance was good only when it was entirely deployed in the edge, but unfortunately it is not feasible for that type of applications. On the other hand, when properly deployed with some services in the edge and others in the cloud the performance was bad [13].

Chapter 4

Measuring the Latency

This chapter presents a complete and detailed description of the tool developed during the thesis work for the analysis of network latency in different designed environment. The focus was to create a simple and easy to deploy program that could extract valuable data and make it available the best way possible for any user. The tool, called **Latency Tester**, is open-source and the code is publicly available for everyone on GitHub [14].

4.1 Goal of the analysis

As anticipated in the previous chapter, during the deployment of an application in the environment set up in the Netgroup data center at Politecnico di Torino, it has been come up against some unexpected and unstable delay in the communication, measuring over 100ms of application latency where the network one is around 10ms. This application was based on a client/server architecture and the two endpoints exchanged images and numerical data through the HTTP protocol. Being the server deployed in a public cloud infrastructure, therefore a very complex one, and since the behavior was very unusual, this thesis work wants to understand if the issue was due to the environment around and investigate what are the other latency-related problems that this particular technology could involve.

To provide other examples, there are some real use-case scenarios that require an end-to-end delay limited to about few milliseconds, like entertainment platforms, healthcare technology, transport services or industrial processes, so that they are able to provide high availability and their potential can be properly exploited.

Moreover, task offloading is a rapidly growing trend, moving the deployment of the services from an access device to edge computing resources, therefore forcing the applications to maintain a certain connection stability. The overall latency-wise performance is a key factor for the provider to earn high revenues, so it cannot be overlooked.

In order to achieve a high level of understanding about the delay in the communication, it was crucial for the analysis to have the total control of all the element actively participating in the data gathering. But most importantly, it was necessary to develop a tool so that it was well known how it worked and that there was the possibility to choose which aspect of the environment to control and track throughout the execution.

The best way to collect a diverse set of data is to gather information in different situations, for repeated times, in different moments of the day. For this particular reason, it was required to make this tool as dynamic as possible, offering to the user a simple way to configure parameters and select how and when it should execute its jobs.

Last but not least, the deployment of the tool must be as smooth as possible, without needing so much knowledge about technical aspects or requiring to spend so much time configuring the environment for it to properly work.

4.2 Architecture

The idea behind the **latency-tester** developed is to create a **Client/Server** application that simulates a real one in a deployment scenario. The server implementation is quite basic, whereas the main working logic resides in the client. The basic principle which it is designed on is to create an application-layer ping, so that is possible to understand and justify the end-to-end round-trip time between processes and not just hosts. In order to do that, it has been used *Websocket* as communication channel, so that it is not necessary to include in the measurement the connection establishment overhead for each message like in a REST scenario, affecting the effective time between the send of the ping and the reception of the corresponding pong message. Moreover in order to observe the behavior of the network environment, the tool also retrieves data about layer-4 and layer-3 latency, by using *tshark* and *ping* commands to respectively measure the TCP ACKs round-trip time and network packets round-trip time between the two hosts.

Finally, to generate the most diverse set of result, it is based on a set of input parameters that make it dynamic and configurable.

While choosing the programming language to design the tool, the ease of use and efficiency were the main characteristics to look for, since dealing with latency measurements it is important to avoid any other delay outside the network one. Nowadays, *GoLang* is an open source programming language that makes it easy to build simple, reliable, and efficient software and it is the most in-demand language [15]. Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. It compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language. Based on the syntax of C, it comes with some changes and improvements to safely manage memory usage, manage objects, and provide static or strict typing along with concurrency. Google started developing this language to increase productivity, code efficiency and performance for business, aspects that are making it the best choice for development.

4.2.1 Latency-Tester Components

The developed tool presents four total components, having one of them that wraps other two:

- **Server**: simple pong server;
- **Enhanced Client**: automated script that pings the server in a configured way, leveraging other two components:
  - **Client**: server pinger;
  - **Plotter**: generator of the result plots;

Given this simple explanation of the role these components cover, each one of them is thoroughly described along with its structure.

Server

The server is the simplest component of the measurement tool. It is just a web server that is constantly listening on an address and waiting for a connection from the

client to come. Once the client requests for a connection, it sends as first message the size of the payload that the server should put inside the response message in the payload field. This feature is required because in a typical task-offloading scenario, the communication between the two endpoints is usually asymmetric, since the client is the one who generates a bigger amount of data while the server responds with a small message that is the result of the computation requested (e.g. a smart car with autonomous driving sends a lot of information gathered from its sensors to the server just to receive as response that it should steer a bit to the left). By default it works without TLS, but it is possible to enable it with a flag and, of course, certificate and private key are needed.

Once established the connection and set the response payload size, the tool is ready to receive the test messages from the client. The messages are received at a certain rate and the server has to do just three simple jobs: replace the client payload with its payload, add the server timestamp and send the message back to the client. It is like an application-layer pong server. The addition of the server timestamp is an information that client side can be post-processed to understand if there is any marked server jitter compared to the client send rate.

Client

Whereas the server has a simple job, the client has a much more complex structure, since it is in charge to handle all workloads and computation. As it is executed, it establishes the connection with the server, exchange the necessary data to set up the following test and is ready to go. It is a very customizable component, since it is possible to provide many interesting input parameters to handle the communication and the testing in a custom way: based on the server which it connects to, it can use TLS or not; it can send messages for a determined number of times or until it is stopped; it is possible to define request and response payload sizes; each message can be sent with a provided interval between one another; in particular cases you could require to specify a preferred source port for the client; the output data can be stored in a file with a custom name. Moreover, in case the user would like to track the pathway between the two endpoints, it is possible to indicate an IP address (that could be the same server host or any other that is as near as possible) to track which nodes the messages run across. Furthermore, by setting a flag, the tool will also retrieve all the client TCP socket statistics and save them to a file, so that if there is a strange behavior in the application-layer delay,

it is possible to compare it to this information and discover valuable observations.

Once the connection has been set, the client generates a parallel thread that is in charge of handling the incoming packets from the server, while the main thread is the one managing their sending. When creating the message to send to the server, the client adds its payload and the current timestamp, so that when it comes back, it is possible to compute the application-layer end-to-end delay. As for the reading thread, for each packet received dispatches a new thread to handle the delay computation and then store the result in the log file.

Since this component could run for an extended period of time using a single connection, it could happen that some very fine-grained programmed firewalls kill a long-lasting connection, therefore ruining the test that was executing. For this particular reason, the client is coded to be resilient and able to resist to a connection kill, by re-establishing a new one without the intervention of the user and picking up where it was interrupted.

But even if this client has many dynamic aspects that could make it so valuable, something more scalable was needed, in order to gather a more diverse set of data and generate higher quality results. The Enhanced Client is the answer to this necessity.

Enhanced Client

As described before, the client is very effective, but it does not allow to schedule it to run in an automated way with different set of input parameters, so it would require a lot of interaction with the user resulting in a huge waste of time and a high risk of human error.

The idea behind the enhanced client described in the next lines, is to wrap the simple client presented above inside an executable, acting like a sort of "scheduler", that leverages many input parameters to describe in details how the process of data gathering should proceed. Between all the setting involved, the main ones are the *endpoints*, the *send intervals* and the *request message sizes*, since by the early tests they came out as the most interesting variable to combine and analyse. Moreover, since the execution environment is changed and is more expanded, some very helpful features have been added too, in order to have a complete overview of the analysed scenario.

By just looking at the application-layer latency, the risk of misinterpreting some behaviors is high, therefore it has been decided to deepen the analysis by retrieving

data concerning also transport-layer and network-layer protocols. In order to do so, the *tshark* [16] tool is in charge of retrieving layer-4 information while *ping* is used to measure layer-3 round-trip time. In addition, since it is important to compare the test result with the effective capacity of the network between the two endpoints, *iperf* [17] is used by the tool to understand the bandwidth available. The only cumbersome part behind this is the fact that the user must manually deploy the iperf server in the same host of the tool server, or any machine as near as possible, otherwise the measure cannot be executed.

Beside the client, the enhanced client puts in place a new component, the plotter. This element is very useful and simplifies the process of data analysis without forcing the user to create its own script to read data and visualise it in a user friendly way.

Plotter

The execution of the enhanced client, generates a lot of useful data. But the real problem about this huge amount of information is how to read it. For a user that uses a measurement tool, data visualisation is the most important aspect he is looking for, otherwise it would be very difficult to understand what really happens by looking at raw numbers.

The job of the plotter is to use all the files generated by the enhanced client and create useful graphs that are immediately available for consultation for the user. It provides different representations, so that all the peculiarities of the test can be highlighted:

- End-to-End latency: it is the description of the behavior of the application-layer round-trip time throughout the entire execution, with the values represented and linked with a line. This graph puts together all the results regarding every combination of the main parameters and plots each one of them in a different page of the same file, so that the user is able to track the delay and notice if anything particular happened at a certain time. For example, this plot is very useful when dealing with 24 hours executions, so that it is possible to see if along an entire day there is a particular moment of stress. Figure 4.1a shows an example.

- Transport layer latency: it is a line representing the round-trip time between the send of a packet and the reception of its ACK, in order to roughly

understand which is the transport-layer latency. Moreover, since there could be congestion or any problem that would ruin the test if not graphically reported, the packet retransmissions are represented with vertical lines at the respective timestamp, because if the application-layer latency is considerable and there are problems in a node along the network, the problem can be quickly discovered.

- Network latency: it is a line representing the network-layer round-trip time, gathered as raw data with the ping command.

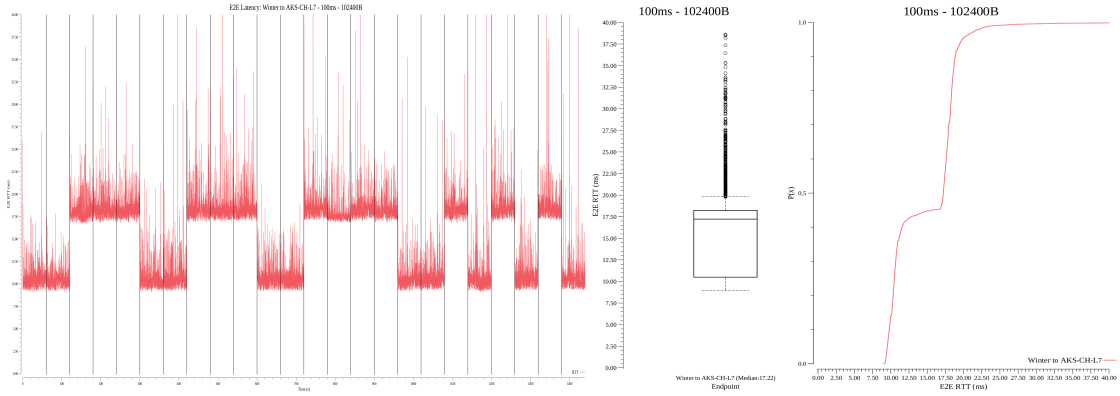
- End-to-End latency BoxPlot: it is a standardized way of displaying the distribution of data based on a five number summary, that are minimum, first quartile (Q1), median, third quartile (Q3), and maximum. It provides a good summary of the results, but the value-timestamp association gets lost and so the behavior throughout the execution time. The component generates three version of this type of graph, with the intent to present the point of view from each one of the main parameters by combining the other two. Figure 4.1b shows an example.

- Per run End-to-End latency BoxPlot: it is a boxplot representation of the latency of all the runs executed for each combination in order to understand in a better way, with a distribution based graph, if there is a different behavior depending on the time of execution.

- End-to-End Latency Cumulative Distribution Function (CDF): based on the gathered data, it shows the probability that the variable will take a value less than or equal to x. This graph is able to show in a clear way the exact percentage of values that are under a certain RTT limit and how much the results vary. Again, the value-timestamp association gets lost and three versions of it are provided with different points of view. Figure 4.1c shows an example.

Furthermore, since it would be very bothersome to look at each graph to understand quickly how the test resulted, the plotter generates a simple summary that for each combination reports the average round-trip time and its standard deviation, so that the user can have an hint on what happened during the execution.

The only data that is inconvenient to plot is the iperf result. The user can check the values of the measurement by looking at the output of the tool in the



(a) E2E/Transport/Network latency **(b)** BoxPlot **(c)** CDF

Figure 4.1: Plotter output examples.

designated file. This information can be very useful, especially when the test result is worse than expected due to some configuration or hardware problems.

Given that the plotting of the results may show data in an unclear way, there is the possibility to modify some parameters of the plotter and manipulate the raw data in order to have a more understandable graphical representation, especially making different situations comparable with one another so that the user is able to deduce valuable information and lessons learned.

4.2.2 Deployment

Deploying an application in an environment usually requires many features and settings that are sometimes difficult to respect. In addition, given that it is dealing with a compiled language, each user would have to download the source code and build it on his machine before executing the program. To ease the deployment process, the containerization is the most helpful choice, since it would mean that to run both client and server, the only necessary thing is a container execution environment. Moreover, since the main focus of our analysis is a cloud native use case, remember that a cloud infrastructure executes all of the applications in separated containers to provide an isolated environment without an excessive virtualization overhead and management.

Therefore, each component of the tool has been deployed using *Docker* [18], since 2013 industry standard for containers. It simplifies and accelerates the workflow, while giving developers the freedom to innovate with their choice of tools,

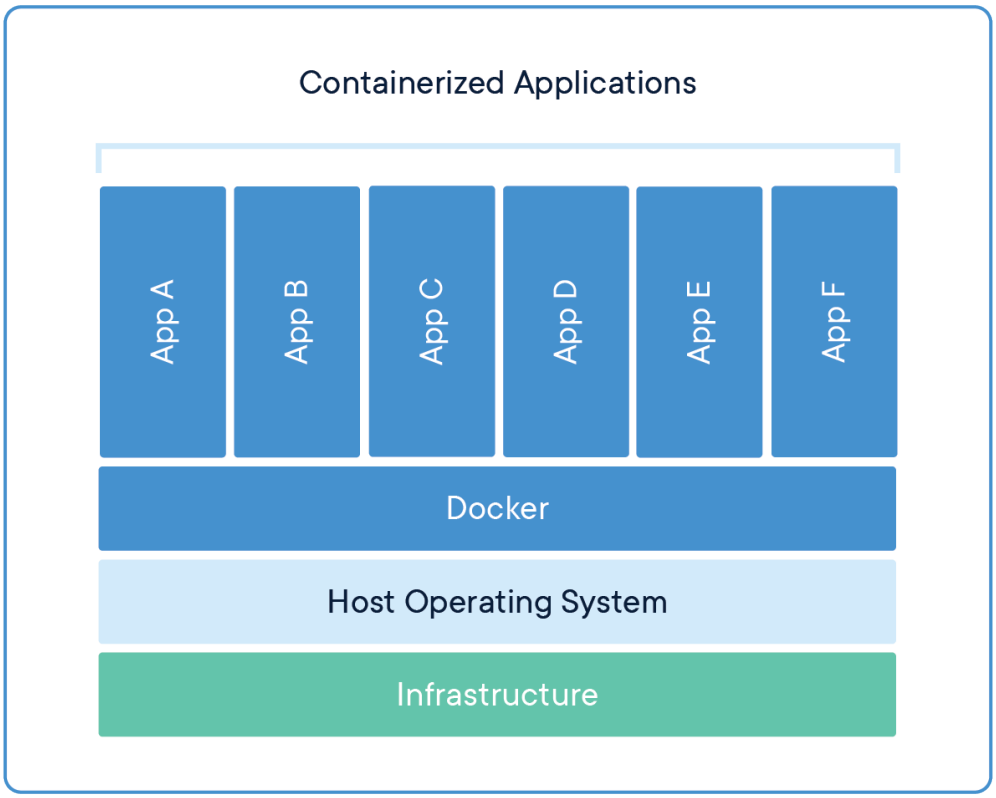


Figure 4.2: Docker containerization.

application stacks, and deployment environments for each project. Docker is an open platform for developing, shipping, and running applications. It permits to separate applications from the infrastructure so that it is possible to deliver software quickly. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, the delay between writing code and running it in production could be significantly reduced. Docker uses a client-server architecture where the Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing the desired Docker containers. The Docker client and daemon can run on the same system, or it is possible to connect a Docker client to a remote Docker daemon. These two components communicate using a REST API, over UNIX sockets or a network interface. The images run in docker are stored in a registry, that can be the public one, Docker Hub, or a private one, that should be manually configured. Notable trait, as many other new emerging technologies, Docker is written in Go. Figure 4.2 shows how each application is containerized with the help of the docker engine running over the host operating system.

4.3 Working Logic

The tool main job is to measure the end-to-end communication latency by exchanging messages like the ping command, but keeping in consideration all the layers up to the application one. Since nowadays services leverage the benefits brought by REST APIs, the main protocol used is HTTP, and the payload is typically encoded as **JSON**. This is an open standard file format and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types. For this reasons it is the message format used also by the tool to exchange the message where timestamps and payload are stored.

However, an HTTP communication relies on TCP, which, as presented in the previous chapters, has a relatively long process of connection establishment that would take so much time. Moreover, in case we take into account a secure channel using HTTPS, the TLS parameters exchange would require even more time, remarkably influencing the communication latency.

Therefore, in order to reduce the impact of these procedures and focus the analysis on the message exchange, a persistent communication channel was necessary. The technology employed to solve this problem is **WebSocket**, a computer communications protocol, providing full-duplex communication channels over a single TCP connection. WebSocket is compatible with but distinct from HTTP, even though both protocols are located at layer 7 in the OSI model and depend on TCP at layer 4. To achieve compatibility between them, the WebSocket handshake uses the HTTP Upgrade header to change from the HTTP protocol to the WebSocket protocol. Once the client and server have both sent their handshakes, and if the handshake was successful, then the data transfer part starts. This is a two-way communication channel where each side can, independently from the other, send data at will. In our case, of course, the server only responds to request messages from the client. The library used in this tool is the one provided by the gorilla web toolkit and obviously it is written in Go [19].

When executing the tests, the main component employed that sets the pace and the rules is the enhanced client, that communicates with the server using the client and then generates graphs exploiting the plotter. In order to have a clear idea of how this element works, by looking at Figure 4.3 it is possible to see the flow graph of the enhanced client as employed in this thesis tests.

While designing a test execution, the first thing to do is to deploy the server

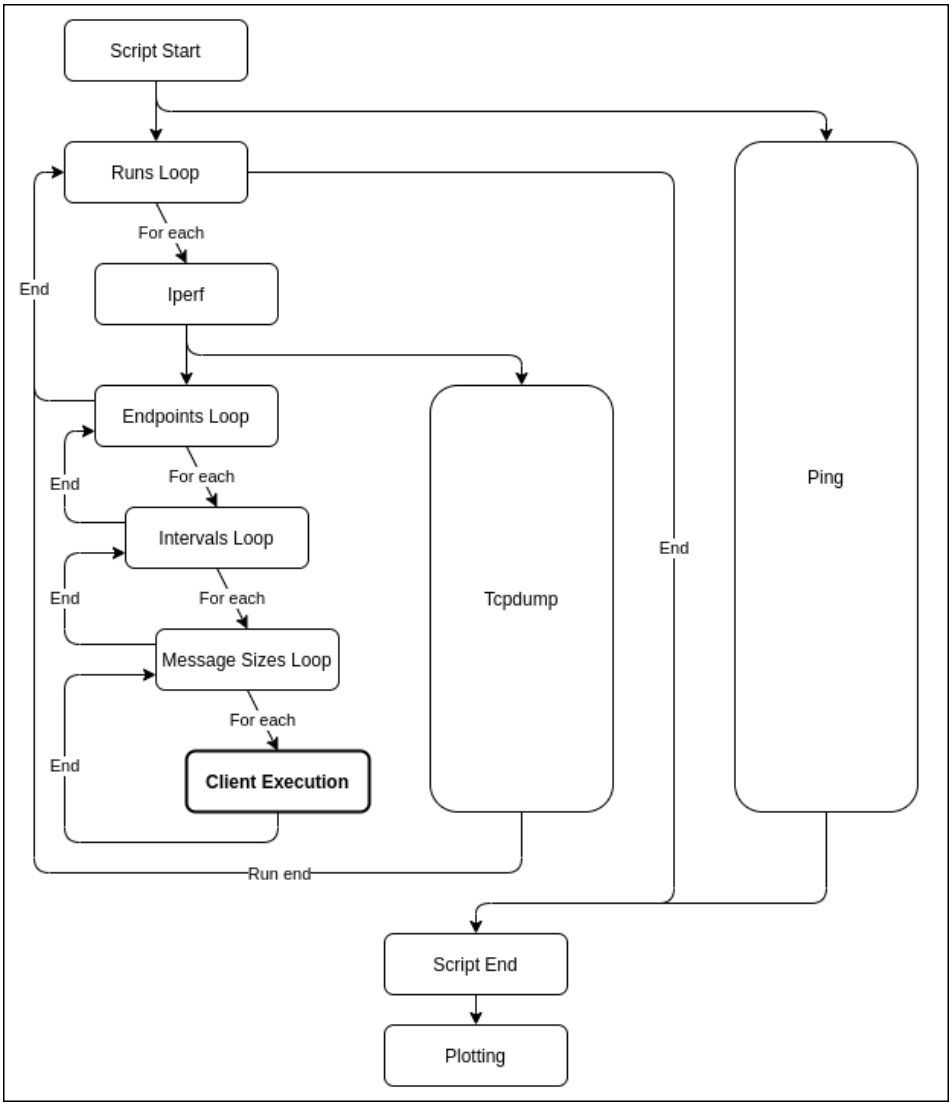


Figure 4.3: Enhanced Client flow graph.

in the hosts and clusters where we want to run our client towards. As explained in the next section, then the user needs to create a configuration file so that the enhanced clients knows how to execute depending on the reported parameters.

Indicating one or more ping destinations, the enhanced client will create a parallel thread for each one at the start of the execution and they run the ping command towards the specified IP until the end of the plotter execution with a custom frequency.

On the other hand, the main thread is responsible to execute a number of runs,

that is the number of iterations the user wants to run its test so that he can gather as much diverse data as possible. Each run is separated by a custom interval, that is the time span between the start of each run. If any iperf destination reported and, of course, a proper iperf server deployed on the server machine, at the start of each run the tool probes the bandwidth between the two endpoints to get an overview of the communication capacity. Then, if requested, another parallel thread is created to run tshark so that it gathers the data of the packets we are interested in so that after the execution the user is able to track the behavior of the transport-layer latency.

Whereas the focal point of the enhanced client is the combination of three set of parameters that the client specified in the settings file: endpoints, send intervals and request message sizes. These three elements are combined in the loop to run the client for a determined time lapse, therefore gathering the application-layer latency between the client and the server for each send interval and request message size.

At the end of all the combinations, if there are still runs remaining to be executed, the enhanced client waits for a certain time span until the start of the next run accordingly to the run interval specified in the configuration file.

When all runs are completed, the plotter uses all the output files of the enhanced client and generates the graphs making them available for the user in a dedicated directory. The generated plots are easy to consult, but it sometimes happens that there are strange behaviors in some of them that it could be useful to highlight, as well as some data may need to be cut out so that the general appearance of the test is at the center of the image. In fact, there could be a very small percentage of values, that are the outliers of the analysis, which stretch the plot so hard that it is impossible to see the actual line representing the result, making the image useless. By handling the plotter settings and re-executing this component again and again until the desired outcome, it is possible to manage the result and extract valuable data representations for the user.

4.4 Configuration

The deployment of the components creating the testing environments implies attention by the user, because it is necessary to configure each one of them so that client and server can correctly communicate as desired.

Server

The first element to be deployed is the server. It needs to be accessible by the client, so if the two component are not in the same LAN, it must be exposed to the outside network. It can be done by a simple port mapping or, if we have a Kubernetes cluster available, it can be contacted through a Service or an Ingress resource. In both cases, executing the server with the default settings will make it listen on all host IP addresses through port 8080, otherwise it is possible to modify this behavior by setting the appropriate flag with the desired address. Moreover, again depending on the relative flag, it can run with TLS or not. Beware that in case it is set to true, key and certificate are required. Table 4.1 shows the list of all the server flags with their respective default value.

Flag	Description	Default Value
-addr	Listening address and port	0.0.0.0:8080
-tls	true if TLS is requested	false

Table 4.1: Server flag parameters

Client

Once all the desired servers are up and running, the counterpart must be configured. The other component used in this thesis is the enhanced client, but as explained before, it leverages the simple client wrapped in it in order to retrieve the desired data. Therefore, by using the parameters defined in the settings file, for each measurement the client is called with different parameters, gathering much information without any other user interaction. The client is set to run indefinitely by default towards the specified address, otherwise a defined number of times if the repetition flag is set. It can be defined the request and the response payload size, indicating the number of bytes. Each message is sent after a defined interval from the previous one and, in case the server is running with TLS activated, also here it must be set to true. By default the result file is simply called log, but it can be renamed with the respective flag. Other flags, not used by the enhanced client, are the traceroute, running the command towards the specified IP address before the standard execution of the client, and the tcpStats, that allows to retrieve all the values of the socket structure inside the kernel. Beware of this last flag: it

keeps polling the kernel without any wait time consuming a lot of CPU and RAM, therefore it is suggested to run the client just for a short time span. Table 4.2 summarize client's flags.

Flag	Description	Default Value
-reps	Repetitions, if 0 it runs until interrupt	0
-requestPayload	Request payload size (bytes)	64
-responsePayload	Response payload size (bytes)	64
-interval	Requests send interval (milliseconds)	1000
-tcpStats	true if TCP Stats are requested	false
-tls	true if TLS is requested	false
-traceroute	if present, address to traceroute	
-log	name of the results file	log

Table 4.2: Client flag parameters

Enhanced Client

As explained before, in this thesis the client is managed by the enhanced client, that is configured with a YAML file, a human-readable data-serialization language, that is parsed at the start of the execution. Here is an example of a configuration file for the enhanced client:

```yaml
# Enhanced Client Settings

# How many times to iterate
runs: 24
# Time interval between the start of two different runs (in minutes)
runs_interval: 60
# How much time each client execution should last (in seconds)
runs_step_duration: 30
# Iperf Destinations
iperf_destinations:
- name: "Deployed-Iperf3-Server"
  ip: "34.56.78.90"
  port: "80"
# Ping Destinations
ping_destinations:
- name: "Pingable-Host"
  ip: "23.45.67.89"
# Interval between ping executions (in seconds)
ping_interval: 30
```

```
# Source port for the client socket (default is random)
source_port: 5555
# List of endpoints to test E2E latency, defined by its name/description and its
    address
endpoints:
- description: "1_Example-Address"
  destination: "12.34.56.67:8080"
- description: "2_Example-Hostname"
  destination: "latency-tester.example.com"
# List of intervals between the send of two messages to test E2E latency
intervals:
- 25
- 100
- 400
# List of request message sizes to test E2E latency
msg_sizes:
- 10240
- 102400
# Response message size
response_size: 1024
# True if TCP ACK RTT is requested
tcpdump_enabled: true
# True if TLS enabled
tls_enabled: true
# Execution directory (if in Docker, this must coincide with the mapped directory)
exec_dir: "/execdir/"
```

The most important fields of the file are:

- **endpoints**: the list of addresses of the server deployments. The tool is able to handle both IP + port and hostname.
- **intervals**: the list of the time intervals between the send of two messages by the client, expressed in milliseconds. Beware to keep this number sufficiently high so that the tool can keep the pace.
- **msg_sizes**: the list of the payload sizes of the requests towards the server. Again, keep this number adequate and suppose a legit dimension for a real-case application.

These three elements are combined together and are the only parameters that change during the execution of the enhanced client. Each combination of these three fields is a single *step*, while the execution of all the combinations is a single *run*. An important factor to keep in mind, is to choose the parameters with a legitimate logic behind: keep in mind the distance between the client and the endpoints and the bandwidth bottleneck when choosing intervals and sizes, because it is crucial that the application delay is not obstructed by physical limits.

The run number specifies how many times all the combinations are executed, with a wait time specified in the runs_interval field. Each combination is executed for the runs_step_duration interval, that calculates how many repetitions the client should make depending on the send interval selected. This means that the gathered values for a short send interval are more than a higher one. One of these two fields can be omitted and inferenced by the other, but at least one must be specified. The response_size, the tls_enabled and the source_port values are fixed for all the combinations. This last one is optional, used only in special cases where it could be useful to use always the same port if it brings benefits in terms of latency (as it will be explained in the next chapter). The ping is executed only if any element in the list is present. The same works for iperf, but of course an iperf server must deployed in the same machine or any other near the server component of the tool. The tcpdump capture is optional, but very recommended since it could produce very usefeul results.

In the example above, there are 2 (endpoints) x 3 (intervals) x 2 (sizes) = 12 combinations. Each steps lasts 30 seconds, therefore the duration of all the combination in a single run is 6 minutes. Between the start of a run and the next one, there are 60 minutes, then the complete duration of all the 24 runs is around 23 hours and 6 minutes (after the last run, there is no waiting time).

Plotter

At the end of the execution of the enhanced client comes the plotting, done by the component wrapped inside it. Usually the plotting is done with default settings, since it is hard to predict how a test will result. With them, the plotter produces plots automatically choosing the parameters, like the minimum/maximum RTT value or the percentiles to remove, so the user will need to do some tuning to extract the maximum value from the data. Therefore, by executing the component itself, it is possible to regenerate the graphs by only adding few settings to the configuration file of the enhanced client without needing to re-run the entire test. Those are parsed by the plotter and used to handle the data and present it in a better way. Here is an example of how these fields look like:

```
1 # Enhanced Client Settings
2
3 ...
4 (The settings used to run the enhanced client)
```

```
5  ...
6
7  # Plotting Settings
8
9  # How many percentiles to remove from the extremities of the plots (default 0 if
       omitted)
10 percentiles_to_remove: 1
11 # If false, BoxPlot and CDF have the same scale
12 equalization_disabled: false
13 # The minimum value in the plot for the RTT (automatically obtained if omitted or equal
       to 0)
14 rtt_min: 10
15 # The maximum value in the plot for the RTT (automatically obtained if omitted or equal
       to 0)
16 rtt_max: 20
17 # The boxplot min whisker percentile (default 1.5x interquartile distance if omitted)
18 whisker_min: 10
19 # The boxplot max whisker percentile (default 1.5x interquartile distance if omitted)
20 whisker_max: 90
21 # The runs we selectively want to plot (default all runs if omitted)
22 runs_to_plot:
23 - 1
24 - 3
```

The percentiles_to_remove field specifies how many percentiles to remove from the upper and lower sides of the latency set of values, showing only the central part. It is useful because in the last or few lasts percentiles there are the outliers, which may stretch the image of the graph making it unreadable and useless. The equalization is convenient for the group of boxplots and CDFs graphs, because if enabled it allows to see them with the same scale and make the interpretation very quick and easy for the user. The minimum and maximum rtt values act like the first field, but they rely on a fixed setting instead of a percentage. Whiskers min and max are the custom values of the boxplot whiskers based on percentages, including in the valuable data set more outliers than the default. If by looking at the per run boxplot is possible to notice particular behavior for a certain run or for a bunch of them, it is possible to specify those in the runs_to_plot list so that the plotter represents on the graphs only the data retrieved by these runs.

4.5 Metrics

The purpose of the tool is to measure the overall performance of the communication between the client and the server of an application. Therefore, to have a complete overview of the environment where the two components run, the tool is able to

collect a diverse set of data at the same moment, making available for the user few metrics to analyse:

- *End-to-End latency*: this is the most important aspect of all, because it represents the effective latency perceived by the application user. For this reason, it is the one that needs to be thoroughly observed and analysed, with the aim to preserve a high quality user experience. Here is an example of the raw output of the enhanced client containing the timestamp of when the message is sent, the timestamp of when it is handled by the server and its round-trip time:

```
#client-send-timestamp,server-timestamp,e2e-rtt
1611336441708429104,1611336441732325106,43.745733
1611336441958579188,1611336441982435400,41.819516
1611336442208693801,1611336442234417528,44.20757
1611336442459715445,1611336442483520866,41.904871
1611336442709978344,1611336442733883971,40.867048

```

- *Transport latency*: this is the metric captured by tshark, analysing outgoing and incoming packets. It is useful because it registers the round trip time between the send of a segment and the reception of its ACK, giving another point of view in terms of latency by using a more fine-grained approach (one application message can be the aggregation of more segments). Moreover, it tracks also retransmissions, indication if there are any physical problems or the network is particularly congested. Here is an example of the raw output of tshark containing the timestamp of the frame, the interval between the send of the packet and the reception of the corresponding ACK, the stream id of the TCP communication, if the packet is a retransmission and the destination IP of the packet:

```
#frame-timestamp,tcp-ack-rtt,tcp-stream-id,retransmission,ip-dst
"1611336442.750143006","0.039583780","0",,"172.17.0.2"
"1611336443.002436800","0.041769282","0","1","172.17.0.2"
"1611336443.251176239","0.040232318","0",,"172.17.0.2"
"1611336443.503958671","0.043235419","1",,"172.17.0.2"
"1611336443.751584653","0.040869367","1","1","172.17.0.2"

```

- *Network latency*: this is the standard network latency measured by the ping command, useful in certain cases if there are particular behaviors that are due to the network and not to the application or the transport protocol. It is the last line of control and it is supposed to always give good results. Here is an example of the raw output of the ping:

```
PING 130.192.31.254 (130.192.31.254) 56(84) bytes of data.
[1611336429.383812] 64 bytes from 130.192.31.254: icmp_seq=1 ttl=242 time
=71.8 ms
[1611336430.358613] 64 bytes from 130.192.31.254: icmp_seq=2 ttl=242 time
=45.1 ms
[1611336431.355965] 64 bytes from 130.192.31.254: icmp_seq=3 ttl=242 time
=42.0 ms
[1611336432.360581] 64 bytes from 130.192.31.254: icmp_seq=4 ttl=242 time
=45.6 ms
[1611336433.361396] 64 bytes from 130.192.31.254: icmp_seq=5 ttl=242 time
=45.0 ms
...
[1611336479.412320] 64 bytes from 130.192.31.254: icmp_seq=51 ttl=242 time
=36.5 ms
[1611336480.412441] 64 bytes from 130.192.31.254: icmp_seq=52 ttl=242 time
=35.3 ms
[1611336481.415124] 64 bytes from 130.192.31.254: icmp_seq=53 ttl=242 time
=37.2 ms
[1611336482.415148] 64 bytes from 130.192.31.254: icmp_seq=54 ttl=242 time
=35.8 ms
[1611336483.414928] 64 bytes from 130.192.31.254: icmp_seq=55 ttl=242 time
=34.5 ms

--- 130.192.31.254 ping statistics ---
55 packets transmitted, 55 received, 0% packet loss, time 54068ms
rtt min/avg/max/mdev = 34.413/52.714/261.110/48.222 ms

```

- *Bandwidth available*: it is as much important as the end-to-end latency, since it is the amount of data that can be exchanged between the client and the server, and it is useful to understand if the results collected are conditioned by a physical limit or are just the representation of the network behavior. Here is an example of the raw output of the iperf:

```
Connecting to host 130.192.31.240, port 80
[  5] local 172.17.0.2 port 39566 connected to 130.192.31.240 port 80
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   464 KBytes  3.80 Mbits/sec    0   55.6 KBytes
[  5]   1.00-2.00   sec   458 KBytes  3.75 Mbits/sec    0   73.2 KBytes
```

```
6     [  5]   2.00-3.00   sec   264 KBytes  2.16 Mbits/sec    0   90.8 KBytes
7     [  5]   3.00-4.00   sec   499 KBytes  4.09 Mbits/sec    0    110 KBytes
8     [  5]   4.00-5.00   sec   312 KBytes  2.55 Mbits/sec    0    127 KBytes
9     - - - - - - - - - - - - - - - - - - - - - - - - -
10    [ ID] Interval           Transfer     Bitrate         Retr
11    [  5]   0.00-5.00   sec  1.95 MBytes  3.27 Mbits/sec    0             sender
12    [  5]   0.00-5.35   sec  1.82 MBytes  2.85 Mbits/sec
      receiver
13
```

Using these raw data files generated by the enhanced client, the plotter will produce the graphs previously described, but they will be showed in the next chapter during the analysis of the data obtained during the investigation we conducted leveraging this measurement tool.

Chapter 5

Evaluating Results

This chapter is focused on the analysis that has been conducted with the help of the latency-tester on a set of designed case-scenarios. There have been prepared few environment setups, taking into account different configuration, executing the latency-tester in the most diverse way so that the data gathered could be as much useful as possible. With the information collected, it was possible to give explanation to particular behaviors that have been observed.

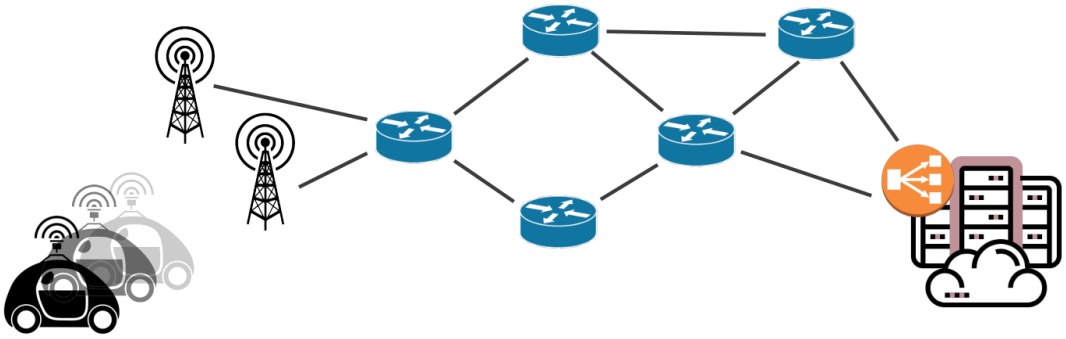


Figure 5.1: An automotive real-case scenario.

As explained in the previous chapter, the tool has the role to act as a client/server application deployed in a cloud-computing scenario (like an automotive one represented in Figure 5.1), simulating a periodic communication between the two endpoints. Talking about the main parameters used in the tests, besides the different endpoints that will be presented for each scenario, the request payload set used is the following:

- **1KB request and 1KB response** simulating *command & control*, a small request like a command line interface tool replied with a simple response;
- **10KB request and 1KB response** simulating *telemetry*, a bigger set of data is sent towards the server, which replies with a smaller response;
- **100KB request and 1KB response** simulating *image recognition*, with the client sending frames to the server that will send back little info about it.

In order to test the communication both in a situation of stress and with breathing space, the send intervals used are 25, 100 and 400 milliseconds. Nowadays application certainly communicates with TLS, therefore the tests are conducted leveraging secure connections. Each section of this chapter represents an environment configuration: all of them are described in detail and then the most peculiar behaviors discovered are presented, with the help of the graphs given as output by the latency-tester. At the end of every case, the lesson learned is presented.

Figure 5.2: Locations of the resources involved in the analysis

In order to clarify what were the resources used for this thesis analysis, here they are listed and Figure 5.2 shows where they are located:

- *Winter* server, on-premise host located in a laboratory data center in the DAUIN department of Politecnico di Torino;

- *CrownLabs* cluster, an on-premise Kubernetes service situated in the LADISPE laboratory at Politecnico di Torino;
- *AKS Clusters*, three Kubernetes clusters deployed in the Zurich, London and California regions of Microsoft Azure public data centers;
- *AWS virtual machines*, three virtual machines deployed in the Paris, London and Oregon regions of Amazon public data centers;
- *Home networks*, one in the province of Turin and the other in the province of Rimini.

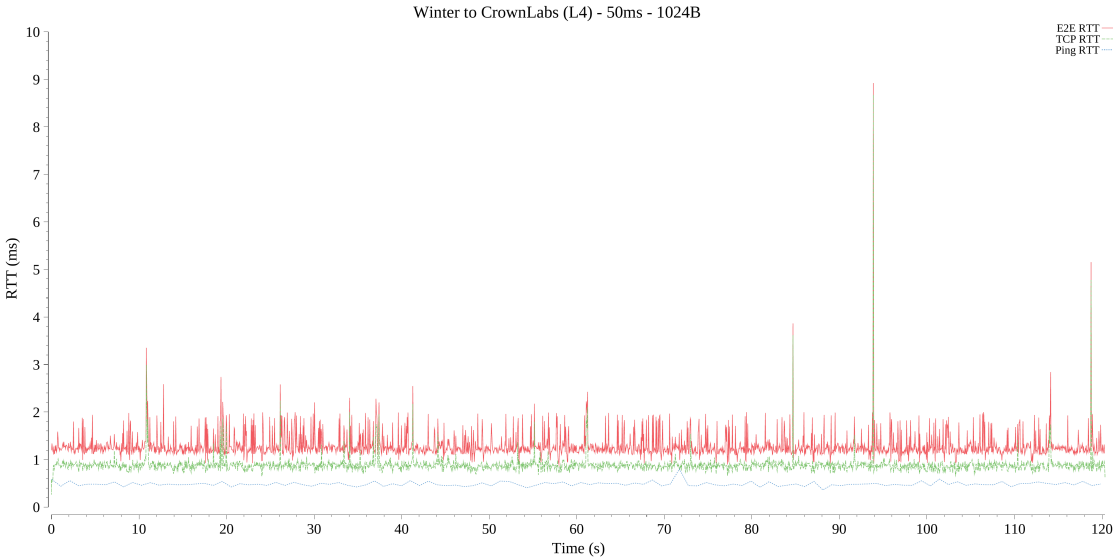


Figure 5.3: Comparison between Network vs TCP vs Application RTTs

Even if more data is gathered and taken into account, the three main metrics considered and compared in Figure 5.3 are:

- **Network level** RTT: the bare performance of the underlying network. Warning: ping measurements are associated with some caveats, as ICMP is often blocked/limited (e.g. by corporate networks and cloud providers), as well as it may be less privileged than traditional TCP/UDP traffic, leading to inconsistent results;
- **Transport level** RTT: the performance of the TCP stream, considering the time required to send one segment and receive the corresponding acknowledgment. It depends on the propagation time, as well as on buffering and possible

processing by middle-boxes. However, it may not represent an E2E performance indicator in case a reverse proxy is present. The difference between the 1KB and the 100KB scenario can be attributed to delay ACK mechanisms;

- **Application level** RTT: the performance perceived by the actual application, including the entire propagation, transmission and processing times.

In the figure above, it is possible to notice that in terms of latency the order respects the layer of each metric, that is the network RTT is the lowest and the application one the highest. Increasing the message size to a dimension bigger than the MSS, it is divided in more than one TCP segment, therefore increasing the E2E delay. On the other hand, the TCP and Ping RTT increase only in case of L4 and L3 buffering respectively. Since E2E and TCP RTT are measured on the same packets, the first is always higher than the second, whereas the ping can oscillate in any way.

5.1 On Premise to Cloud

The first scenario analysed is the one where the client is deployed in a server in an on-premise cluster whereas the servers in public clusters around the world.

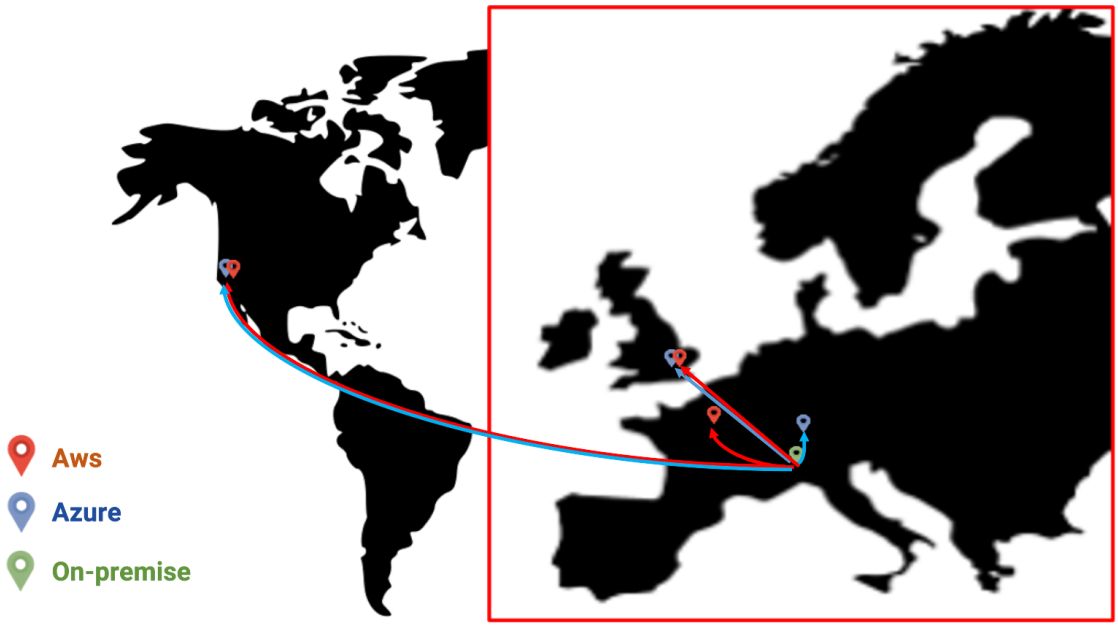


Figure 5.4: Flows of the on premise to cloud scenario.

Figure 5.4 represents the flows of the analysis, with the cliend deployed in the Winter server in PoliTo (Turin) and the server endpoints are:

- CrownLabs cluster in Turin, with the service exposed both through L4 and L7 load balancers;
- AKS cluster in Noth Switzerland, with the service exposed both through L4 and L7 load balancers;
- AWS virtual machine in West Europe, L4 load balancer;
- AKS cluster in South UK, L7 load balancer;
- AWS virtual machine in South UK, L4 load balancer;
- AKS cluster in West US, L7 load balancer;
- AWS virtual machine in West US, L4 load balancer.

The server deployed in a Kubernetes cluster, instead, leverages the benefit of having a declarative description of the deployment, where the only thing to add is the hostname where requested, depending on the domain name of the Ingress Controller. Here is the example file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: latency-tester
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: latency-tester
  namespace: latency-tester
  labels:
    app.kubernetes.io/name: latency-tester
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: latency-tester
  template:
    metadata:
      labels:
        app.kubernetes.io/name: latency-tester
    spec:
```

```
23      containers:
24      - name: latency-tester
25        image: richimarchi/latency-tester_server
26        args:
27        - -tls=true
28        imagePullPolicy: Always
29        ports:
30        - containerPort: 8080
31  ---
32  apiVersion: v1
33  kind: Service
34  metadata:
35    name: latency-tester
36    namespace: latency-tester
37  spec:
38    selector:
39      app.kubernetes.io/name: latency-tester
40    ports:
41    - port: 8080
42      name: latency-tester
43  ---
44  apiVersion: networking.k8s.io/v1beta1
45  kind: Ingress
46  metadata:
47    annotations:
48      nginx.ingress.kubernetes.io/secure-backends: "true"
49      nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
50    labels:
51      app.kubernetes.io/name: latency-tester
52    name: latency-tester
53    namespace: latency-tester
54  spec:
55    rules:
56    - host: ${HOSTNAME}
57      http:
58        paths:
59          - backend:
60              serviceName: latency-tester
61              servicePort: latency-tester
62            path: /
63    tls:
64      - hosts:
65        - ${HOSTNAME}
66        secretName: latency-tester-tls
```

With the access to the command line interface of Kubernetes, that is **kubectl**, with this file properly filled out, it is possible to easily deploy the server with this simple command replacing the last argument between the angle brackets with the name of the file:

```
kubectl apply -f <file.yaml>
```

The client is simply executed inside a Docker container, by giving as input parameter the name of the settings file described in the previous chapter. Once gathered the data from this computation lasted 24 hours, the following takeaways are extracted.

5.1.1 The impact of core network routing

When talking about the communication delay, obviously it is not possible not to take into account the physical distance between the two endpoints, the propagation delay, that is the first and unavoidable obstacle. Therefore, the first important thing in the communication is to traverse the shortest and fastest path possible from the client to the server and back. The protocol responsible for that is *IP*, the standard routing protocol adopted in the internet, that uses a certain strategy to forward every incoming packet towards the right direction in the quickest and most balanced way possible, taking into account hops and traffic.

During the standard test conducted using Winter server as client, it has been noted a particular difference in the delay between each run. The initial time interval between them was one hour, so it has been reduced to one minute and the same result was evident.

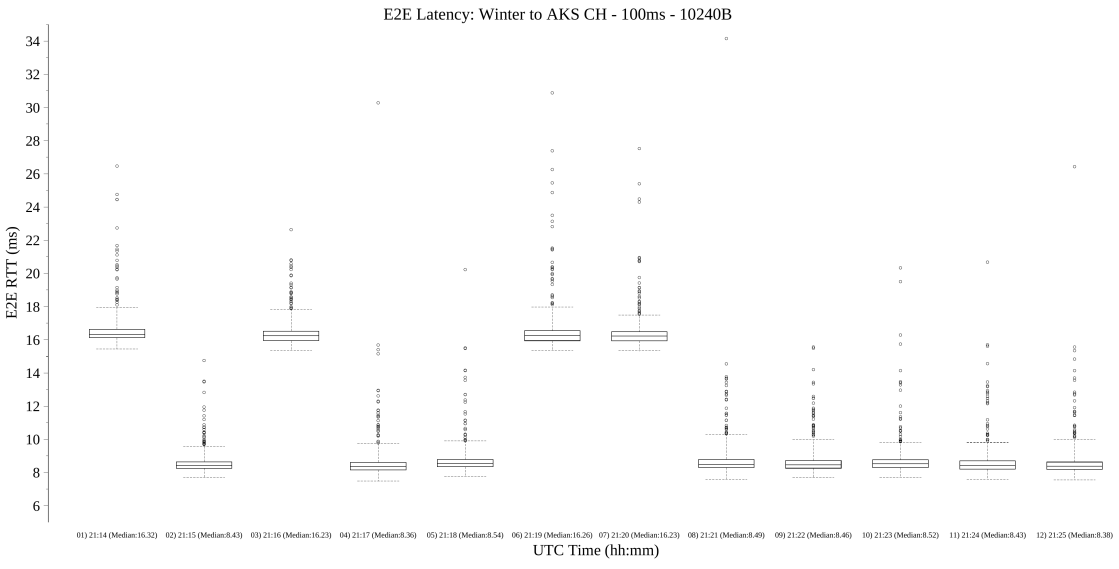


Figure 5.5: Core network routing effect on E2E latency.

Figure 5.5 is the boxplot representation of the end-to-end roud-trip time, with the median result reported in the X axis along with the time of execution. As we can see in this image, there have been executed 12 runs from the Winter client towards the same AKS server in Zurich, using the same exact configuration of 100 ms as send interval, 10 KB request message and 1 KB response message. The runs are executed almost at the same time (i.e. just 1 minute interval) and by looking at the graph it is possible to see that there is a clear 8 ms E2E latency difference between two sets of runs, one lying around 8 ms as median and the other around 16 ms.

Using a common command line interface tool called *traceroute*, it has been discovered a particular behavior of the communication: once the packet was delivered into the Internet, it was randomly routed through two different pathways, therefore resulting in two different physical distances and again two different latencies. The general routing strategy used is the **Equal-cost multi-path (ECMP) routing**. It is a routing strategy where packet forwarding towards a single destination can go through multiple best paths with equal routing priority, adopting a per-hop decision-making technique at each router independently. The Winter server is inside the PoliTo network, that has the GARR network as Internet provider, so each packet that goes towards the Internet has to go through this one. Exactly in this network, it has been discovered that with ECMP the packets are directed through an Internet exchange point in Milan or in Rome, where it enters the Microsoft backbone and is then routed towards the final destination. Deepening the analysis, we also tried to execute the latency-tester client with a fixed source port and the result was different: the median latency was the same for all runs, stable at either 8ms or 16ms depending on the source port chosen. In fact, by doing this, all the packets in all the runs had the same quintuple, that is the group of L4 protocol, source address, source port, destination address and destination port, used to identify the same TCP stream. Typically, ECMP routing is performed at session level based on the quintuple, to prevent reordering which is problematic in case of TCP as it triggers retransmissions, therefore explaining the behavior experienced in the tests.

Lesson Learned

From this analysis it is possible to see that the quickest pathway is not always granted, because the routing protocols in place in nowadays networks could have

some problems and are not impeccable. In this particular case the delay worsening is not that much important, but it could be not negligible if we deal with a safety critical application, that requires a very fast communication between the endpoints.

As explained before, the possible solution is to try to understand which is the quintuple combination that allows your stream to take the quickest path, maybe choosing to use a custom source port as it has been done with the latency-tester. Another way to fix this behavior is to force the packets directed to a certain group of destinations to go through a defined link, but of course the only way to do that is to get in touch with the responsible of this routing strategy, in our case the GARR consortium, and ask them to implement a specific rule in their network.

5.1.2 Interaction with different Cloud providers

The cloud computing spread has affected every kind of commercial activity, since it offers an easier and more scalable way to do the business by only having to focus on those aspects that concern the real objective of the company. Moreover, what really helps them to separate the infrastructural point of view from the business one is the fact that there are few IT giants that offer a platform where they can rent the services needed without the necessity to put in place their own on premise data center infrastructure.

Exactly for this reason, a company that wants to rent the so called *public* cloud computing service has to analyse many facets of every solution in the market, in order to find the most appropriate one depending on the needs of the company itself and those of the application that will be deployed in that infrastructure. Of course the first focal point of the discussion is the economic one, because costs are definitely an important factor in every business choice. On the other side, it is also very important to guarantee to the user the best user experience possible performance-wise so that the service offers the desired QoS.

In Figure 5.6 the Winter server acting as client evaluates the connection performance towards servers deployed in the AKS Kubernetes cluster in Zurich and London regions and the AWS virtual machines in Paris and London regions. The send interval is 100 ms, with a request message of 100 KB and a response message of 1 KB. The decision to choose the Paris region for the AWS virtual machine is forced because of the fact that with the student's plan of that service there was not any nearer one available. It may seems that this aspect invalidates the analysis, but it offers another point of view when it comes to choose the best provider instead. In

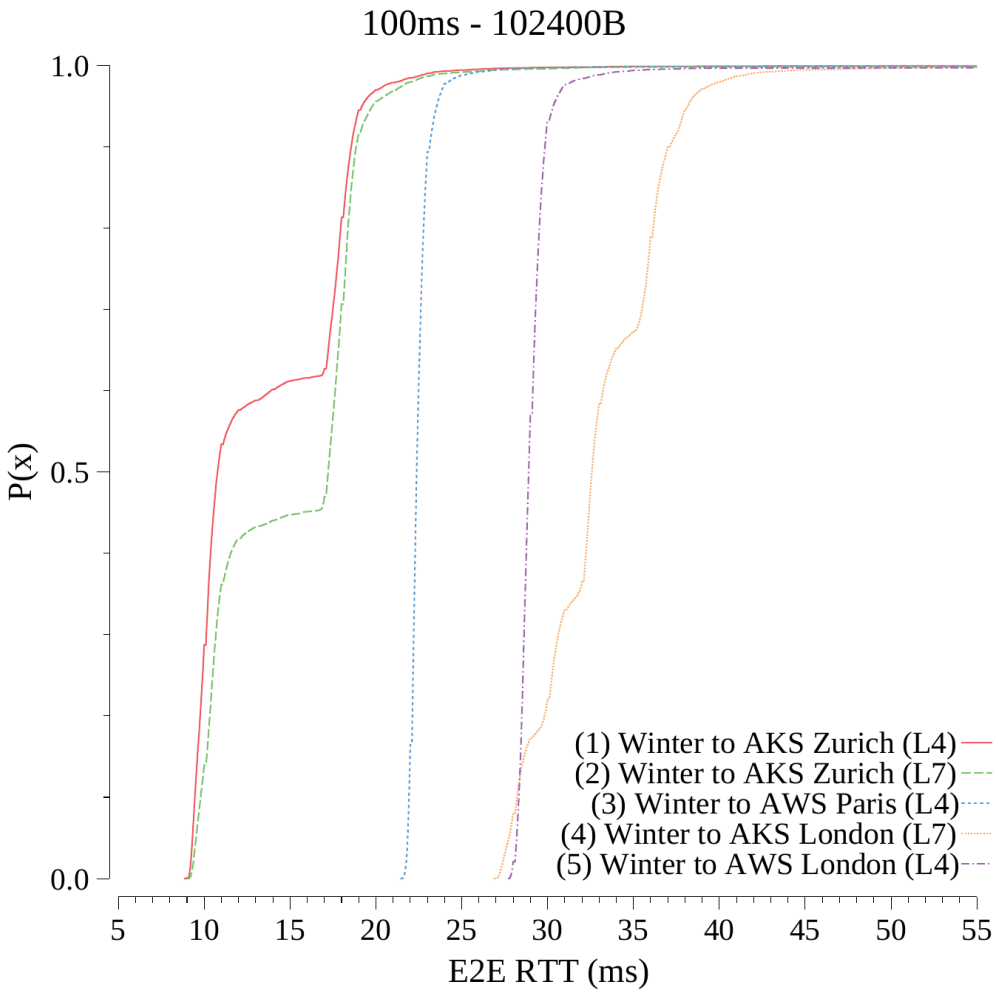


Figure 5.6: Performance towards different cloud providers.

fact it is important to always keep into account where the users are going to access the service from and also evaluates the better hardware with the best performance, since it will play a big role. From the graph above it is possible to see that of course the communication performance towards AKS Zurich is way better than the one towards Paris, because the physical distance influences the result a lot. When comparing AKS London and AWS instead, it is possible to see that the result is the opposite, partly because AKS uses a L7 load balancer, that slightly impacts the latency, partly because of other reasons: better hardware, less traffic congesting the network, a different way of reaching the destination network. Talking about this last one, the packet pathway towards the destination could be a significant

factor in the overall E2E delay, because it depends whether the packets enter the provider's network almost immediately or near the final destination: supposing that the segments of a message needs to reach the London region from Turin, probably it will go through an Internet exchange point where it enters the backbone of the destination provider and eventually reaches the application server. In the case the nearest IXP is connected to only one provider, probably that one is the most convenient one, because in the next sections we will see that the provider internal backbone tends to be faster. Notice that the hump for lines 1 and 2 is due to the ECMP routing problem described in the previous section.

To better describe this behavior comes in help Figure 5.7. The client has been deployed in a server on-premise located in Turin, whereas the server is running in two Kubernetes cluster: CrownLabs (PoliTo network) and Azure (Zurich). Thinking about what could happen, anyone would expect that the communication between the client and the CrownLabs server would be way better than between the client and the Azure one, because they are physically much more near the one to the other (same exact city). By looking at the graph we can see that it is the other way around. The way the two cluster are connected to the Internet plays a big role in this communication and the result is that a Turin to Zurich communication has a lower latency than a Turin to Turin one, probably because the packets of the exchanged messages have to go through a much longer path due to routing inefficiencies.

Lesson Learned

The public cloud is a fundamental resource for the companies and should be exploited as much as possible because it allows them to focus only on the business side of the job. If properly used, it allows to save much more money than having and managing an on-premise data center, due to the scalability of the exploited resources.

When it comes to decide which cloud provider to choose where to deploy a certain application, it is important to keep track of many factors:

- **Costs**: each cloud providers has a different price list depending on the infrastructure in place and the strategical position, therefore to maximize the profit this is not a negligible aspect;
- **Physical distance**: if the service deployed is accessed only by users residing

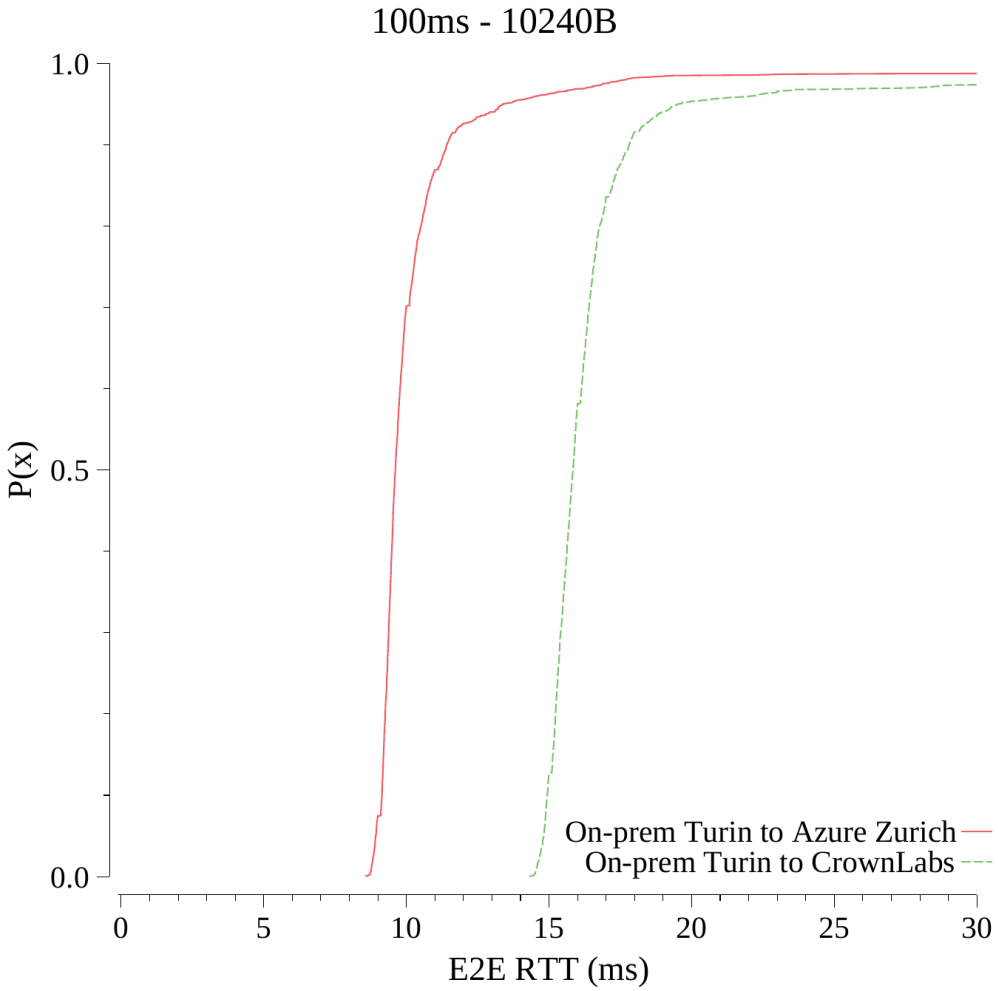


Figure 5.7: Comparison between near endpoints.

in a certain zone, it is the best choice to choose that zone, because of course the propagation delay and the policies involved affect the E2E latency;

- **Network infrastructure**: take into account how the provider is connected to the Internet, control that the path followed by the packets is not much longer than the expected one, otherwise it would affect the user-experience;

- **Hardware performance**: providers have a different set of hardware components, therefore check that the chosen one respects your application needs.

5.1.3 L4 vs L7 Load Balancer performance comparison

One of the most important aspects to configure when deploying an application is to make sure that it is available for the user to contact it the best way possible. Traditionally the service would have been exposed using a web server but nowadays the trend has changed. In fact, since the new applications developed are based on the microservice architecture, the deployment is usually done inside a Kubernetes cluster, in order to facilitate the whole process.

As seen in a previous chapter, Kubernetes creates more than a single instance for an application and makes it available to be contacted through a resource type called Service. The service is a virtual entity that redirects the request to one of the backends running in the cluster. But by default this resource can be accessed only by the internal network, so we need to put in place a way to make it reachable from the outside, otherwise it would be useless.

There are two solutions available that presents different characteristics:

- **Layer-4 LoadBalancer**: it is done by publishing the application with a public IP and a port that identifies that specific service. As said in the name, it uses a layer-4 routing and there is not any node in the middle that terminates the connection. It is a bit inconvenient and costs more because it reserves a public IP.

- **Layer-7 LoadBalancer / Ingress Routing**: it is done by deploying an Ingress Controller in the cluster that operates as reverse proxy and manages the incoming requests with defined rules. The service is published with a hostname, with the routing executed at application level, and the connections towards the applications are terminated by the controller that will then forward them to the appropriate backend. This solution is easier to use and cheaper, therefore the most employed one.

The NodePort solution is excluded because usually the nodes of a Kubernetes cluster are not exposed with public IPs, so again the service would be only accessible from the internal network.

However, even if there is a solution that is the most convenient one, it is interesting to analyse the difference in performance of the two available solutions. Figure 5.8 shows the comparison of the two load balancers inside a private cloud Kubernetes cluster and a public cloud one.

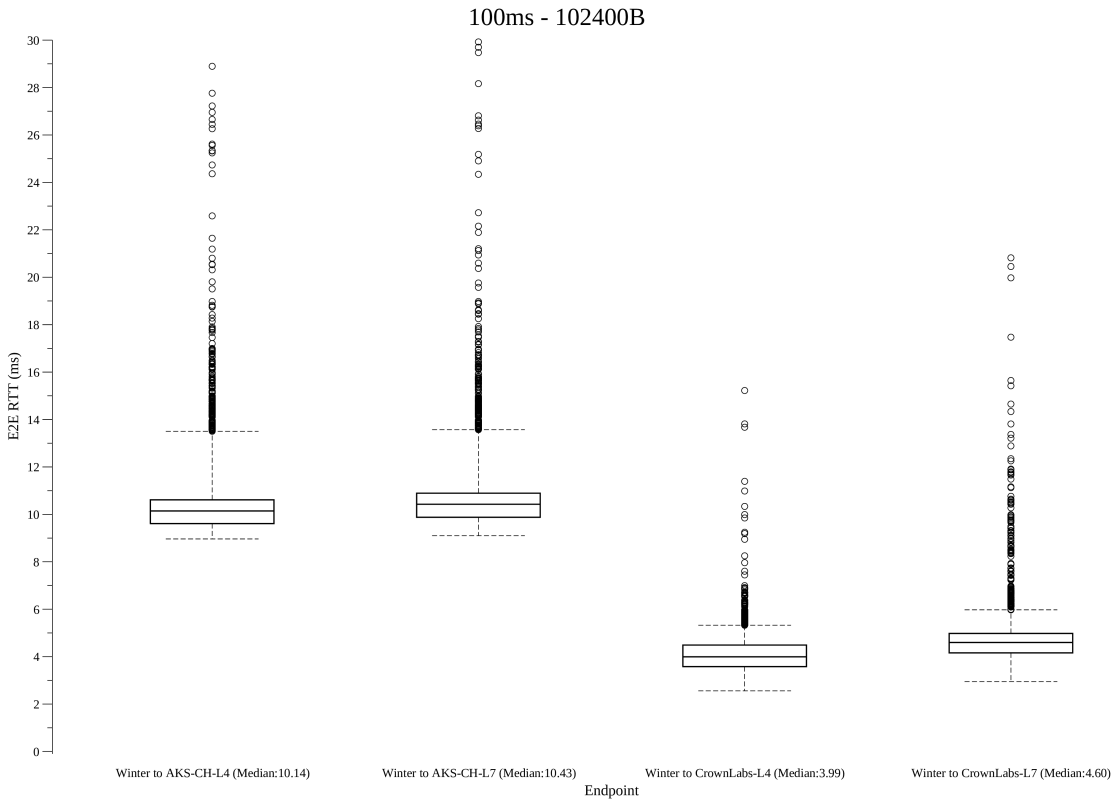


Figure 5.8: Comparison between load balancer configurations.

The test has been executed deploying the server in the CrownLabs Kubernetes cluster in PoliTo and AKS Kubernetes cluster in Zurich, in both environments exposed using a L4 and a L7 load balancer. The client has been deployed in the Winter server in PoliTo. The send interval is 100ms, the request message size is 100 KB and the response message size is 1 KB. It is important to keep in mind that there is no interest in showing the absolute performance difference between the two endpoints, because in this case it is irrelevant, but the two destinations have been chosen because they are near the client and this way the latency difference is more evident if present. In fact, by looking at the graph, the overall performance is pretty similar but it is possible to notice that there is a little deterioration when dealing with a L7 load balancer. The reasons behind it could be multiple: the TLS channel termination, the ingress controller component not involved in L4 load balancing. However the comparison is so similar that the E2E communication is not affected very much and the benefits of a L7 load balancer play a big role.

Lesson Learned

The exposition of a service to the external network, so that it can be accessed by users, is a key aspect of the deployment of an application. Nowadays, they are based on a microservice architecture, therefore the environment exploited is Kubernetes. This platform provides two different resource types to publish a service and each one has its positives and negatives sides: the L4 load balancer offers better performance but it is inconvenient and more expensive; the L7 has more components involved, affect the communication latency between the client and the backend but it is more easy to use and typically cheaper.

Since the overall performance of the two solution almost coincide, the benefits brought by a L7 load balancer beat the L4 load balancer and this is the reason why it is the most adopted resource to expose an application from inside a Kubernetes cluster. On the other hand, in case of services that require very tight latency constraints the L4 one may be the best choice.

5.2 Cloud to Cloud

This second scenario is a more complex one and presents the deployment of the client and the servers in all the available providers and regions so that it is possible to analyse the performance between different public clouds.

Figure 5.9 represents the flows of this analysis. The clients are deployed in:

- Winter server in Turin;
- AKS cluster in Noth Switzerland, inside a K8s pod;
- AWS virtual machine in West Europe, inside a Docker container;
- AKS cluster in South UK, inside a K8s pod;
- AWS virtual machine in South UK, inside a Docker container;
- AKS cluster in West US, inside a K8s pod;
- AWS virtual machine in West US, inside a Docker container.

The servers are deployed in:

- CrownLabs cluster in Turin, both L4 and L7 load balancers;

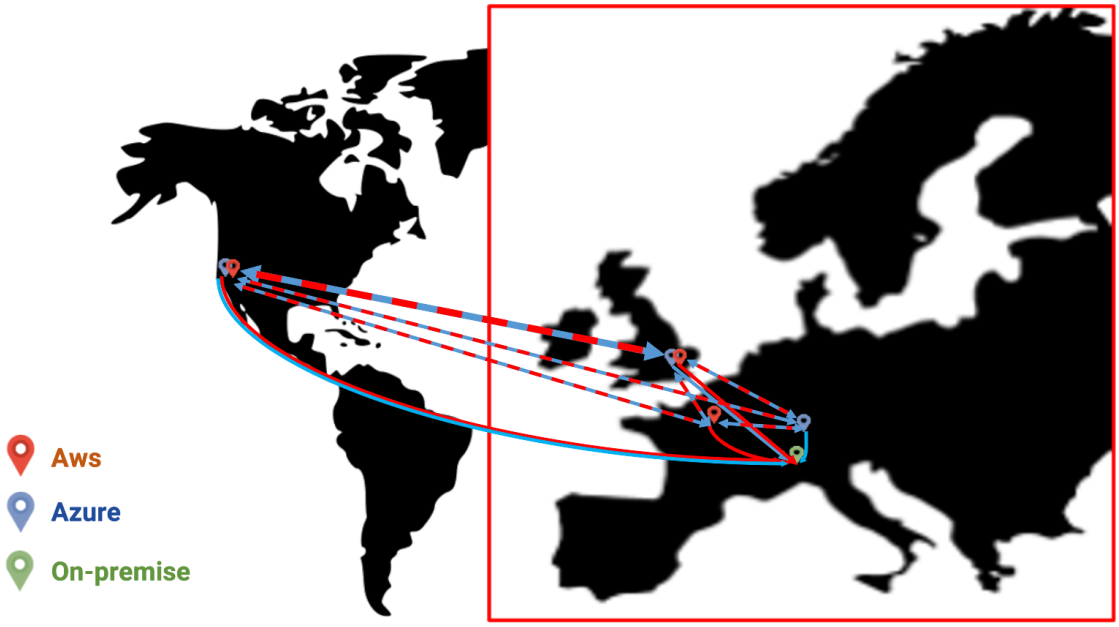


Figure 5.9: Flows of the cloud to cloud scenario.

- AKS cluster in Noth Switzerland, both L4 and L7 load balancers;
- AWS virtual machine in West Europe, port mapping acting as L4 load balancer;
- AKS cluster in South UK, L7 load balancer;
- AWS virtual machine in South UK, port mapping acting as L4 load balancer;
- AKS cluster in West US, L7 load balancer;
- AWS virtual machine in West US, port mapping acting as L4 load balancer.

The workflow for each endpoint involves to contact all other destinations but itself. The deployment details for Docker and Kubernetes environments can be seen in the previous section.

5.2.1 Long-distance network performance

Examining a situation where a company has many branches spread around the world, the performance of the communication between these locations must be on point: for example, a call carried out through a VPN service between the two sites must have a response time that is under 200ms, otherwise the user-experience would

be so bad that it would be impossible to have a clear conversation. For this reason it has been executed a test, examining what is the behavior of a similar situation using the latency-tester. The gathered results are represented in Figure 5.10.

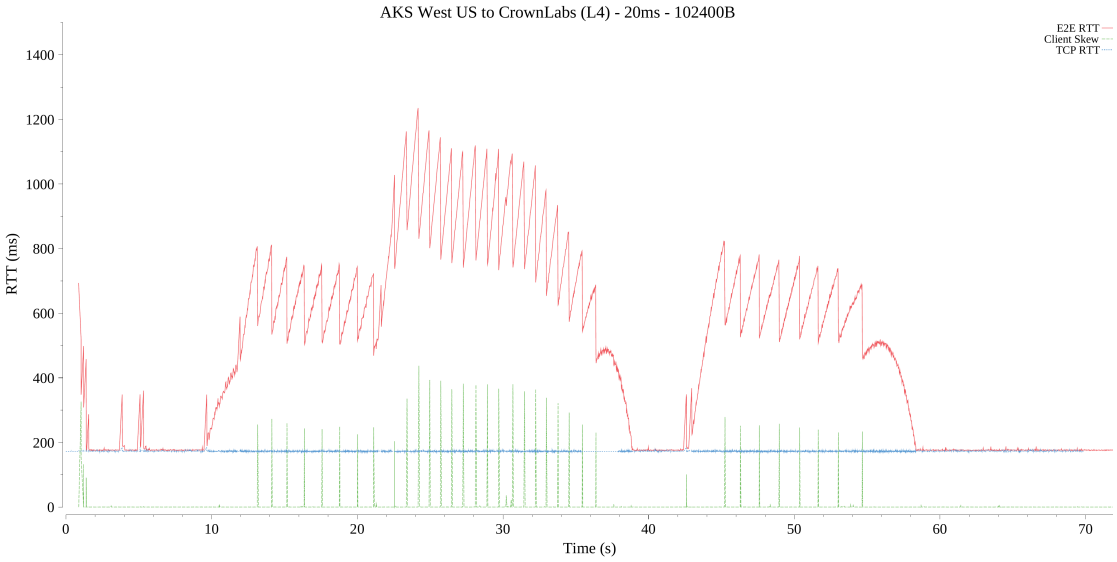


Figure 5.10: Long distance network behavior.

In this scenario the server has been deployed in the CrownLabs Kubernetes cluster in PoliTo whereas the client in the AKS cluster in the West US region. The send interval is 20 ms, the request message size 100 KB and the response message size 1 KB. The red line represents the application E2E round-trip time, the green one the transport-layer round-trip time, that is the time elapsed between the send of a segment and the reception of its ACK, and the blue one the client skew, that is the difference between the send of consecutive messages independently from the send interval.

What is interesting to see is the fact the the E2E latency reaches very high values, deteriorating the communication in an excessive way. On the other hand, it is possible to notice that the TCP round-trip time is pretty stable in the behavior. The client skew highlights the fact that there is not a regular behavior in the send of the messages, therefore it probably states that the local socket buffer of the sender gets filled due to some TCP stall and forces the segments to wait before being sent, hence leading to an head-of-line-blocking effect.

The problems that could cause the overhead detected in the graph above are due to the long standard network round-trip time, that stretches the duration of

the procedures and reduces the responsiveness of the communication. For example, the congestion window grows slowly, causing a tight throughput at the start of the transmission, and the packet losses limit the bandwidth perceived by the application, because the loss event is detected later, causing retransmissions after a bigger time span.

Lesson Learned

Intercontinental communications are becoming more and more a trend for those companies spread around the world. But the important aspect to take into account is the fact that the propagation delay could also cause bigger problems on the protocols involved, even though some of them are designed to properly adapt on the underlying network infrastructure. In some cases the effect on the application latency could be catastrophic, making the long-distance solution useless.

Obtaining good performance in terms of bandwidth and latency on long-distance networks (e.g. between Europe and US) turned out to be definitely more difficult compared to situations in which the two endpoints are closer. This issue depends both on the high network RTT, since TCP timings mostly depend on that, as well as the additional links/devices that need to be crossed during the path, increasing the probability of congestion and packet loss.

5.2.2 Intra provider vs Inter provider communications

One of the most important principles when deploying an application is the *fault-tolerance*, that means that it is necessary to put in place a disaster recovery plan to avoid downtime of the service. For this particular reason, it is important that the application backend is distributed in more than one region. Other than that, when a company has to choose where to deploy a certain service, it has to take into account many factors, like explained in a previous scenario, and then decide which provider is the most suitable one. But after some time the company may realise that another provider has some features that would be very useful to be integrated in the custom service developed, therefore it has to rent a cluster along with the desired feature. This solution where more than one provider is adopted is called **Multicloud** and it is very much widespread nowadays, letting these services to interact and cooperate with each other even if they do not belong to the same provider. Moreover, in the case the company has a local private data center that

would like to integrate along with the other public clouds so that they would combine and work together, it would have a **Hybrid cloud**.

Having until now primarily analysed the on-premise to public cloud situation, therefore the hybrid one, the question that arises is if the performance of a multi-cloud environment is the same of a single cloud provider one, so that it is possible to understand if there is the possibility to choose a different provider in a different region, e.g. if it offers lower prices, or keeping the same vendor represents an advantage latency-wise. Figure 5.11 shows the E2E latency differences between intra and inter cloud communications.

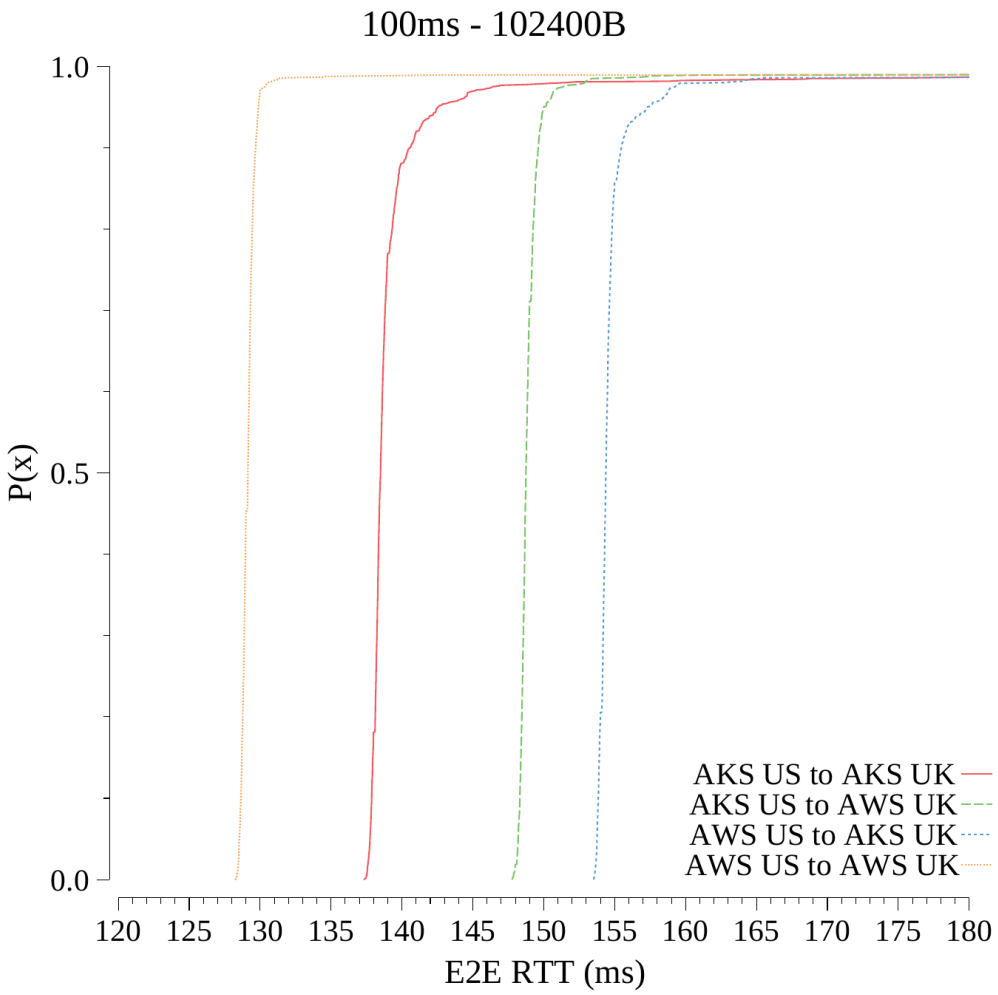


Figure 5.11: Intra vs Inter provider communication performance comparison.

In this test there have been used the AKS and AWS infrastructures. The clients

have been deployed in the respective data centers inside the West US region, whereas the servers in the data centers inside the South UK region. The send interval in this use case is 100ms, the request message size is 100 KB and the response message size is 1 KB. By using a long-distance communication environment is easy to spot the difference in performance in the use of multi-cloud compared to a single-vendor solution. In this specific case the AWS to AWS communication is the best one, immediately followed by the AKS to AKS one. The reason for this difference between the two cloud providers cannot be discovered, but it is easy to say that the internal backbone inside each vendor has a performance that is far better than having to go through a IXP and accessing a different network. On the other hand, it is possible to also notice that AKS to AWS is better than AWS to AKS. This difference is not traceable with any tool, because inside the backbone of any provider there is no possibility to use ICMP to discover the path followed by the packets. The hypothesis behind this behavior is that it depends on where the packet exchange happens between the two vendors and therefore it could happen that the packets of a communication follow a longer path than the ones of the other.

Lesson Learned

It is a wise choice to deploy an application in more than one region and, in order to take the most profit possible from a certain environment configuration, it is very important to evaluate which provider offers the best solution in each region.

A multicloud solution would probably bring many positive aspects and different useful features, but if the main requirement in the communication is to have the lowest latency possible, then the single-provider solution is the most suitable one, because it is possible to achieve lower delays between data centers and the packets would never leave the vendor's backbone, with the guarantee to have a direct path from the client towards the destination and therefore less congestion given by IXP and policies control.

5.2.3 Time of the day influence

The way internet is used is very diverse during a 24 hours time span. From 9 to 18 the network is used mostly for work purposes, whereas in the evening people tend to use it to stream video or play video-games. On the other hand, at night, most of the people are asleep and therefore the Internet traffic is not that much,

but it could be used by data centers to exchange huge amount of data instead, like something similar to backups and machine migration. But still the workload is much less than the one during the day [20]. To sum up, in the daytime there is the peak of traffic whereas in the nighttime the core network has some breathing space.

Giving this as a fact, it is interesting to understand if by using the network in different hours of the day it is possible to see any particular behavior due to some kind of congestion caused by the other traffic around, or the effect of that aspect is irrelevant.

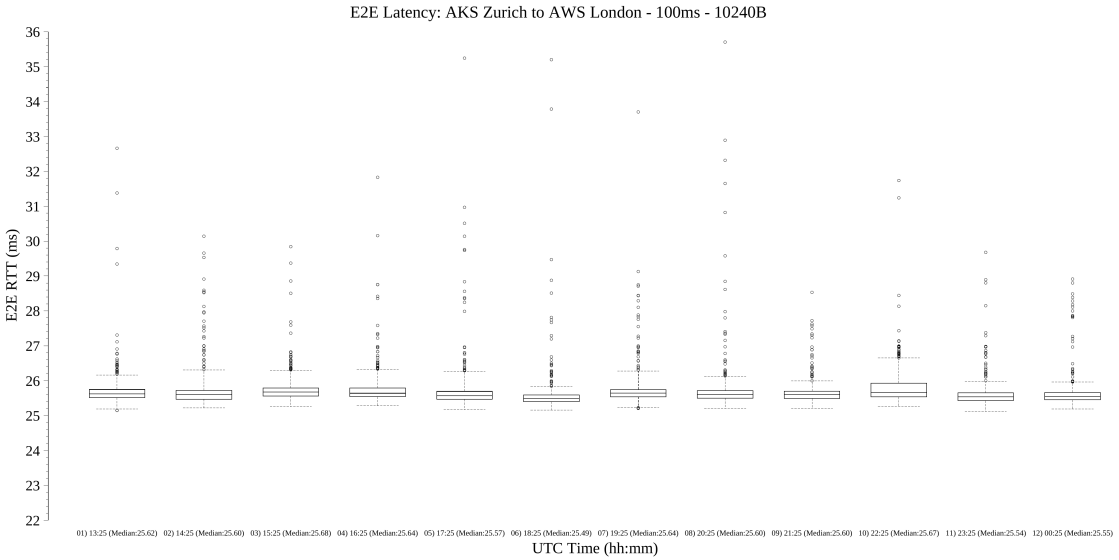


Figure 5.12: E2E latency during the day.

Figure 5.12 is the boxplot representation of the end-to-end latency of 12 different runs separated by 1 hour between a client deployed in a AKS cluster located in the Zurich region and a server deployed in a AWS virtual machine located in the London region. The send interval is 100 ms, the request message size is 10 KB and the response message size is 1 KB. This figure shows only the most congested 12 hours but this pattern is repeated in the other 12 not reported in this graph. There is a little bit of oscillation between one and another but it is almost insignificant as the median varies of only 0.1 ms during the 24 hours.

Lesson Learned

When considering the E2E latency of a service, it may be possible to assume that the E2E latency depends on the hour the server is contacted. Business hours or

evening time are the most congested time of the day and they surely affect the service availability but at the same time it is possible that the network itself could be a source of delay.

Still, from the different measurements, it appears not to exist any relevant and consistent correlation between the performance in terms of E2E latency and the time of the day and as the previous figure shows the difference between one hour and the others does not even go beyond 1ms.

5.3 Access to Cloud

This last scenario involves the analysis of a less powerful connection, where the bandwidth capacity is very low and therefore there could be more problems. But at the same time it is important not to ignore this kind of situation because nowadays user-oriented services are becoming more and more a trend.

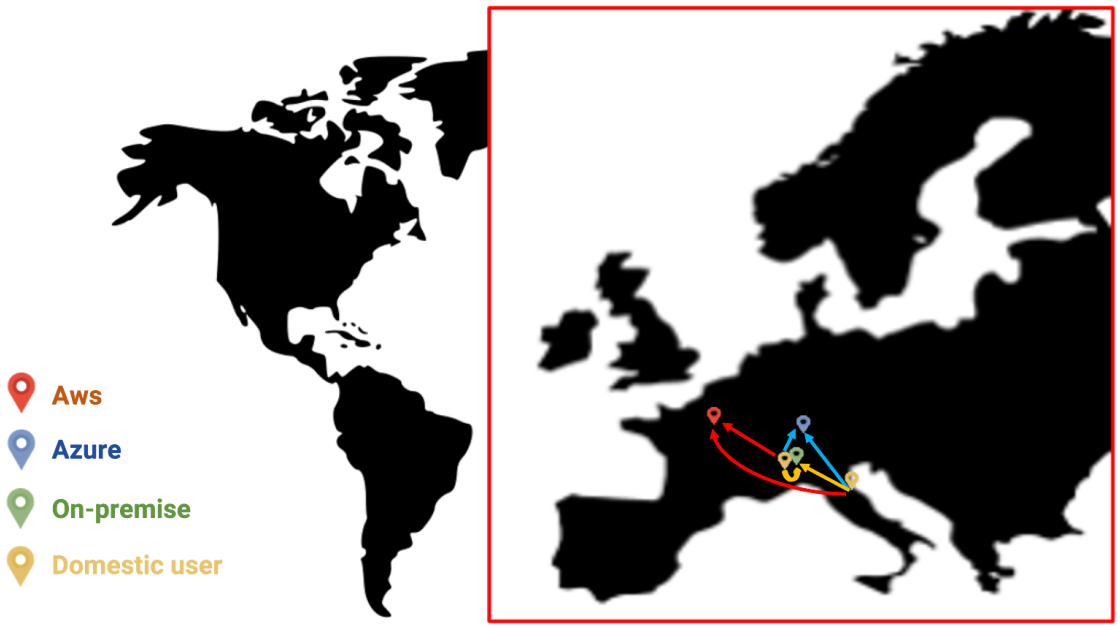


Figure 5.13: Flows of the access to cloud scenario.

Figure 5.13 represents the flows of this analysis. The clients are deployed in:

- Personal computer from home connection in Turin, inside a Docker container;
- Personal computer from home connection in Rimini, inside a Docker container.

The servers are deployed in:

- CrownLabs cluster in Turin, both L4 and L7 load balancers;
- AKS cluster in Noth Switzerland, both L4 and L7 load balancers;
- AWS virtual machine in West Europe, port mapping acting as L4 load balancer.

The deployment details for Docker and Kubernetes environments can be seen in the previous section.

5.3.1 Limited bandwidth consequence

Data centers and in general all the nodes of a backbone network, are provided with very powerful Network Interface Cards (NICs), that are able to guarantee a very high throughput. In order to properly exploit their potential, of course the cables that connect them to the network and with each other must be able to support a big workload, otherwise the employment of those NICs is wasted. Therefore the combination of these two powerful components makes available a huge bandwidth.

On the other hand, it is not possible to say the same for an average access network, or better not every user is able to reach a decent speed. Moreover, usually the Internet Service Providers (ISPs) tend to offer an asymmetric connection to the Internet, favouring a better download speed rather than the upload one, because it is factual that the average user receives more data than it generates. For this reason it is interesting to analyse what is the behavior of the E2E delay in case of a narrow upload bandwidth.

Figure 5.14 represents the E2E round-trip time between two clients and the same server located in the CrownLabs Kubernetes cluster in PoliTo. The two clients have been deployed in personal computers in two home networks, one in the province of Turin and one in the province of Rimini. The comparison between the two connections is possible since the average ping between the endpoints is the same, around 35 ms, because of the fact that the Turin one inexplicably goes through Rome before going back to the city. The bandwidth available for the two Internet connections is very narrow and different between them, because the Turin one has a speed of 15 Mbps while the Rimini one 3 Mbps. Moreover, let us specify that both computers were not loaded with other work so that the result of the test was not affected by any third party.

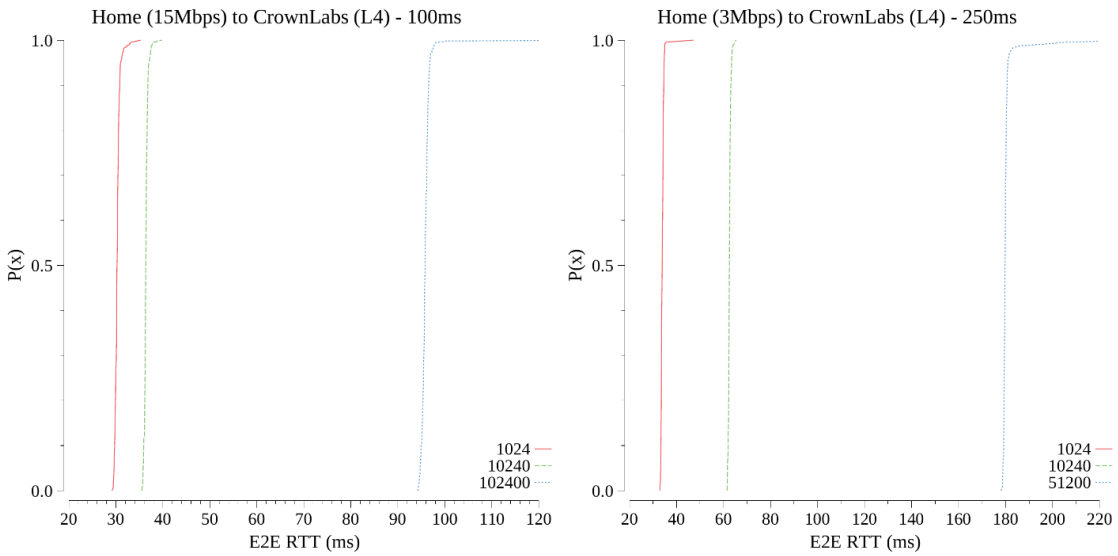


Figure 5.14: Performance comparison between different bandwidth.

One would think that at least with a very small message size the behavior of the two communication would be the same, because the bandwidth cannot be saturated in any way. Unfortunately the figure above shows that it is not that way. In these graphs there are represented three different request message sizes for each case. One thing that comes up is the fact that both clients have a pretty stable connections towards the server, because the CDFs are almost vertical indicating that the values were similar during the whole execution except few outliers. Even if the Rimini test is characterized by a longer send interval, so that it is harder to saturate the uplink, also with small messages there is a deterioration of about 5ms in the E2E latency. Increasing the size to 10K, the difference in even more evident, with about 25 ms between the two cases. As last message dimension, in order not to stress the connection and run into some sort of bottleneck, the Rimini one is half-sized compared to the standard 100 KB used in the Turin one: even if the rate is about 1.5 Mbps inside a 3 Mbps bandwidth, the performance is again very much worse than the other scenario.

Lesson Learned

The RTT at the network or TCP level is just one of the different factors influencing the E2E round-trip time perceived by the applications. Specifically, when dealing

with links, or more generally connections, characterized by relatively limited bandwidth (e.g. tens of Mbps or less), the transmission time can significantly impact the total E2E latency. Indeed, even with quite small messages (i.e. 100KB), the transmission time becomes predominant compared to the network RTT.

5.4 Common Behaviors

This section gathers the common behaviors that have been discovered during the analysis, independently from the scenario in place. As it is going to be presented, it is not only about the new technologies and the configuration chosen when dealing with unexpected higher latency, but the standard protocols in combination with the way the application works could cause the deterioration of the overall performance.

5.4.1 The Buffering problem

The previous real-case scenario offered a detailed analysis of what would happen in cases where the bandwidth available is limited, showing that, even if its capacity is not saturated, the application-layer round-trip time gets worse. However there could be some cases where the problem is not only due to the application throughput, but the presence of other traffic in the network plays an important role for the communication performance.

As described in a previous chapter, the TCP congestion control algorithm is an important feature for the protocol success but it could happen that in some cases it has the opposite effect, causing problems to the application relying on it. Especially, talking about the default congestion control algorithm implemented in the Linux kernel, CUBIC, it increases the transmission rate until any loss event is detected. This inevitably causes the buffers before the bottleneck link to be saturated. In this case the buffers are already full of segments coming also from other parallel TCP streams, so packets need to wait until the previous ones are sent towards the destination, hence increasing the perceived E2E latency. Figure 5.15 perfectly represents what has been just described.

This test has been conducted deploying the server in the CronwLabs Kubernetes cluster in PoliTo and the client in a personal computer in the province of Turin. The send interval is 50ms and both request and response message sizes are 1 KB. In order to create an environment where there are more than a single TCP stream, represented by the latency-tester, a parallel one has been simulated with a simple

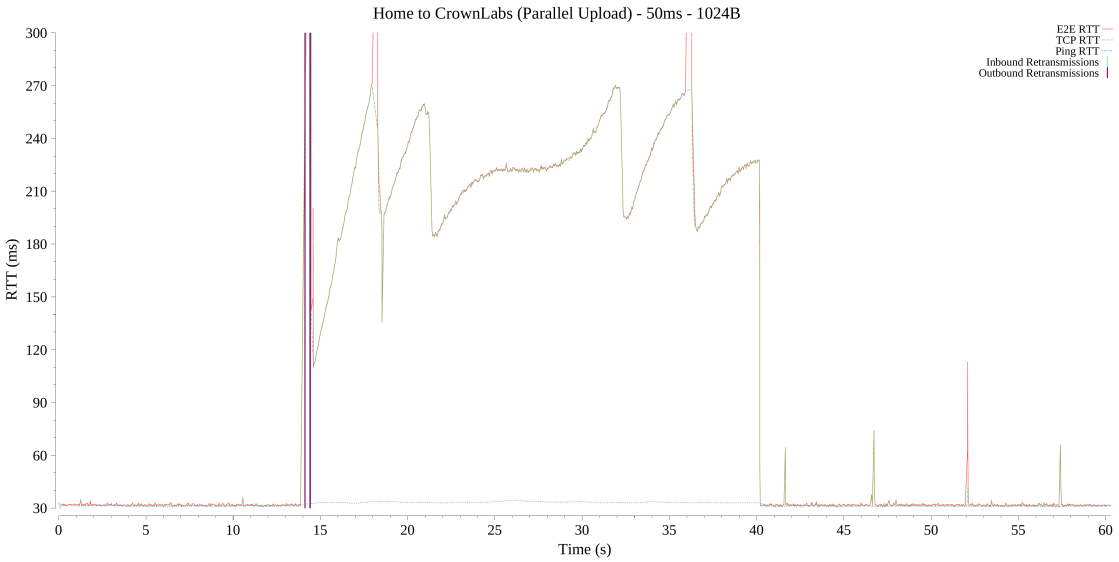


Figure 5.15: Bufferbloat detection during upload.

file upload, which starts at T=15 and stops at T=40. As explained before, it is clear that the communication starts very much stable, but as soon as the uploads is started both the TCP and E2E round-trip time starts deteriorating in a clear way, keeping an unpredictable behavior. Moreover, it is very interesting how it is possible to see how the CUBIC congestion control algorithm operates, because what happens between T=25 and T=32 is the delay that increases as the cubic function increases the congestion window. Once the parallel upload terminates, the connection has all the buffers for itself, so once again it returns to the expected application and transport layer delay, along with the network one. It is interesting to notice that the network RTT is stable throughout the whole execution because it does not depend on transport layer buffers.

Lesson Learned

The TCP CUBIC congestion control algorithm operates continuously increasing the transmission rate until a loss event is detected. Nonetheless, this behavior causes the filling of the buffers located before the bottleneck link, as a retransmission gets detected only when the buffer is full and packets start to be dropped.

However, this behavior damages parallel latency-oriented flows, as those packets need to wait for the buffers to empty before being transmitted, hence increasing the perceived E2E latency. This problem, named **bufferbloat** is particularly relevant

when there is a significant difference between the transmission rate (e.g. 1Gbps LAN) and the bottleneck link (e.g. tens of Mbps), and the buffers at the bottleneck are relatively big compared to the transmission speed. Common solutions to this problem involve the usage of smart queue management algorithms with different QoS classes, in order to prioritize latency-oriented flows. Nonetheless, those may not be available on low-end devices, such as home gateways. For this reason, the bufferbloat is still an open issue in the networking world.

5.4.2 TCP behavior implications

In one of the previous scenarios it has been discovered that the responsible for a particular delay in the message transmission may be the IP protocol, that due to the ECMP routing strategy make the packets go through a longer pathway. But between that protocol and the application one, there is the TCP, that even if it brings many benefits, it also has some drawbacks that comes along with its implementation.

The overview of the protocol and its peculiar features are presented in one of the previous chapters, but with the help of Figure 5.16, the discovered problem related to the TCP is now explained. This image is a boxplot representation of the E2E communication delay between Winter acting as client and the server deployed in a Kubernetes cluster in the Azure data center (Zurich). The request message is 100 KB while the response is 1 KB as always. The only difference between each boxplot is the send interval between the messages. As it is possible to see, as the send interval increases, there is a certain point where the round-trip time clearly increases too, up until a point where it stabilises.

It is important to highlight that this behavior was not happening when the request message size was 1K, therefore it made us think that it was something related to the transport layer, which is in charge of splitting the message bigger than the Maximum Segment Size (MSS) in L4 segments. Since with that set of data there was no explanation for this strange and damaging behavior, further tests have been conducted, deepening the approach and gathering more data. Adding to the tool the ability to retrieve the socket statistics, it has been found out that the problem was related to the congestion window size.

Figure 5.17 represents the sender TCP congestion window behavior during the whole execution of the previous test with 1 second as send interval. Once the client needs to send the message, the window increases, up until all the segments related

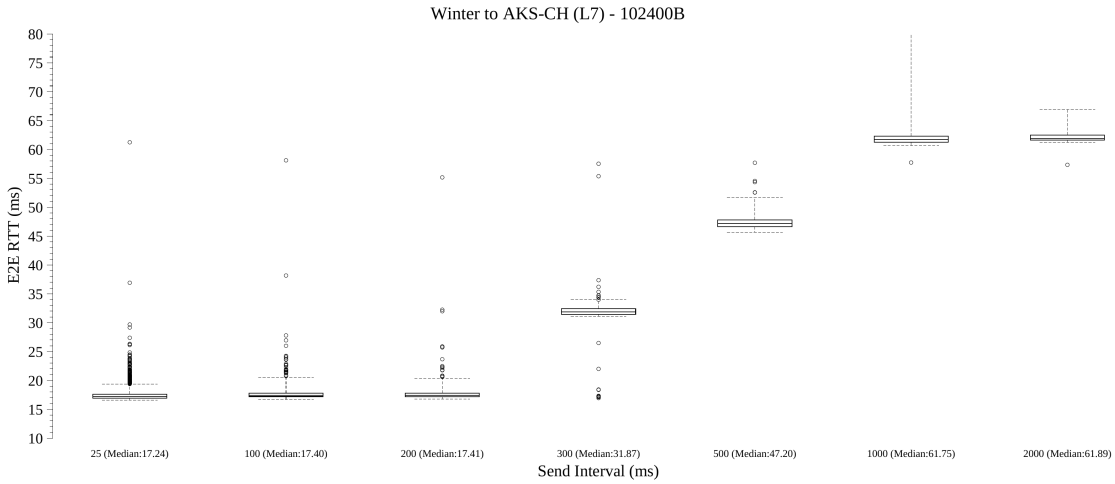


Figure 5.16: The effect of slow start on idle TCP connection.

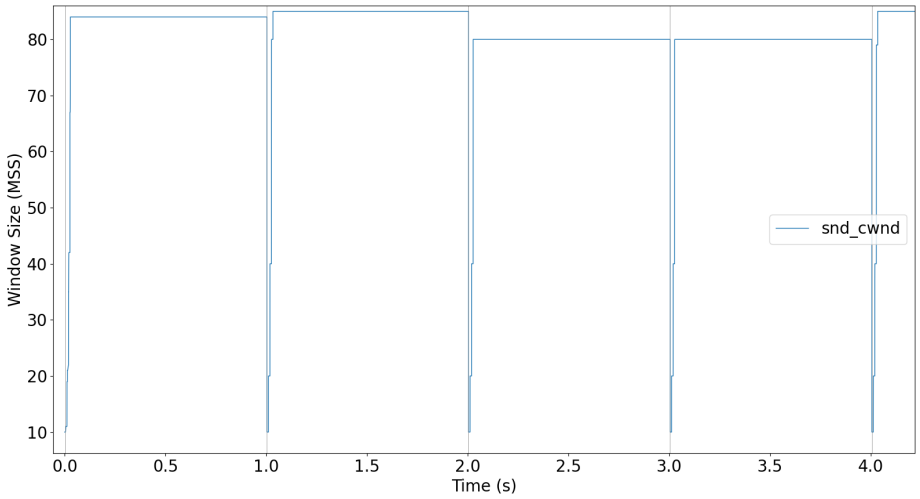


Figure 5.17: TCP cwnd evolution with default settings and 1 s send interval

to that message are sent towards the server. Then the socket remains idle until it has to send the second message, but by looking at the graph it is possible to see that the window decreases to the previous starting point and it has to grow again, hence requiring multiple RTTs to send the 100 KB messages. This goes on throughout all the execution. Given that this problem does not happen or it is less evident with lower send intervals, probably the source of all this is due to the TCP

congestion control algorithm implementation.

In fact, analysing the RFC related to the standard TCP congestion control algorithm [5], it emerges it would be a problem allowing a potentially inappropriate burst of traffic to be transmitted after TCP has been idle for a relatively long period of time. Indeed mutated network conditions may have rendered the TCP's notion of the available end-to-end network capacity between two endpoints, as estimated by cwnd, inaccurate during the course of a long idle period. Therefore the TCP triggers a slow start to recover from a connection that has been idle after a period of about 200ms, reducing the congestion window of the sender and again needing more RTTs to deliver the segments of a message.

As it is possible to see in the boxplot image reported above, with a periodic message exchage of about 1 second or more, this behavior would cause a relevant latency that could be avoided, so that the application is free to leverage all the available bandwidth. In order to do so, the solution discovered in this analysis is to handle the flag that manages the TCP slow start in case of an idle connection. The flag that needs to be modified in the sender executing environment (physical host or docker container) is the net.ipv4.tcp_slow_start_after_idle, which is by default set to 1 and needs to be disabled in order to avoid the reduction of the congestion window, maximizing the application throughput [21]. It is important to notice that all the test executed in this thesis are done with this feature disabled, so that the whole E2E latency was not influenced.

Figure 5.18 represents the execution of the application with the previous cited flag set to 0, therefore disabling the slow start after idle periods. Notice that after the send of the first message, the congestion window size is the same as in the previous figure was throughout the whole execution, whereas in this case, second by second and so message by message, the window increases until it maximizes the available bandwidth. It maintains that dimension up to the end of the communication between the two endpoints and therefore the deletion of the socket.

Moreover, it is important to notice that the congestion control algorithm used in these analyses is CUBIC, that is the standard one adopted by the Linux kernel. This particular aspect of the slow start after idle is not happening with other congestion control algorithms like BBR and Reno, because probably they overwrite the standard TCP behavior and does not let the congestion window to shrink. The reason behind this difference in these algorithms is because in the standard TCP congestion control algorithm RFC the slow start after idle is *RECOMMENDED*, therefore not mandatory to be kept in all extensions.

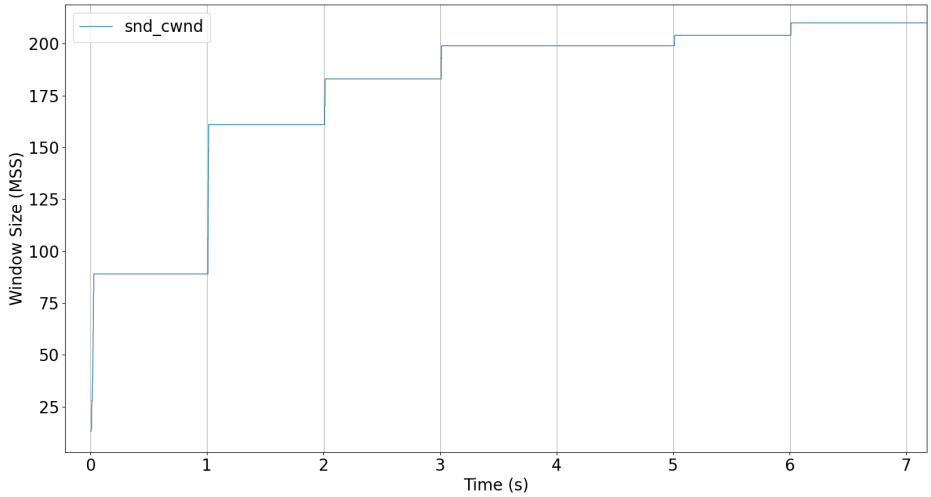


Figure 5.18: TCP slow start after idle not enabled.

Lesson Learned

The TCP is a very useful transport-layer protocol, which is highly recommended due to its reliability and its congestion control algorithm, that allows the user to transmit the right amount of data at the same time taking into account the available bandwidth and the traffic in the network. In certain situations, along with these benefits, it also has drawbacks caused by its implementation design, that would limit the effective throughput reachable by the application.

In the case the application developed uses a websocket, or any channel implementation that involves that the TCP socket is alive the entire execution, and there is a periodic exchange of messages, keep track of the application behavior and edit the protocol settings if needed. In the case presented above, it could be improved by modifying the net.ipv4.tcp_slow_start_after_idle and setting its value to 0, in order to disable the congestion window shrinking and achieve a better overall performance and quicker response time, as well as using a different congestion control algorithm. It is important to notice that all these modifications require admin privileges, therefore it is not possible to leverage them if we do not have the total control of the underlying infrastructure.

5.4.3 REST-like application vs Websocket

The objective of this thesis' analysis is to make an overview of what happens latency-wise in a cloud computing environment, taking into account a diverse set of real-case scenarios. Therefore, given the fact that we are dealing with microservices, the most important aspect of this pattern is the fact that it is designed using the REST (REpresentational State Transfer) architecture, the number one choice when talking about distributed systems. It is a subset of HTTP and is used because of its fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running. Moreover, just to express its crucial role in nowadays IT systems, also Kubernetes uses REST APIs as interface for the user.

As explained before, the latency-tester uses a websocket for the communication between the client and the server, so that it is not necessary to establish a new connection for every message. But is it really useful or is it possible to create a new channel every time a new request-response message is exchanged? Or better, does websocket bring a huge improvement in performance or its initial overhead is not worth?

In order to explore this situation, the tool has been modified so that it works as a stateless REST-like application: the protocol used in the message exchange is HTTP, that effectively works on a TCP connection. Therefore, for every message sent and received the client will have to establish a new TCP connection adding significant overhead in the process. Moreover, along with this performance deterioration, there may be another aspect that could impact it even more: in the case, as usually nowadays applications do, a reliable channel is requested to keep the information secure and private, in order to use HTTPS it is also necessary another procedure to upgrade the channel security, that is the TLS negotiation.

Figure 5.19 shows four CDFs graphs comparing the REST approach to the traditional one (noted as persistent) used by the latency-tester that is based on the websocket channel. The test has been executed using Winter as client and the server has been deployed in the Azure cluster in the Zurich region. The send interval used is 100 ms, the request message size is 1 KB and 100 KB, the response message size is 1 KB as always and the TLS has been used for the persistent connection while there are cases where the REST communication is secure (noted with "TLS") and where it is (noted with "nosec"), in order to better highlight its impact.

As it is possible to see, the websocket persistent communication is much more

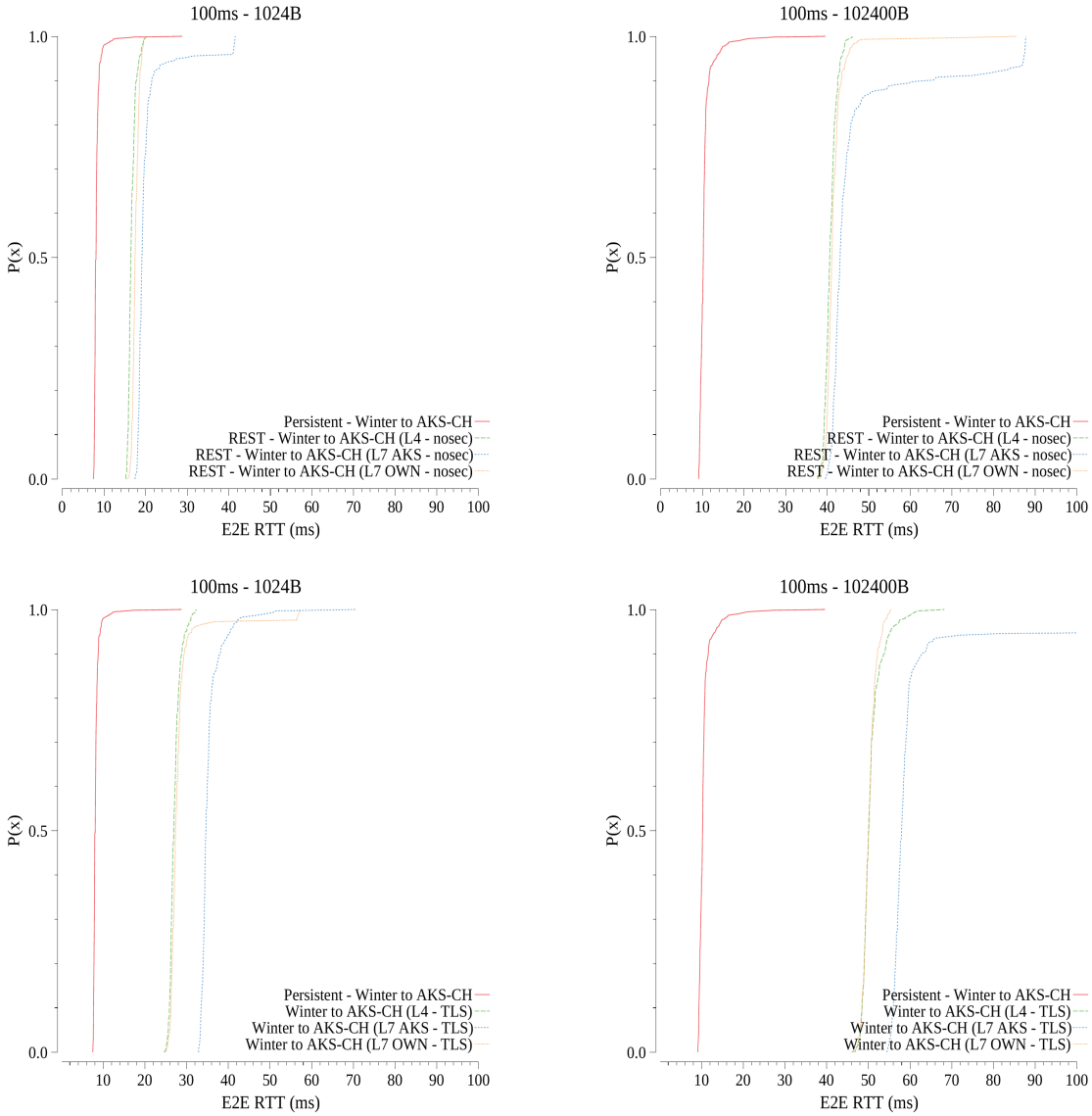


Figure 5.19: Websocket vs REST connections comparison.

efficient and regular than the REST one. In fact, by looking at all four cases, even with the use of TLS and the increase of the message size, the distribution of the application RTT is almost the same. Whereas the REST communications encounter more delay as expected. From the graphs it is possible to see that there is a clear deterioration of the E2E latency. With the increase of the message dimension this overhead is more relevant, but it increases even more with the use of HTTPS.

Of course, since the procedures and the segments delivery time depends on the network round-trip time between the two endpoints, it is important to notice that the increase in the application perceived delay is even worse if the RTT gets higher.

Analysing the possible motivations for this performance degradation in terms of application latency, there may be few reasons that explains why the persistent websocket channel is a better solution:

- There is no need to establish a new TCP connection, hence executing the three way handshake again and again, for every message exchanged between the client and the server, saving at least one RTT (in case TLS is enabled, its negotiation time for each message is avoided too);
- The client is able to leverage a bigger congestion window going on with the communication, because each message benefits from the increase triggered by the previous one (segments of the same message do not need more RTTs to be delivered but can be sent altogether as a block);
- The L7 load balancer will re-use the already-established connection towards the designated backend application server, hence preventing the initial overhead.

Just to clarify another strange behavior that can be seen in the figure above, the difference between the custom L7 load balancer and the one provided by Azure is due to the fact that the second one is not suitable for a production environment and apparently it does not support TLS 1.3 that reduces TLS negotiation from two to one RTT.

Lesson Learned

The REST paradigm is the most used one in nowadays application and its design plays a big role in the application performance. By default, any message exchange between a client and a server is done by establishing a different connection each time, therefore involving a huge amount of procedures that require significant time to set up the communication channel.

When dealing with applications that require a frequent message exchange between the two endpoints, the improvement brought by a websocket or any persistent communication channel makes a very big difference in terms of performance. TCP connection establishment and TLS parameters negotiation are very burdensome processes that with the help of such technologies would be executed just one time

throughout the whole execution of the application, significantly reducing the overall latency in the message exchange.

Chapter 6

Conclusions and Future Works

This thesis work summarises the analysis executed to discover more about the potential of the new technologies that are emerging in the cloud native environment and that will soon be predominant in the market. With the growing interest in developing microservice-based applications, the main requirement is to keep the end-to-end latency limited to defined constraints, especially for the most demanding ones. Therefore, it is important to understand whether such complex infrastructure is suitable for them. Even if, at first, most of the assumptions could have been conceivable, the effort put into this dissertation produced many interesting and useful results.

Although there were open-source solutions available to measure the latency of a certain application in a distributed environment, the intention to develop a custom latency-tester was born from the fact that those programs are typically very complex and may not offer the total control on any parameters and execution behavior. In addition, extending the test application with a tailored plotter was an important and useful move, because, even though the sharpening has been hard and time-consuming, the graphs generated were the best way to understand what was happening in the network by combining all the raw data gathered.

Afterwards, the analysis process has been executed. The resources available were limited to what a student could afford, therefore other than the PoliTo servers there were few available cloud providers restricted plans (Azure and Amazon Web Services). After repeating the tests, even more than once, with different parameters

and in many environment configuration, the final result was successful and it was possible to extract interesting conclusions. The main lessons learned with the work accomplished during this thesis are here reported:

- A REST-like approach leads to high E2E latency due to the overhead introduced by the connection establishment: a persistent connection has a better performance compared to a REST one, because whereas in the first case there is a single TCP connection throughout the entire communication, in the second case there is a new one for each message exchange.

- The application behavior can impact the network performance, since the protocols involved can behave in an unpredictable and inefficient way: the standard TCP congestion control algorithm, CUBIC, by default shrinks the sender congestion window in case the connection is idle for more than 200ms, therefore those applications that have lower message rate register higher latency than those with a faster one.

- ECMP routing in the core network can have unexpected influence in the E2E latency, causing the packets to take longer paths: in the case of the PoliTo network, to reach an outside destination (e.g. a data center in Zurich) the packet could sometimes go through a node in Rome rather than going through the nearest one in Milan, because the number of hops is the same.

- Traffic within the same cloud provider seems faster than multi-cloud: the connection that remains inside the provider, going from a region to another, performs way better than a connection that goes to a cluster of another provider in the same region. It is important to notice that this analysis is just preliminary and it would be interesting to deepen it with a bigger number of scenarios and providers.

- Different on-premise to cloud providers combinations can be characterized by different E2E latency, depending on many factors both tenant and provider wise: physical distance, network infrastructure and hardware performance are the main ones, surrounded by economic reasons that change the point of view of the tenant.

- Being physically near does not necessarily mean low RTT, due to the effect of possible routing inefficiencies: even though two peers are physically near, due

to the way their networks are interconnected or how far the closest IXP is, the performance could be much worse than expected;

- The E2E latency seems not influenced by the time of the day, because the backbone network is able to handle the traffic without any congestion even during the Internet rush hours.

- With limited bandwidths the transmission time becomes predominant, impacting the E2E latency even without saturating the available resources.

- Buffering and parallel transfers can drastically impact the RTT: the TCP CUBIC congestion control algorithm operates continuously increasing the transmission rate until a loss event is detected, causing the filling of the buffers located before the bottleneck link, as a retransmission gets detected only when the buffer is full and packets start to be dropped. However, this behavior damages parallel latency-oriented flows, as those packets need to wait for the buffers to empty before being transmitted, hence increasing the perceived E2E latency.

- It is much harder to get good performance with long-distance networks, since the tuning of the protocol requires more time: the issue depends both on the high network RTT, since TCP timings mostly depend on that, as well as the additional links/devices that need to be crossed during the path, increasing the probability of congestion and packet loss.

- Different Load Balancer types introduce limited differences, but the choice of the most suitable one depends on the specific scenario: the benefits brought by a L7 load balancer beat the L4 one and this is the reason why it is the most adopted resource to expose an application outside a Kubernetes cluster, but on the other hand, in case of services that require very tight latency constraints the L4 LB may be the best choice.

There is also an additional note that is interesting to point out: during the tests, the most important thing is to first of all assure that there are not any hardware problems in the testing environment, otherwise it would be difficult to spot it by only looking at the results of the test. Possible problems, which can get completely unnoticed with small RTT connections (e.g. few milliseconds), can turn out to have a much higher impact for higher RTTs. For instance, a malfunctioning cable/switch

which corrupts packets can drastically reduce the bandwidth, with devastating effects also on the E2E latency.

As a future work, an interesting move would be to keep improving the latency-tester and to add even more features so that it would be possible to gather even more data and to improve the data visualization, that at the moment is a bit burdensome and not so much handy (e.g. export data in the format used by Prometheus and then visualise it with Grafana). The project is open-source, so it would be great if any developer would like to contribute in any way.

Furthermore, on the analysis point of view, it is easy to say that the most clear continuation would be to obtain access to more public cloud providers and different on-premise deployment, so that with huge amount of resources it would be possible to make the most complete gathering of data and have the possibility to discover other interesting aspects. Especially, since the Edge computing adoption is slowly increasing along with new technologies like autonomous vehicles, it is important to understand if this solution brings real benefits instead of the traditional one. Another possible future work could be to analyse the wireless environment, by measuring the overall communication performance with different access technologies (i.e. Wi-Fi, 4G, 5G). Eventually, in case of the deployment of a latency-sensitive application, it would be interesting to measure what is the overhead of that particular infrastructure, so that we are able to evaluate the scenario itself and maybe discover possible problems.

The cloud native environment is so vast and in evolution that it would be a long and probably endless path to test every possible scenario.

Bibliography

[1] By Xiaolin Jiang, Hossein Shokri-Ghadikolaei, Gabor Fodor, Eytan Modiano, Zhibo Pang, Michele Zorzi, and Carlo Fischione. «Low-Latency Networking: Where Latency Lurks and How to Tame It». In: *Proceedings of the IEEE* (Feb. 2019) (cit. on pp. 1, 30).

[2] Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. «Latency Comparison of Cloud Datacenters and Edge Servers». In: *GLOBECOM 2020* (Dec. 2020) (cit. on pp. 2, 31).

[3] J. F. Kurose and K. W. Ross. *Computer Networking - A Top-Down Approach*. England: Pearson, 2013. Chap. 3 (cit. on p. 6).

[4] Information Sciences Institute - University of Southern California. *Transmission Control Protocol (RFC 793)*. `https://tools.ietf.org/html/rfc793`. Sept. 1981 (cit. on p. 6).

[5] M. Allman, V. Paxson, ICSI, and E. Blanton. *TCP Congestion Control (RFC 5681)*. `https://tools.ietf.org/html/rfc5681`. Sept. 2009 (cit. on pp. 6, 81).

[6] *Bandwidth-Delay Product*. `https://en.wikipedia.org/wiki/Bandwidth-delay_product` (cit. on p. 12).

[7] *What is Kubernetes?* `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes` (cit. on p. 19).

[8] *Kubernetes API*. `https://kubernetes.io/docs/concepts/overview/kubernetes-api` (cit. on p. 22).

[9] *How Fast is Real-time? Human Perception and Technology*. `https://www.pubnub.com/blog/how-fast-is-realtime-human-perception-and-technology/` (cit. on p. 28).

[10] Jim Gettys and Kathleen Nichols. «Bufferbloat: Dark Buffers in the Internet». In: *Communications of the ACM* (Nov. 2011) (cit. on p. 29).

[11] *The Bufferbloat Project*. `https://www.bufferbloat.net/projects/bloat/wiki/TechnicalIntro/` (cit. on p. 29).

[12] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. «Understanding Operational 5G: A First Measurement Study on Its Coverage, Performance and Energy Consumption». In: *SIGCOMM '20* (July 2020) (cit. on p. 30).

[13] Chanh Nguyen, Amardeep Mehta, Cristian Klein, and Erik Elmroth. «Why cloud applications are not ready for the edge (yet)». In: *SEC '19* (Nov. 2019) (cit. on p. 31).

[14] Riccardo Marchi. *Latency Tester*. `https://github.com/richiMarchi/latency-tester` (cit. on p. 33).

[15] *GoLang*. `https://golang.org/` (cit. on p. 35).

[16] *Tshark*. `https://www.wireshark.org/docs/man-pages/tshark.html` (cit. on p. 38).

[17] *Iperf*. `https://iperf.fr/` (cit. on p. 38).

[18] *Docker*. `https://www.docker.com/` (cit. on p. 40).

[19] *Gorilla Websocket*. `https://github.com/gorilla/websocket` (cit. on p. 42).

[20] *Internet Rush Hour*. `https://en.wikipedia.org/wiki/Internet_rush_hour` (cit. on p. 73).

[21] *Docker run command*. `https://docs.docker.com/engine/reference/commandline/run/` (cit. on p. 81).

Acknowledgements

And now the end is near, and so I face the final curtain...

Between relief, pride and a little bit of melancholy I finally conclude my career as student and a new chapter of my life begins. In these last 2 years at Politecnico di Torino I learned a lot, I found my way and above all I grew as a man.

I want to thank prof. Fulvio Risso, for how he teaches with passion and deals with students. It has been a pleasure attending his lessons and being able to do many projects and this thesis work under his supervision. A special thanks goes to PhD. Marco Iorio, that with his knowledge and kindness supported and helped me during this final project.

The biggest thanks goes to my family, Gloriana, Stefano, Filippo and Patricia, that always supported me and always will, respecting all my decisions and giving me that unconditional love that took me where I am now. Thank you also to all my other relatives, having you all by my side played a big role.

A thanks goes to Turin, I was not able to enjoy it very much the last year but it gave me one of the best experience of my life and wonderful friendships. Thank you to Simone and Francesco, any word cannot describe what you mean to me. Enrico, Ilaria, Giulia and Daniele, you were fundamental for the many unforgettable moments we lived together and I am grateful for those who will continue to share such emotions with me in the future. Thanks to all the UniTo girls, we had very little time to enjoy but it has been incredible and the bond we built is very important to me.

Last but not least, a big thanks goes to all my old friends, that even if we lived in different cities we had our share of fun when possible. They are the real heroes, still bearing me after a very long time.