UNIVERSIDADE FEDERAL DE PERNAMBUCO

Centro de Informática
UFPE

Universidade Federal de Pernambuco
Centro de Informática
Bacharelado em Ciência da Computação

An Analysis of QUIC Use on Cloud Environments

Mário Victor Gomes de Matos Bezerra
mvgmb@cin.ufpe.br

advised by
Prof. Dr. Vinicius Cardoso Garcia
vcg@cin.ufpe.br

Recife
December 06, 2021

Agradecimentos

List of Figures

List of Tables

Contents

Abstract

QUIC is a transport-layer protocol created by Google intended to address some problems of TCP while maintaining compatibility with existing network infrastructure. It has shown to improve user-experience on multiple services and its adoption is increasing everyday.

There is a lot of research showing the improvements of QUIC over TCP/TLS in multiple scenarios, but most of them focus on user-facing applications, generally using HTTP/3. The new version of HTTP runs exclusively over QUIC and it's already supported by most of the browsers.

This document will analyze the use of QUIC for interservice communication in a cloud environment and compare them with more traditional protocols. HTTP/3 will also be compared to its predecessors. Different environments will be used on the experiments to simulate real world production configurations, including high availability setups and use of Kubernetes.

Cost is an important factor when running applications on cloud, compute resources generally are charged by the hour based on the amount of CPU and memory allocated. Therefore, the cost of running applications with QUIC on these environments will also be analyzed.

1 Introduction

¡INSERT¿

1.1 Objectives

¡INSERT¿

1.2 Document Structure

¡INSERT¿

2 Background in Protocols

In this chapter, each observed protocol is briefly analyzed. Their description focus on explaining how they work, which layer they belong to, and their characteristics.

This background serves as a base to understand the motivation during experimentation and better comprehend their results.

2.1 UDP

The User Datagram Protocol (UDP) is a simple message-oriented transport layer protocol that provides a way for application programs to send messages to other programs with a minimum of protocol mechanisms [16]. It has a checksum for data integrity and port numbers for application multiplexing.

Although data integrity can be verified through the checksum, the UDP is considered an unreliable protocol since it offers no guarantee about delivery or order [3]. Furthermore, there is no need to establish a connection between hosts prior to data transmission, making it a connectionless protocol.

Given the lack of reliability, applications that use UDP may come across some packet loss, reordering, errors or duplication issues [3]. These applications must provide any necessary confirmation that the data has been received. Nonetheless, UDP applications usually do not implement any kind of reliability feature, since loss of packets is not usually a problem. Real-time multiplayer games, Voice over Internet Protocol (VoIP), and live streaming are examples of applications that use UDP.

2.2 TCP

In contrast to UDP's unreliability, the Transmission Control Protocol (TCP) is a connection-oriented protocol designed to provide reliable, ordered, and error-checked transmission of data segments that supports multi-network applications [17].

TCP must be able to deal with incorrect, lost, duplicate, or delivered out of order data to provide reliability [17]. This is achieved by assigning a sequence number to each byte transmitted and requiring an Acknowledge (ACK) from the receiver. The sequence number is used to order data that may be received out of order and to eliminate duplicates, and the ACK ensures data has been delivered, otherwise retransmitted after a given timeout. Finally, incorrect data is handled by adding a checksum to each segment transmitted, and discarding any incorrect segments.

To be able to control the amount of data sent by the sender, TCP returns a "window" with every ACK indicating the range of acceptable segments beyond the last successfully received segment [17]. Therefore, the window specifies the number of bytes that the receiver is willing to receive.

Since TCP is connection-oriented, it needs to keep certain status information about the data stream to ensure reliability and flow control. A connection is

defined as a set of information about the transfer, including sockets, sequence numbers, and window sizes. Each connection is distinguished by a pair of sockets between the two hosts. To initiate a connection, the TCP uses a three-way handshake (Figure 1) to establish a reliable connection [20].

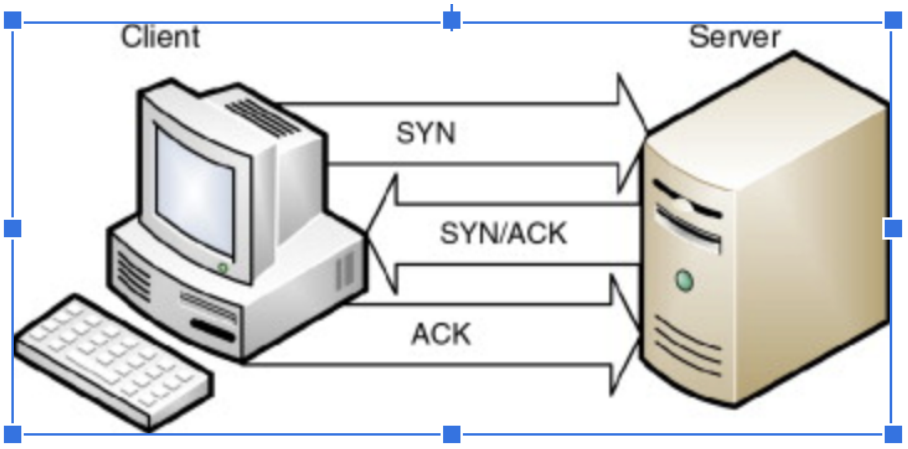

Figure 1: TCP's Three-Way Handshake
Source: ADD SOURCE

The sender chooses the initial sequence number, informed in the first Synchronize (SYN) packet. The receiver also chooses its own initial sequence number by informing in the Synchronize/Acknowledge (SYN/ACK) sent to the sender. Each side acknowledges each other's sequence number by incrementing it, allowing both sides to detect missing or out-of-order segments [20].

TCP is stream oriented, effectively meaning it has the ability to send or receive a stream of bytes [17]. It can bundle up data that comes from the application layer and send it in a single segment, being responsible for breaking the stream of data into segments, and reassembling once they reach the other side. UDP, on the contrary, is a message-oriented protocol where the division of data into user datagrams is made by the application [3].

To be able to offer more functionality, TCP ends up sacrificing efficiency. While in the case of a connectionless and unreliable protocol such as UDP, it turns out to be faster due to the lack of any extra features [19].

2.3 TLS

As more people have access to the Internet, the higher the requirement is for it to be secure. Data generated by users can have many harmful implications since it is directly related to privacy. While offering reliability, which is a necessary attribute to Internet's communication, TCP's communication is not encrypted. Therefore, anyone with a minimum knowledge of networking can see everything that's traversing the network, breaking with users' privacy.

The Transport Layer Security (TLS) protocol is a cryptographic protocol designed to provide privacy and data integrity between two communicating applications [13]. The connection is considered private since symmetric cryptography is used for data encryption and every connection has an unique generated key negotiated between parties.

Peers' identity can be authenticated through the use of asymmetric cryptography [1]. This is important because both sender and receiver can establish a trusted relationship by verifying if their certificates and public IDs are valid and have been issued by a certificate authority (CA) listed in their respective list of trusted CAs.

To be able to establish a TLS connection, the sender and receiver must negotiate a connection by doing a TLS handshake. It allows hosts to authenticate with each other and to negotiate a cipher and generate session keys in order to use symmetric encryption before the application protocol actually starts to transmit data.

TLS can be used to encrypt TCP connections by including the TLS handshake to TCP's connection establishment flow (Figure 2). This adds an overhead by increasing the number of Round-Trip Times (RTT) needed before the actual data transmission starts.

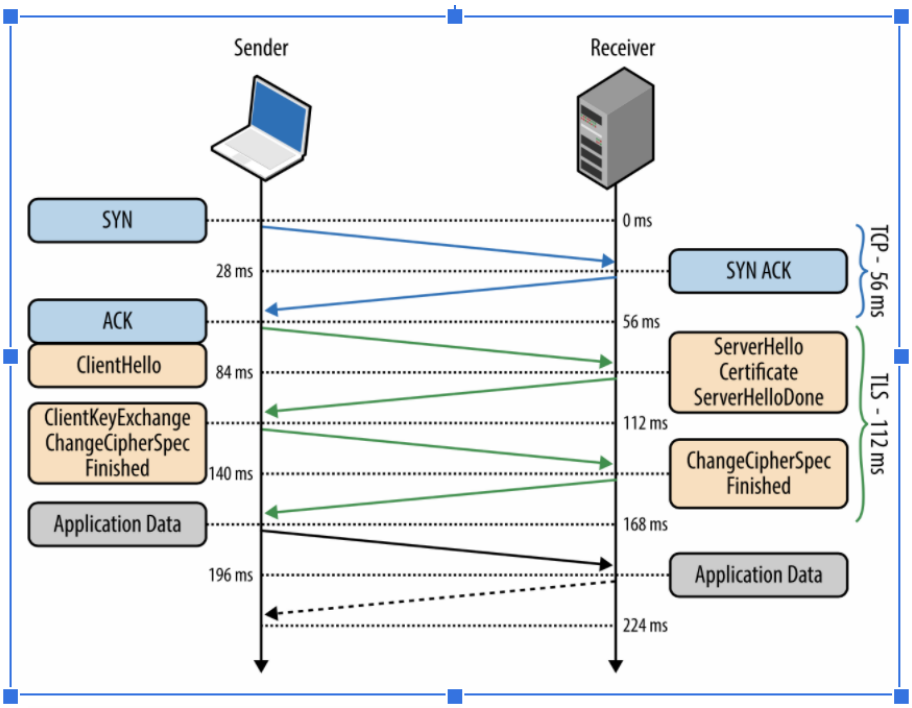

Figure 2: TLS Handshake
Source: ADD SOURCE

Working alongside one another, TCP and TLS have kept the Internet reliable and encrypted. However, they only provide the channel of communication, how applications communicate is determined by the application-layer protocols.

2.4 HTTP

The Hyper Transfer Protocol (HTTP) is considered the foundation of the World Wide Web. It is an application-level protocol for distributed, collaborative, hypermedia information systems that runs on top of the other layers in the network protocol stack [8, 9, 22].

It is a generic, stateless protocol which uses a predefined set of standards and rules for exchange of information through a request-response process [8, 9]. The sender submits an HTTP request message to the receiver, which provides resources, such as HyperText Markup Language (HTML), or performs some action on behalf of the sender. The receiver then returns a response message to the sender, it contains status information about the request and possibly the requested data in the message body, enabling the sender to react in a proper way, either by moving on to another task or handling an error [9].

The following subsections introduce HTTP versions, what they improved and a general feeling of how they work.

2.5 HTTP/1

The first version of HTTP, known as HTTP/0.9, was a simple protocol for raw data transfer across the Internet [9]. It only had the GET request method, equivalent to a request to retrieve data from the server. Message types were limited to text, and it had no HTTP headers, meaning there was no metadata, such as status or error codes, on the request/response.

HTTP/1.0 improved the protocol by allowing messages to support the Multipurpose Internet Mail Extensions (MIME) standard, which extends data types supported by messages, being possible to send videos, audio, and images [9]. In addition, metadata was also incorporated into requests and responses, increasing the available request methods, while improving error handling by the use of status and error codes.

HTTP/1.1 is considered a milestone in the evolution of the Internet, since it eliminates a lot of problems from previous versions and introduces a series of optimizations [22]. Connections can be reused in favour of having to create a connection to every request and can be pipelined, improving the time needed to perform multiple requests. It also provides support for chunked transfer, which is a streaming data transfer mechanism that divides the data stream into "chunks" that are sent out independently of each other. This allowed for a more efficient transfer of large amounts of data due to concurrency.

2.6 HTTP/2

HTTP/2 purpose is to optimize transport for HTTP semantics, while keeping the support for all of the core features of the HTTP/1.1.

Even though HTTP/1.1 added pipelined connections, it still suffers from the Head of Line blocking (HOL blocking) problem [15]. It happens when the number of allowed parallel requests is used up, and subsequent requests need to wait for the former ones to complete. HTTP/2 solves this by implementing multiplexing. This is achieved by having HTTP request/response associated with its own stream and since streams are independent of each other, a blocked request/response does not prevent progress on other streams.

HTTP/2 adds a new interaction mode where a receiver can push responses to the sender [15]. This resulted in senders not needing to send periodical requests for new data to the server by using polling methods, trading network usage for some improvement in latency.

Because HTTP header can contain a lot of redundant data, HTTP/2 uses a compressed binary representation of metadata instead of a textual one, this reduces the space required [15].

2.7 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is an extension of HTTP, conceptually equivalent to HTTP over TLS [10]. HTTPS is when the HTTP client also acts as a TLS client, it should perform all TLS requirements, such as establishing a connection through a TLS handshake. Once it finishes, the client may initiate the first HTTP request, the difference being all data will be encrypted. HTTPS maintains HTTP behaviour while providing TLS features for instance encryption, data integrity, and authentication [10].

2.8 QUIC

HTTPS provides a reliable and secure connection and is considered the main application-layer protocol used by the World Wide Web. However, it has some limitations due to requiring the use of TLS for security and TCP for reliability.

QUIC is a secure general-purpose transport protocol designed to improve performance for HTTPS traffic and to enable rapid deployment and continued evolution of transport mechanisms [7]. It replaces most of the traditional HTTPS stack: HTTP/2. TLS, and TCP (Figure 3).

As the Internet evolves, software requires fast deployment of changes both to improve and to secure it. TCP is a kernel-space transport protocol, meaning that any vulnerabilities or improvements require an upgrade to the Operating System (OS) kernel. Since such changes have a huge impact on the entire system, they must be made with caution and may take years to become widely spread [18]. QUIC was developed as a user-space transport protocol on top of UDP, facilitating its deployment as part of other applications, resulting in meaningful impact in a relatively short time [7].

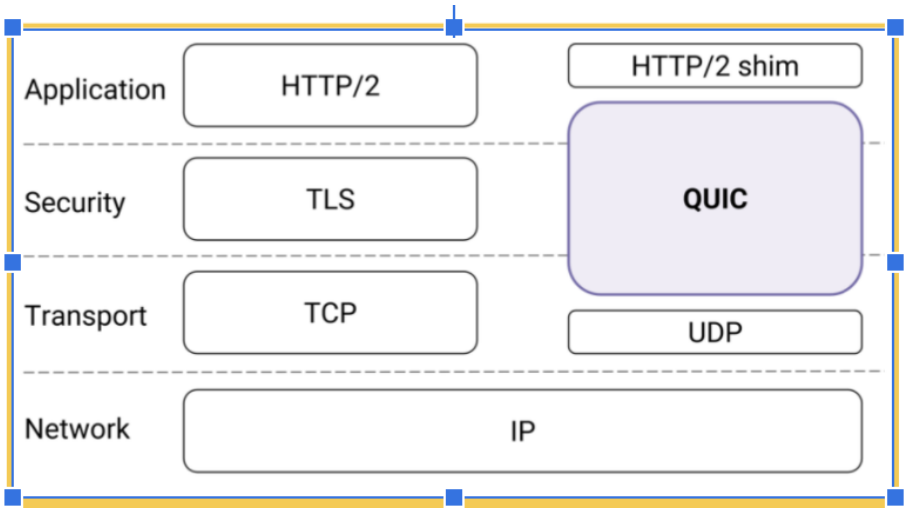

Figure 3: HTTPS vs QUIC Layers
Source: ADD SOURCE

A middlebox is when a device adds functionality other than packet forwarding to an IP router, such as filtering, altering, and manipulating traffic [12]. Improvements to transport protocols is reduced due to the fact that these kinds of devices are hard to be removed or upgraded, creating a dependency between them. For example, firewalls tend to block any unknown traffic for security reasons, meaning that new transport protocols need to be explicitly supported [12]. QUIC addresses this issue by encrypting its packets, therefore avoiding middlebox dependency and data tampering [7].

glshttp/2 solves the HOL blocking problem in the application layer by introducing request multiplexing, however it still suffers from this problem in the transport layer. Even though HTTP/2 streams are independent of each other, they still had to share a TCP connection and deal with TCP's window size. If the first window segment fails, the window cannot go further and blocks new segments from being transmitted. QUIC solves this problem by implementing stream multiplexing. More than one stream within a connection means that even if one stream drops packets, other streams will not be blocked in any way.

HTTPS is required to perform both TCP and TLS handshakes to establish a secure connection, resulting in generally a 3-RTT connection setup (Figure 4) to be able to start transmitting the actual data [14].

QUIC improves the handshake delay by minimizing the steps required to establish a connection, being able to perform a 0-RTT connection setup once the server is known (Figure 5) [7, 18]. It does this by caching information about the server on the client after a successful connection. If it tries to set up a connection with expired information, the server sends a reject message that contains all the information necessary to establish a connection.

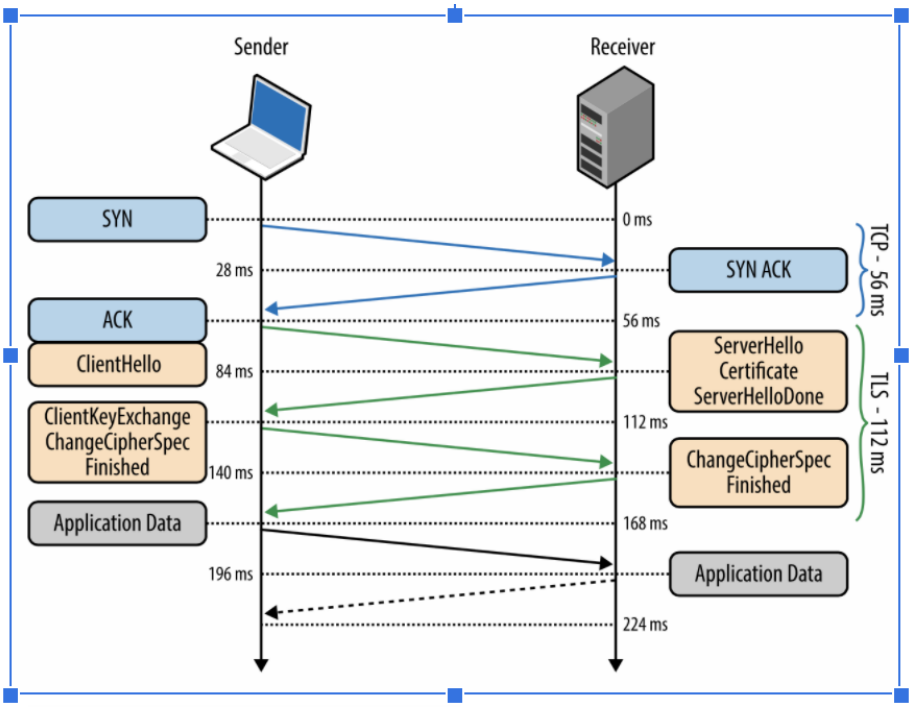

Figure 4: TLS Handshake
Source: ADD SOURCE

TCP loss recovery mechanisms are able to detect when a segment is probably dropped, triggering its retransmission. This process uses RTT estimation to improve its efficiency, which in the case of TCP, is based on TCP's sequence numbers. However, once a retransmission is made, there is no way of knowing if the ACK received is for the original or the retransmitted segment since both segments use the same sequence number. Additionally, dropped retransmission segments are usually detected by the use of timeouts, further slowing TCP [7, 11, 18, 23].

QUIC eliminates TCP's retransmission ambiguity problem by including a new packet number to all packets, even those including retransmitted data [18]. This means it can measure RTTs precisely since it always knows which packet each ACK refers to. To maintain the data order, QUIC adds a stream offset to the stream frames present in the packet, decoupling the need of ordered packet numbers [7].

While QUIC improves transport efficiency, it demands more computational resources when compared to TCP+TLS. The focus during its design was improving transport, not resource efficiency, resulting in an initial raise by 3.5 times of Central Processing Unit (CPU) utilization when compared to TCP+TLS traffic [7]. Optimizations were made after this assessment, decreasing CPU usage

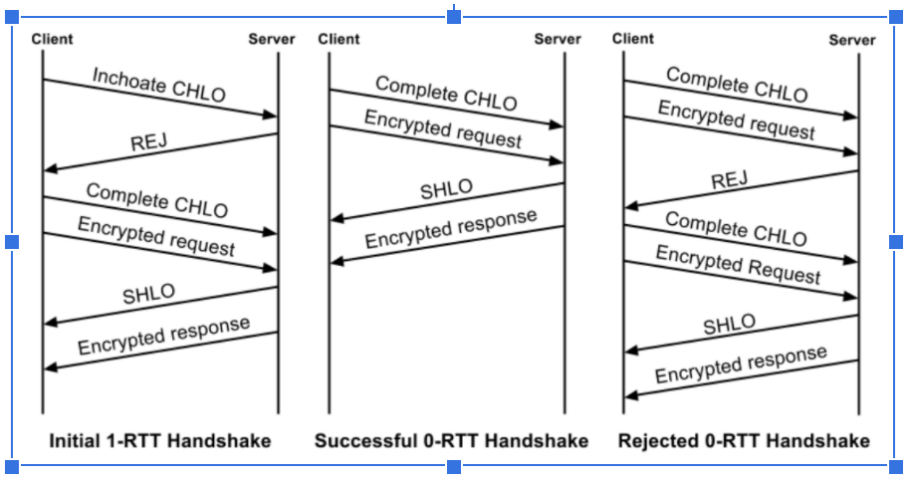

Figure 5: QUIC Handshake
Source: ADD SOURCE

difference to 2 times TCP+TLS'.

2.9 HTTP/3

HTTP/3 is effectively HTTP over QUIC. It provides a transport for glshttp semantics using QUIC as transport protocol, therefore it maintains all the same request methods, status codes, and messages fields. It still does not have an official Request for Comments (RFC), however it possesses an initial draft [32]. Nonetheless, it is supported by 74

2.10 Summary

¡INSERT¿

3 Background in Distributed Systems on Cloud

The previous section introduced the QUIC protocol and all protocols that it's either trying to improve, or their previous versions. Their differences and motivations were defined and will be used to explain the results of the experiments described further ahead.

In order to understand the experiment's motivations, this section provides an overview of the challenges to build distributed applications on cloud and what they offer to be used in production environments by many large scale companies.

3.1 Distributed Systems

Back in the day, the most common way to solve a computing problem was to write a single application that was responsible for performing some kind of task. These applications are called monolithic applications.

Once the application grows to a point it demands scaling, it will demand more replicas of itself to be able to handle more traffic. Though this might not be a problem, since all functionality was incorporated into a single application, in most cases, only a few parts will actually require scaling. Consequently, resulting in a waste of resources.

Distributed systems come into the scene to try solving these problems. It makes use of multiple computing devices spread over a network and have them coordinate efforts to perform some kind of task [4]. It allows complex applications to be broken down into manageable chunks, each performing a different kind of functionality.

Its components can be broken down into two categories of clients: persistent and ephemeral.

An ephemeral client creates a single connection to the server, performs some sort of exchange of data, and then closes the connection once it's finished. A common example is that of a job, a finite task that has to perform a large computation, batch-oriented tasks, or creating a database backup [6].

A persistent client, however, also establishes a single connection to the server, but it maintains it open throughout its entire lifetime, making requests when needed. This can be represented by a service that needs to communicate with a database to provide some kind of functionality.

By defining these components and how they work, simulations can be performed to see how they perform on different scenarios. This allows for speculation about their behaviour in a real-world scenario.

3.2 Cloud

The term "cloud" was used to refer to platforms for distributed computing as early as 1993 [21]. Cloud computing the delivery of computing services, such as databases and storage, over the Internet. It can be divided into three main categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and

Software as a Service (SaaS). From here on, the focus will be the IaaS cloud computing category.

3.2.1 IaaS & Containerization

IaaS is a form of cloud computing that has become one of the most common ways to provision on-demand availability of infrastructure services without having to be directly managed by the user.

Users are not required to have their own data centers to be able to serve their applications, they can request resources to use a third party's computing infrastructure using a pay-as-you-go method. Therefore, they only need to pay for the resources actually used, enabling them to easily scale when needed.

Multiregional servers are also possible. Users are able to request resources in distinct regions to deal with traffic from different countries or due to some contractual requirements. Some providers even have more than one data center per region, capaciting users to build high availability systems that are able to deal with disaster scenarios, for instance the data center experiences a power outage.

As data centers machines processing power and capacity increased over the years, many resources never got to be used by the applications whereas their requirements were much lower. This causes a waste of computing resources that could otherwise be used by users, increasing providers' revenue. Thus, Virtual Machines (VMs) came into existence.

VMs consist of the virtualization of an OS, allowing one single physical host machine to have more than one virtual machine running at the same time while keeping them isolated from each other. This enables cloud providers to optimize the use of their resources by serving multiple VMs while only having to use one physical host machine, consequently serving more than one user per machine.

Nevertheless, VMs improve data centers resource efficiency, it can take up a lot of system resources. Each VM requires a copy of an OS and a virtual copy of all the hardware that the OS needs to run, adding up to a lot of memory and CPU. This is still more efficient than running separate physical host machines, but can be a deal-breaker for applications.

To improve application development, deployment and flexibility, containers were created. Instead of virtualizing the entire host machine, containers use linux namespaces to run on isolated environments while sharing the underlying OS (Figure 6), resulting in less requirements since they only need to package the actual application and all files necessary to run. In addition, their lightweight characteristic allows faster startup time, while VMs may take minutes to be provisioned, most containers are ready in a few milliseconds.

Developers usually write code locally possibly using their laptop, and then this code will be deployed on the server. Any differences between these two environments, laptop and server, can cause unexpected bugs. Containers solve the problem of environment inconsistency. Developers can work on a local environment that contains all of the dependencies of any environment whether it's development, testing, or production.

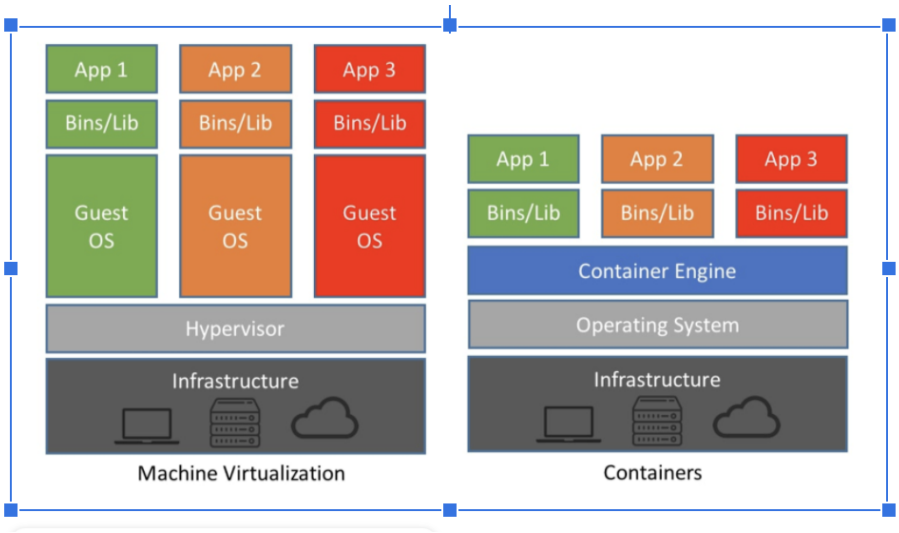

Figure 6: VM vs Container Architecture
Source: ADD SOURCE

3.2.2 Kubernetes

Dealing with distributed systems over the cloud have given companies the power to build complex and intelligent solutions that are able to solve really hard problems that otherwise would require a lot of investment and time. However, managing such systems comes with its own challenges.

Due to containers being lightweight and ephemeral, running them in a production environment can become overwhelming. A containerized application might need to operate hundreds to thousands of containers. A container orchestrator is the automation of operational effort required to run such containerized applications.

According to the Cloud Native Computing Foundation (CNCF), Kubernetes (K8s) is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications [5]. Additionally, it's considered the most widely used container orchestration platform by CNCF's K8s Project Journey Report from 2019 [2].

The fundamental premise behind K8s is that it enforced desired state management. Consequently, a K8s cluster is fed with specific configuration files, called manifests, and it's up to the cluster to provide the necessary infrastructure to be able to meet the desired state.

K8s' smallest and most basic deployable object is called pod. It consists of a wrapper around containers that has computing resources. One or more containers can be inside a single pod, consequently sharing the pod's resources with each other.

In the context of containers, a pod is a set of Linux namespaces and the

same things that are used to isolate a container. Therefore, in the context of K8s, it's common to refer to pods when talking about scaling and deployment.

Each cloud provider often provides a managed way of running K8s, they take care of the control plane, component responsible for managing the worker nodes and the pods, management while enabling users to actually use the cluster to their needs. For instance, Amazon Web Services (AWS) offers the Elastic Kubernetes Service (EKS).

Some cloud providers allow applications to be deployed in different regions. In the case of AWS, it even allows you to choose a data center within a region, defined as Availability Zone (AZ). Hence, there can be multiple AZs within a single region.

Within the K8s it's possible to have pods deployed to different regions or AZs based on their affinity and tolerations by tainting a node with a label. This allows the user to have full control over where each application is going to run.

3.3 Summary

¡INSERT¿

4 Experiments Setup

This section explains the reasons for each experiment's characteristics, why each scenario is considered, what they bring to the table, and what kind of metrics are going to be collected.

UDP, TCP, TCP+TLS and QUIC transport-layer protocols will be compared. In addition, HTTP/1, HTTP/1+TLS, HTTP/2, HTTP/2+TLS and HTTP/3 application-layer protocols will also be compared.

Ephemeral and persistent clients are going to be used, each running in three scenarios:

- Multi-AZ, different nodes in different AZs
- Same-AZ, different nodes in the same AZ
- Local, same node

Persistent clients are used in two kinds of experiments: sequential and concurrent. The former simulates clients that are able to maintain a connection, but are only able to send one request at a time in a sequential manner. The concurrent experiments, however, simulates most services, which are able to maintain a connection and perform multiple requests simultaneously. These connections are limited to a maximum of 100 simultaneous requests.

All experiments will be performed in a K8s Cluster deployed on AWS using EKS. Metrics collected will be:

- Latency
- Throughput
- CPU
- Memory

All experiments are executed using a simple demo application that implements every protocol in the most efficient and similar way. It is a simple application since all it does is establish a connection with the server, send a fixed amount of data, wait and receive its response, and, depending on the type of client, either close the connection or keep sending requests.

The fool

4.1 About Protocols

Since QUIC is a transport-layer protocol, it makes sense to compare it with other protocols from the same layer. This enables observation on how they perform under the same circumstances.

QUIC replaces most of the HTTPS stack: HTTP/2, TLS, and TCP. Therefore, it will be compared against UDP, TCP, and TCP+TLS transport protocols.. By making this comparison only in the transport layer, there will be less variables to consider when analysing the results.

Since QUIC uses UDP as part of its implementation, it is also considered, as it might show how effectively it's used by QUIC.

UDP is used in its pure implementation, therefore no additional logic is incorporated to be able to grasp its performance. This means there might be scenarios in which it will not be able to successfully finish experiments due to its unreliable nature.

In addition, HTTP/3, also known as HTTP over QUIC, will be compared against its predecessors: HTTP/1 and HTTP/2. Comparing application-layer protocols enables us to observe QUIC's impact on a real-world situation, since applications will not use QUIC directly, but through HTTP. Furthermore, QUIC also performs TLS' role of securing HTTP's traffic, therefore HTTP/1+TLS and HTTP/2+TLS protocols will also be tested.

4.2 About Clients

By separating clients into persistent and ephemeral categories, experiments are able to simulate two very common scenarios on distributed systems and observe how QUIC behaves when having to deal with each one of them.

The first scenario explores ephemeral connections, representing when we have a job that needs to perform some sort of finite task, creating a single connection to the server, and closing it once it's done.

The second scenario explores persistent connections, representing when we have a service that needs to communicate with a database to provide some kind of functionality, creating a single connection to the database throughout its entire lifetime.

4.3 About Kubernetes

All experiments are made inside a K8s Cluster since it is a production-grade container orchestrator, a common production environment used by most companies that have to deal with container management in the cloud.

By choosing to use this environment, it not only reflects an actual production scenario, but also allows easy networking setup for nodes running on different AZs. Running nodes on different AZs is a requirement due to high availability, as every data center is always under the danger of downtime due to a natural disaster or to some event that damages some of the datacenter's infrastructure. Therefore, Multi-AZ is a must have for companies that require a disaster recovery plan to be able to deal with such catastrophic events.

During the experiments, three scenarios are taken into account: when the client and server are running in the same node, when they are running in different nodes while in the same AZ, and when they are running in different nodes while in different AZs. These are all possible scenarios when using K8s since client and server applications may be running in any node, depending on the configuration.

To be able to control which nodes were going to be used by each application during experiments, pod affinity and taint were configured to be able either

force a pod to be scheduled on required nodes.

During the local scenario, pod taint was used to make pods to be scheduled to a node in the same AZ, and pod anti-affinity and affinity were used to make client and server pods to be scheduled to the same node. During the different nodes in the Single-AZ scenario, pod taint was also used to schedule pods to nodes in the same AZ, but only pod anti-affinity was used to make sure there was only one pod running in each node. The last scenario required pods to be running on nodes in different AZs, pod taint was used to schedule pods to different AZs, while pod-affinity was also used to make sure there was only one pod running in each node.

4.4 About Metrics

QUIC has an increased usage in memory and CPU since the focus during development was performance and not efficiency, resulting in an initial raise by 3.5 times of CPU utilization when compared to TCP+TLS traffic [14]. Optimizations were made after this assessment, decreasing the CPU usage to 2 times TCP+TLS', however it's still believed that, even after more optimizations, this increased cost will always exist.

Latency and throughput are also analyzed to be able to grasp a better understanding on how performatic each protocol is in relation to how fast it can respond to requests and exactly how much data can be transmitted every second.

4.5 About Demo Application

Developed in Go, this demo application aims to implement every protocol used during these experiments in the most efficient and similar way.

Each protocol must implement an ephemeral client, a persistent client and a server.

The ephemeral client's job is to establish a connection with the server, send a single request, wait for its response and close the connection, this can be performed multiple times, always resulting in a closed connection after a single request.

The persistent client's job is similar to the former, the only difference being multiple requests can be made within a single connection. Two kinds of experiments are performed with this type of client: sequential and concurrent. The former simulates clients that do not support concurrent requests, therefore is only able to send one request at a time. The concurrent experiment, otherwise, simulates services that are able to perform multiple requests at the same time. Microsoft states most gRPC servers set concurrent requests limit to 100 by default [27]. Therefore, this experiment set the same limit to be able to simulate a real-world scenario.

The server's job is to receive a request, and send a response with a fixed amount of data in it.

The size of data sent within the requests and responses is predetermined in compilation time. This value varies in the following manner: 2KiB, 8KiB, 32KiB, 128KiB, and 512KiB. These values were chosen due to the fact that they are a reasonable size of data transfer amongst services running in the cloud. Kafka's maximum package size's default value is 1MB, because packages with more than 10MB can affect the performance of the cluster [25]. gRPC limits incoming messages size to 4MB to help prevent it from consuming excessive resources [26].

4.6 About AWS

AWS is one of the most popular cloud providers, offering a wide range of services. It was used as a provider due to its accessible price and since it met all experiments requirements. For instance, it allows use of AWS Elastic Cloud Computing (EC2), AWS' equivalent to VM, to perform experiments.

Pricing was also taken into account when preparing for experiments. AWS not only charges for the EC2 instance and EKS, but also for data transfer between AZs, resulting in a hefty cost when exchanging terabytes of data.

During each experiment, 10000 requests are made. Therefore, one of the Multi-AZ experiments, transferring requests and responses with 512KiB of data of a single protocol, transfers a total of 9.77 GiB of data. AWS charges $0.02 per GiB transferred between AZs, resulting in an approximate cost of $0.20 per experiment. With 9 protocols, ephemeral client, and persistent client with sequential and concurrent scenarios, all Multi-AZ experiments with 512KiB of data costs approximately $5.28.

AZs utilized during experiments were us-east-1 (use1-az1) and us-east-1b (use1-az2). The latter was only used during Multi-AZ testing, while other experiments only used use1-az1.

The EC2 instance type used during experiments was m6i.xlarge. Initial experiments were performed with the smallest instance m6i.large and CPU limit on K8s was set to 1 CPU per pod. This, however, resulted in interfering with QUIC results since this protocol was suffering throttled. To avoid throttling, this CPU limit was removed and EC2 instance type was changed to xlarge, which contains 2 CPUs instead of 1, assuring the pod will have all computational power it needs to perform as intended.

4.7 Summary

¡INSERT¿

5 Transport Protocols Experiments

5.1 Latency & Throughput

Charts on Figures 8, 9, 10, 11, 12, 13 represent the 90th Percentile (P90) of Latency and Throughput of all 10000 requests made by the persistent and ephemeral clients during the experiments.

Throughout these charts it is possible to observe an order pattern. Local experiments usually were more performatic when compared to Single-AZ experiments. Additionally, the latter results were better than Multi-AZ experiments due to the overhead from sending the datagram to another data center, even though remaining in the same region.

UDP Multi-AZ and Single-AZ experiments only succeeded on the first two iterations of each run when the data transferred was 2KiB and 8KiB. This was expected since pure UDP does not have any reliability and is expected to lose user datagrams along the way. As was stated before, no additional logic was added to UDP since this would interfere with the results because it would alter UDP natural behaviour.

Local TCP experiment was by far the most efficient, with almost zero latency when connection was persistent, reaching approximately 19Gb/s of speed when transferring 512KiB of data per request. It outperformed UDP, that even though is considered faster due to its unreliability and connectionless characteristics, it only reached 2.8Gb/s in the same scenario. This happened because, while TCP is stream oriented, which enables it to send a continuous stream of data with no overhead, UDP is message oriented.

Nonetheless, ephemeral clients demonstrated that establishing connections can have a significant impact on TCP's performance. Local TCP latency went from 0.19ms to 1.79ms and throughput from 19Gb/s to 2.2Gb/s. Both results are still better than other TCP and all TCP+TLS' experiments P90, but brings them to a similar level.

It is possible to observe TLS encryption overhead on TCP during persistent clients experiments. Furthermore, during ephemeral clients this overhead increases due to having to perform a TLS handshake before every request.

QUIC's experiments were the worst, with almost 7 times slower than TCP+TLS' Multi-AZ experiment (with persistent client) latency and throughput. This demonstrates TCP+TLS performs better than QUIC on a reliable network, with a low packet loss rate. However, QUIC performs better than TCP+TLS on an unreliable network.

QUIC's specifically designed to be used in environments such as a wireless network or a mobile device, both which have a high tendency to lose packets, and to break TCP connections. QUIC solves these problems by implementing a more efficient way to deal with retransmission of lost packets, improving its throughput, and having a 0-RTT handshake, which deals with broken connections by removing the need to perform a TCP and TLS handshake.

Consequently, QUIC's results show it's not meant to be used on distributed systems. These environments are usually part of a reliable network, meaning that TCP is usually a better fit.

5.2 Parallel

All previous experiments performed requests sequentially, undermining the possible gain in efficiency of protocols that contain improvements to concurrent requests. Therefore this experiment tries to explore this scenario by performing all 10000 requests with 100 Goroutines, each performing 100 requests.

However, when packet size is the same, we fallback to a scenario where multiplexing is useless since any request performed in sequence is going to take more than the previous to complete. Therefore, this is similar to a pipelining scenario, where the requests finish in the order they were made.

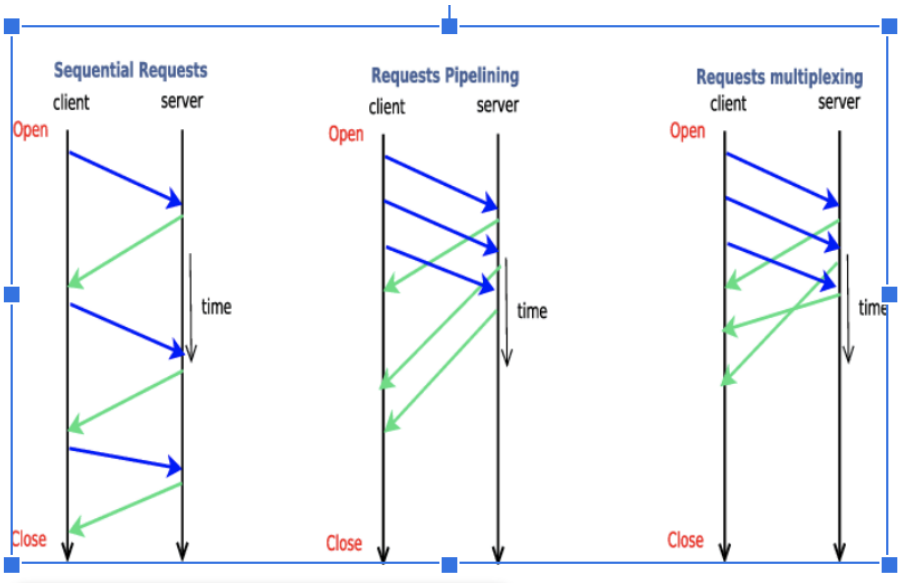

Figure 7: Pipelining

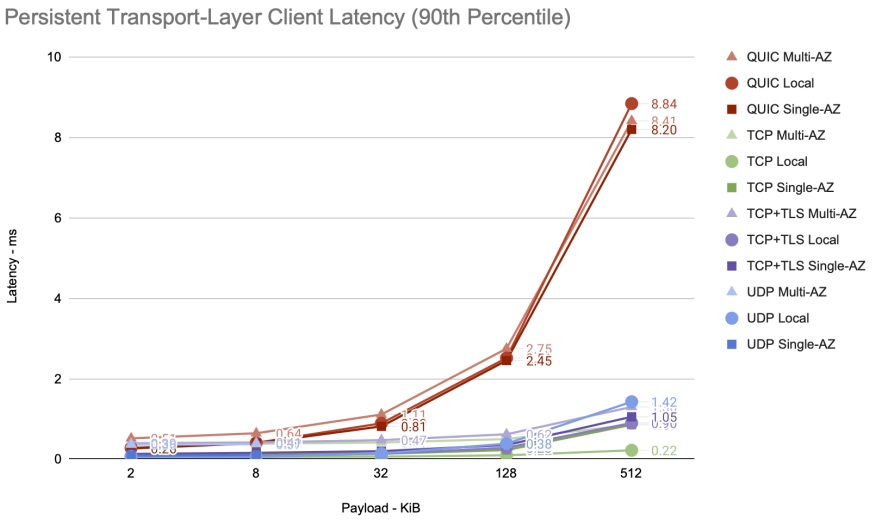

Figure 8: Persistent Transport-Layer Client Latency (90th Percentile)

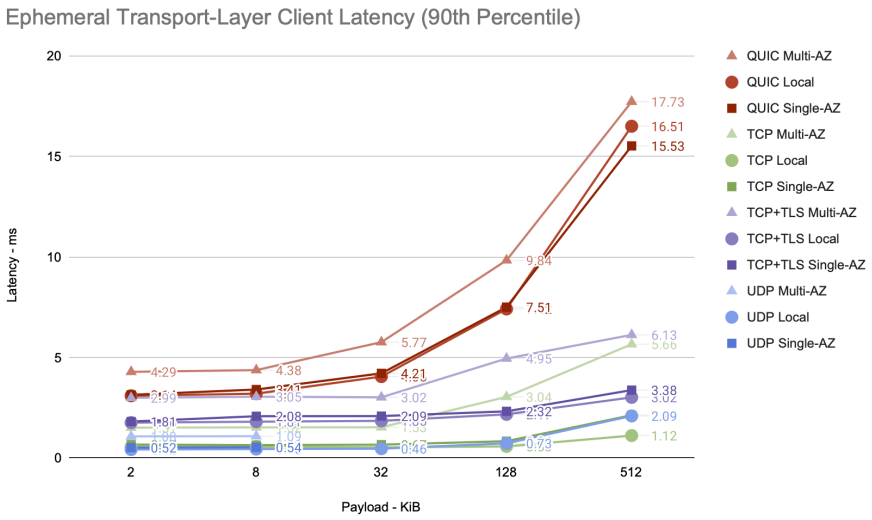

Figure 9: Ephemeral Transport-Layer Client Latency (90th Percentile)

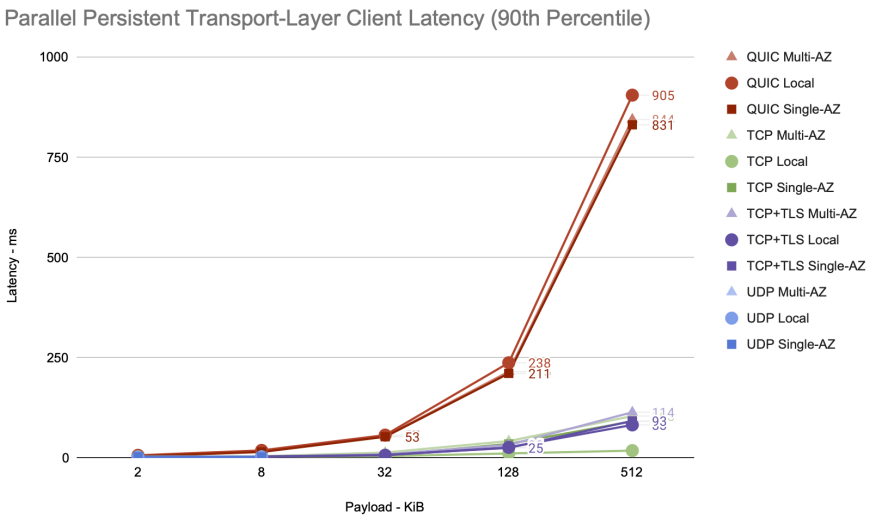

Figure 10: Parallel Persistent Transport-Layer Client Latency (90th Percentile)

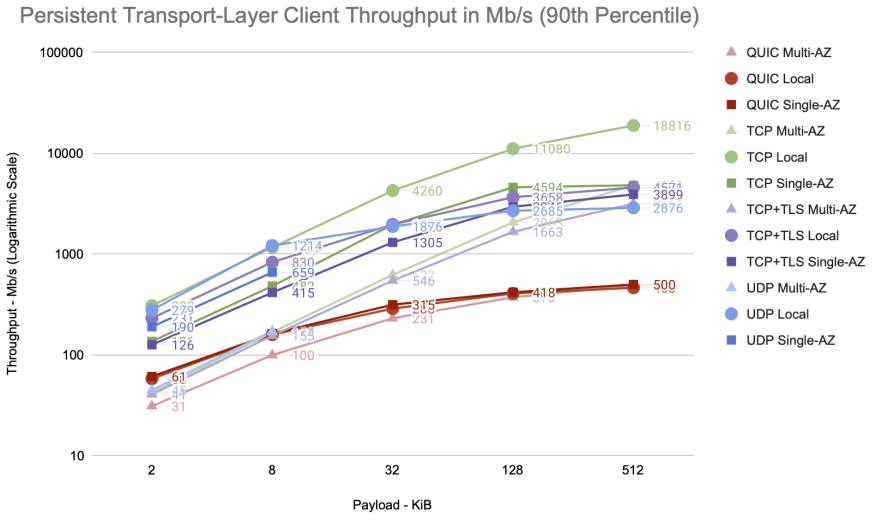

Figure 11: Persistent Transport-Layer Client Throughput in Mb/s (90th Percentile)

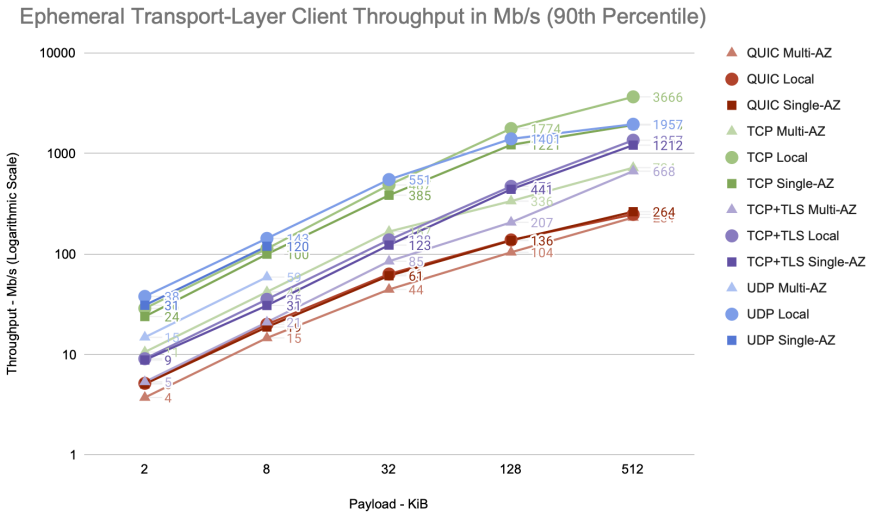

Figure 12: Ephemeral Transport-Layer Client Throughput in Mb/s (90th Percentile)

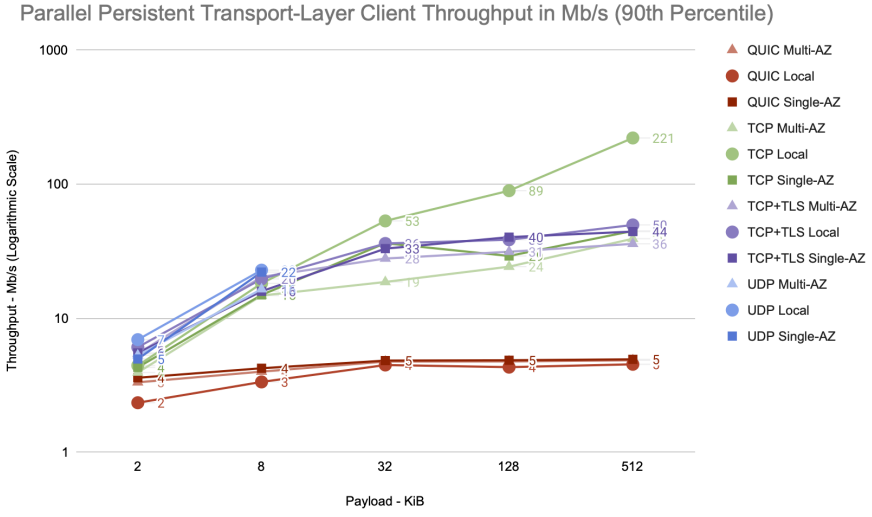

Figure 13: Ephemeral Transport-Layer Client Throughput in Mb/s (90th Percentile)

5.3 CPU

These charts represent the CPU usage of clients and servers during ephemeral and persistent experiments.

CPU usage is basically mirrored between client and server. This happens due to the fact that client requests and server responses send the same amount of data, resulting in the same CPU usage.

During the first two packet sizes, UDP is the most efficient overall. However, from 32KiB onward, it increases CPU usage and surpasses TCP costs.

In the case of the other protocols, TCP is the most efficient. TCP+TLS has to deal with the extra TLS overhead, having to encrypt data and performing TLS handshakes to establish a connection. Finally, QUIC has an increased usage of CPU as a tradeoff to efficiency, since it has to deal with cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state.

CPU also increases between experiment types. Local experiments cost less than Single-AZ, which costs less than Multi-AZ. This happens due to the increased latency each scenario adds to each experiment, demanding most CPU time.

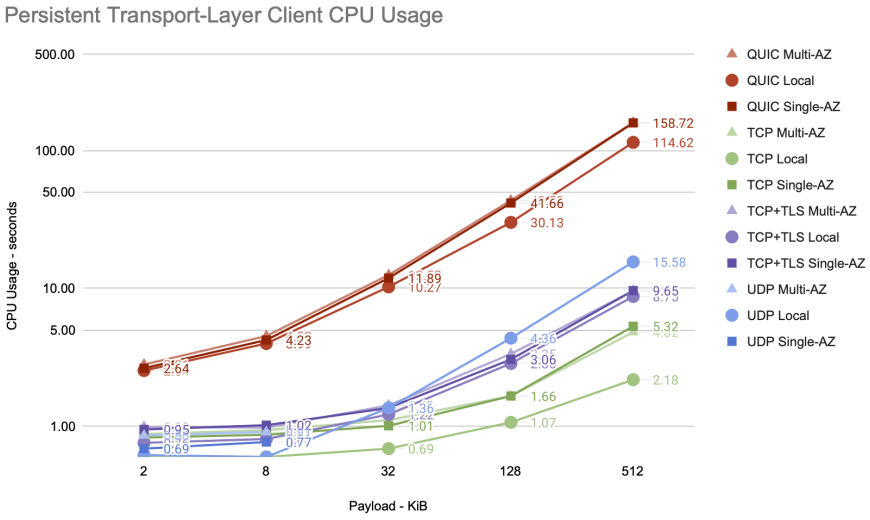

Figure 14: Persistent Transport-Layer Client CPU Usage

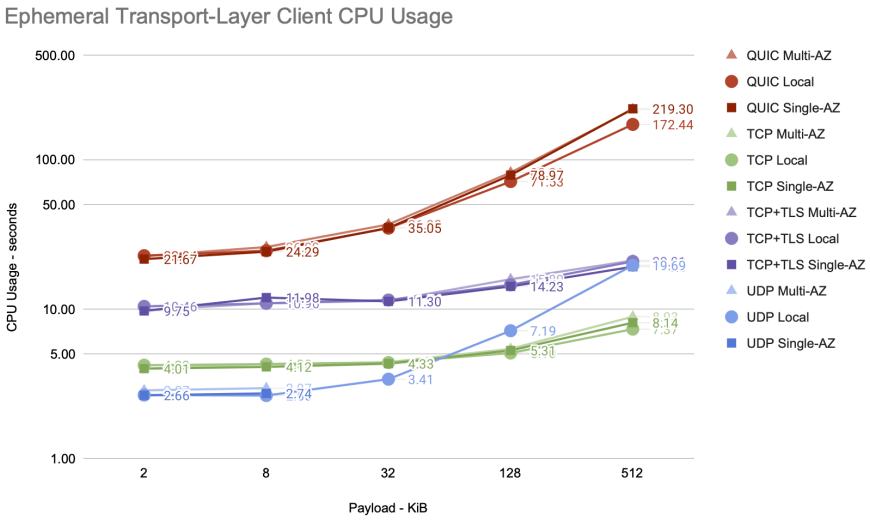

Figure 15: Ephemeral Transport-Layer Client CPU Usage

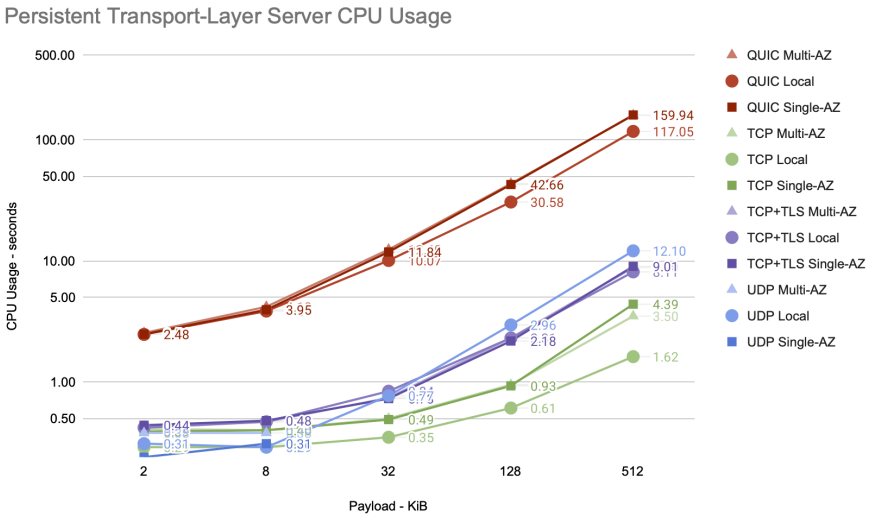

Figure 16: Persistent Transport-Layer Server CPU Usage

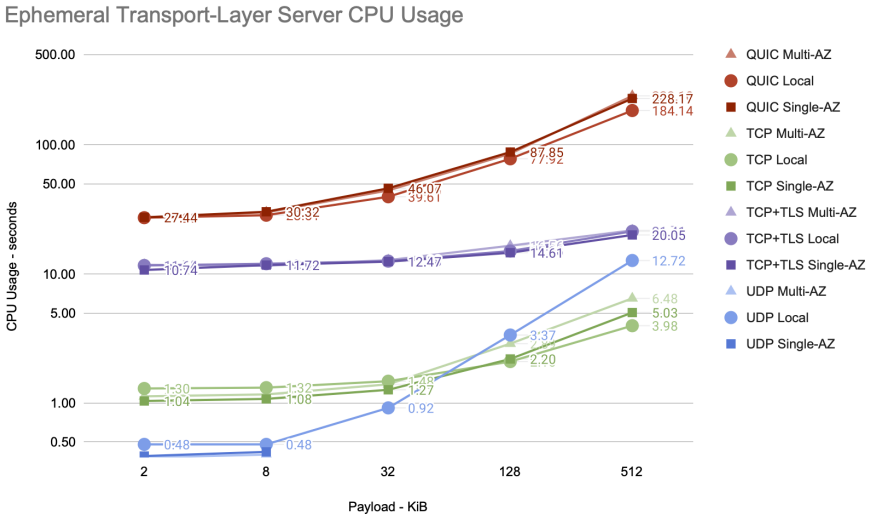

Figure 17: Ephemeral Transport-Layer Server CPU Usage

5.4 Memory

These charts represent the memory usage of clients and servers during ephemeral and persistent experiments.

During persistent and ephemeral experiments, it's possible to observe that UDP and TCP had the lowest memory usage. However, TCP+TLS was more than double. That can be explained by TLS' data encryption overhead, which demands a bit more memory usage.

QUIC uses around double memory when compared to TCP+TLS experiments. Nothing special since it's expected to have more memory usage as a tradeoff for efficiency on unreliable networks. Nonetheless, on ephemeral experiments it had an interesting behaviour. On smaller data sizes it uses a ton of memory, around 10 times TCP+TLS', while only using half on the largest data size. This happens during ephemeral experimentations only because we are constantly creating and deleting clients, therefore Go's garbage collector is unable to quickly clean unused clients. On smaller data sizes, the rate of client creation is higher because the requests end faster, in contrast to big data sizes, which require more time to complete requests, allowing Go's garbage collector more time to do its job.

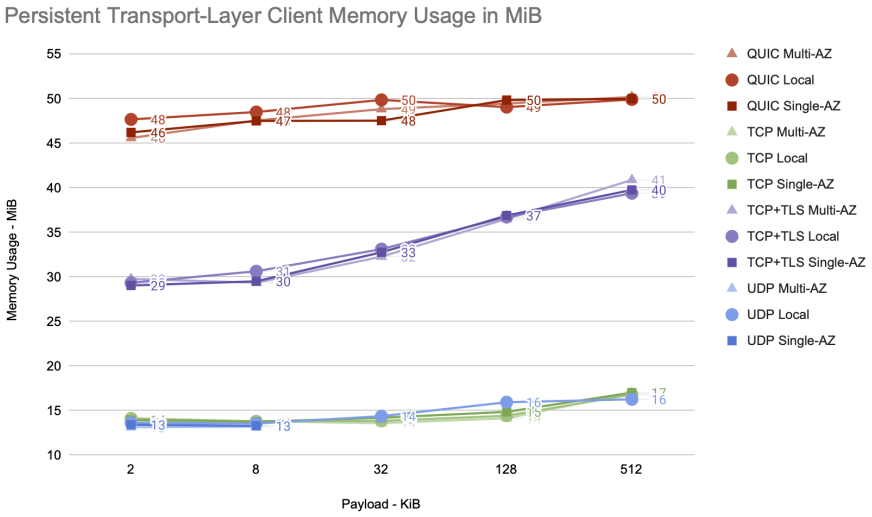

Figure 18: Persistent Transport-Layer Client Memory Usage in MiB

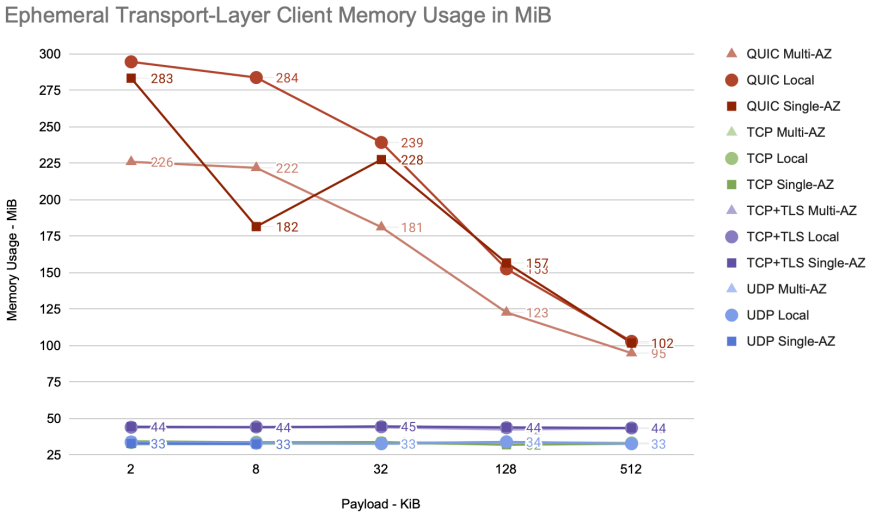

Figure 19: Ephemeral Transport-Layer Client Memory Usage in MiB

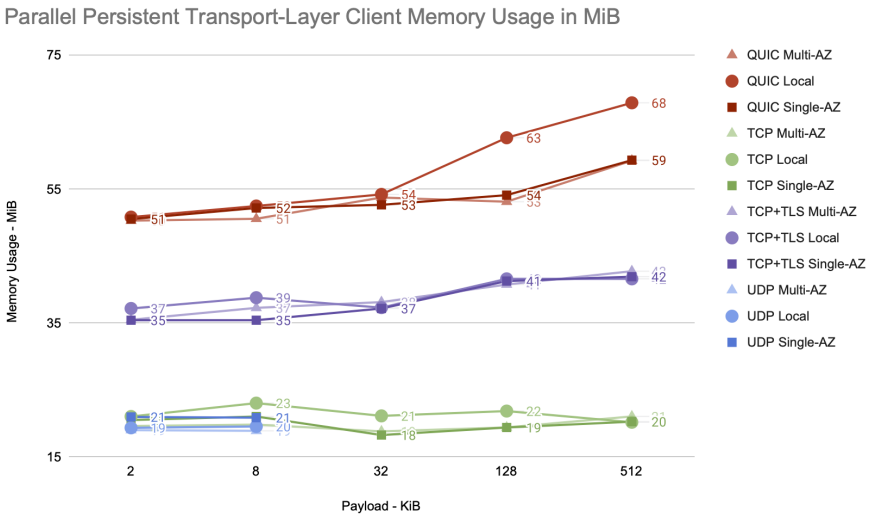

Figure 20: Parallel Persistent Transport-Layer Client Memory Usage in MiB

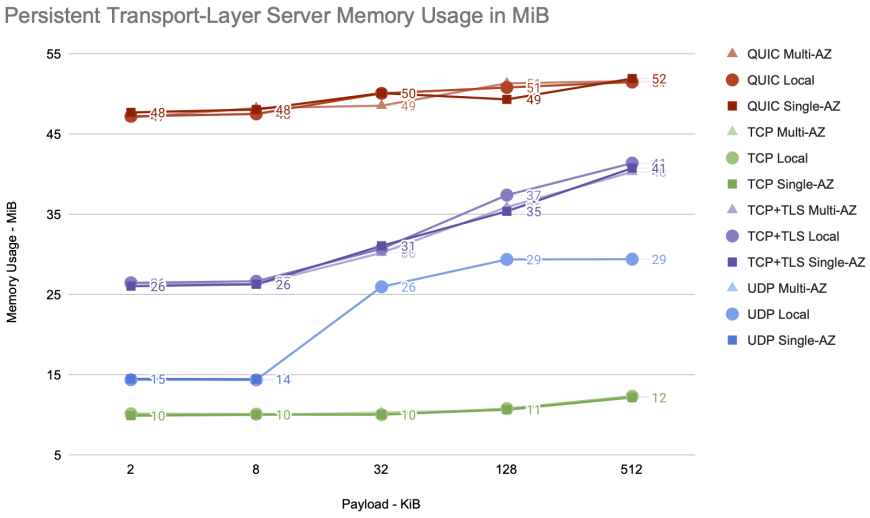

Figure 21: Persistent Transport-Layer Server Memory Usage in MiB

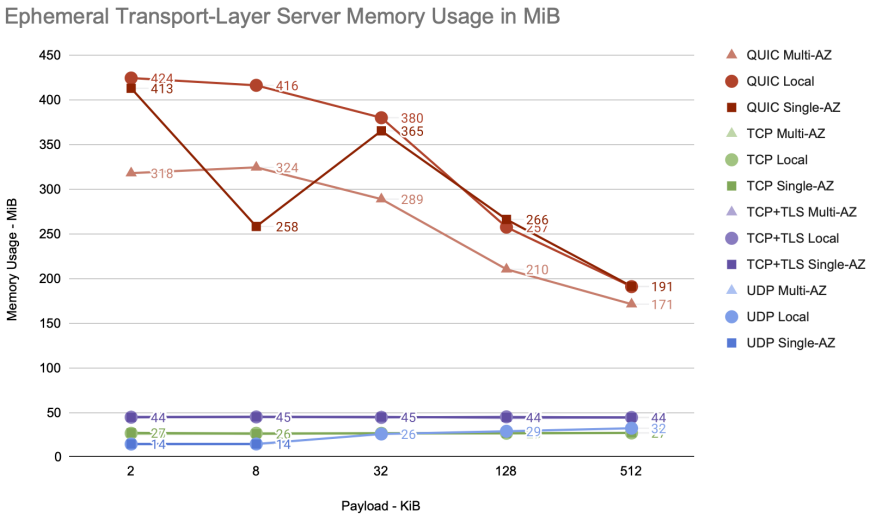

Figure 22: Ephemeral Transport-Layer Server Memory Usage in MiB

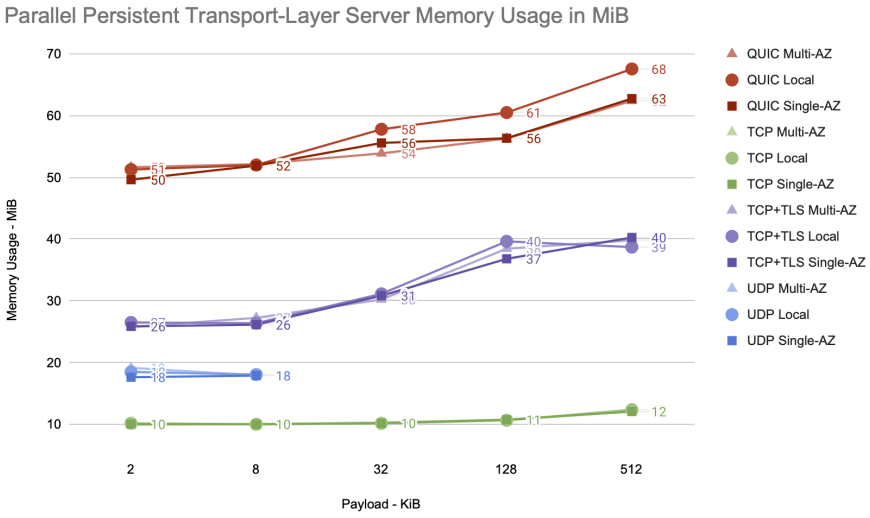

Figure 23: Parallel Persistent Transport-Layer Server Memory Usage in MiB

6 Cost

¡INSERT¿

6.1 Summary

¡INSERT¿

7 Application Protocols Experiments

7.1 Latency & Throughput

These charts represent the P90 of Latency and Throughput of all 10000 requests made by the persistent and ephemeral clients during the experiments.

Local HTTP/1 experiment had the best results overall, with lowest latency and highest throughput, since it fits perfectly with sequential requests. Local HTTP/2 was also at the top, only losing to HTTP/1 due to the extra features it implements overhead. For instance, HTTP/2 always compresses its requests/responses headers, but in these experiments there was close to nothing in the headers. Consequently, HTTP/2 spent some time dealing with compression, while HTTP/1 simply sent the request with headers in plain text. This pattern continues throughout Single-AZ and Multi-AZ experiments.

Both HTTP/1 and HTTP/2 maintained the previous observed pattern when using TLS. However, latency and throughput got worse due to TLS' data encryption and TLS handshake requirements. The latter is better observed in the persistent experiments, while the former is the main reason for the further difference between protocols during ephemeral experiments.

HTTP/3 starts with similar results to HTTP/2 with TLS, but starting from 32KiB data size it starts widening the difference between them both. As data size is increased, QUIC only gets worse, further demonstrating it performs poorly on a reliable network when compared with HTTP/2 with TLS. This is similar to the comparison between QUIC and TCP+TLS protocols, since it appears they are the deciding factor when it comes to performance between HTTP/2 and HTTP/3.

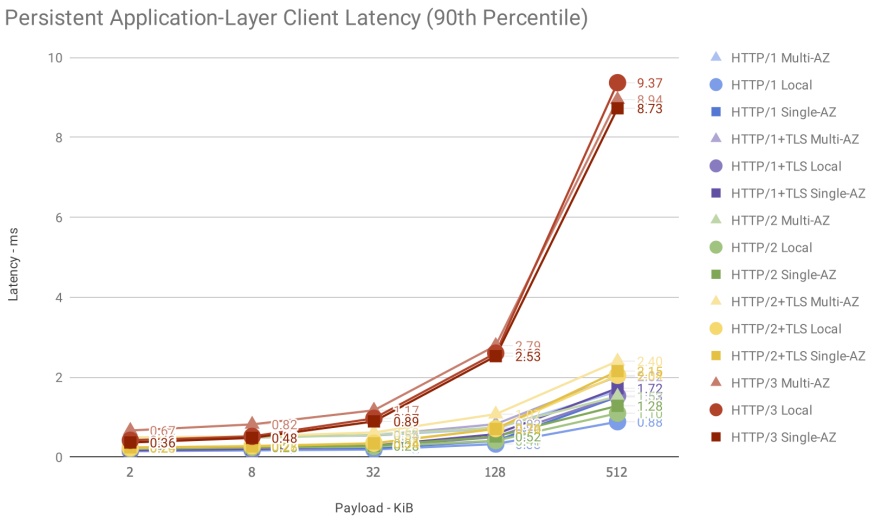

Figure 24: Persistent Application-Layer Client Latency (90th Percentile)

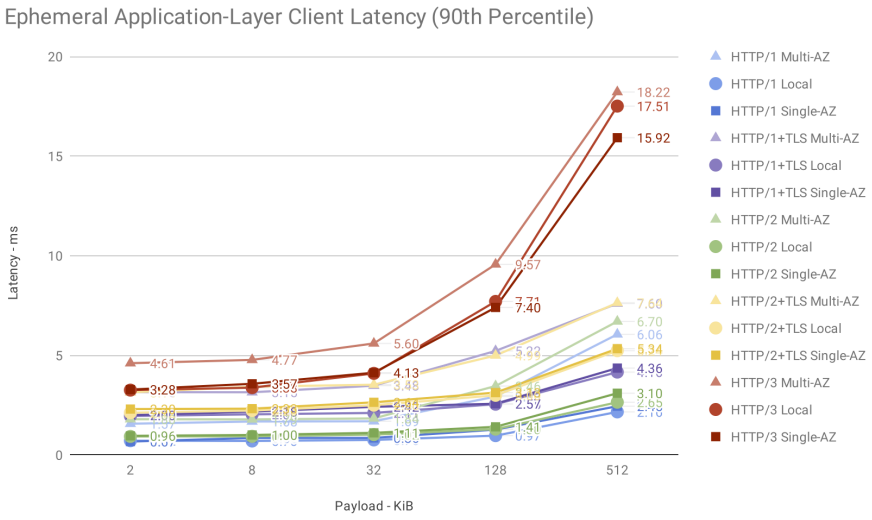

Figure 25: Ephemeral Application-Layer Client Latency (90th Percentile)

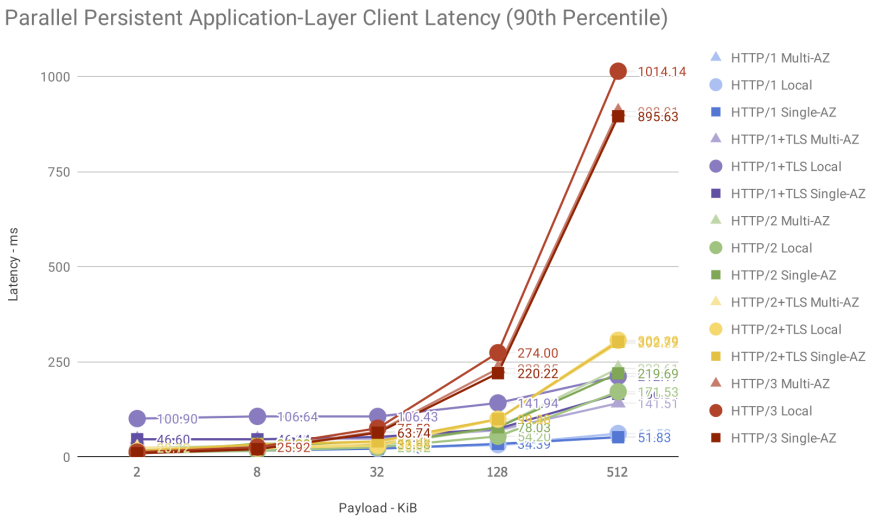

Figure 26: Parallel Persistent Application-Layer Client Latency (90th Percentile)

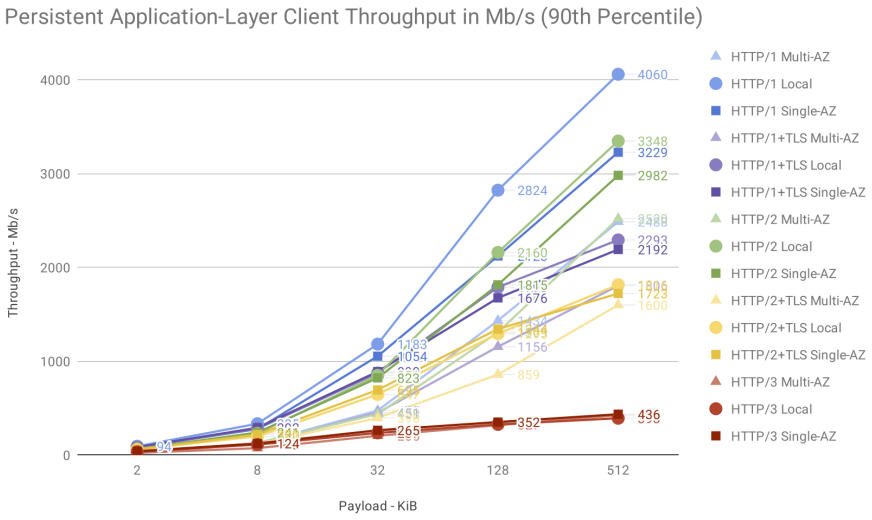

Figure 27: Persistent Application-Layer Client Throughput in Mb/s (90th Percentile)

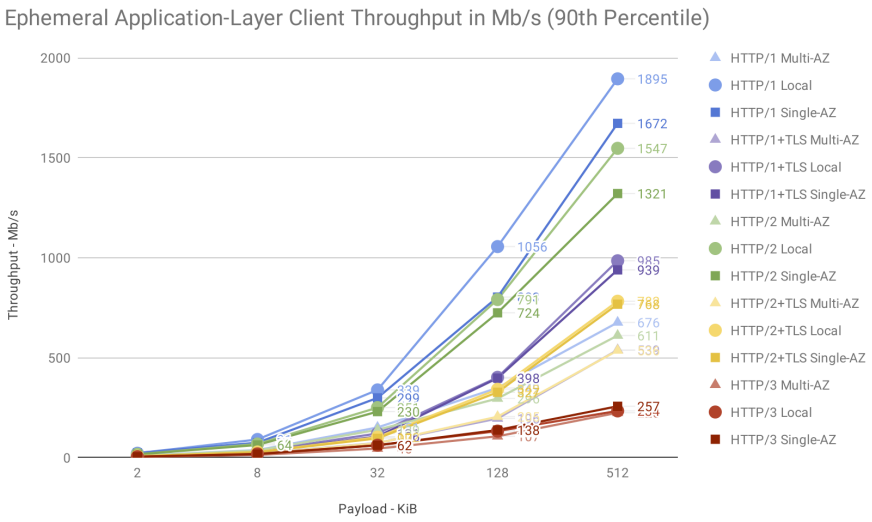

Figure 28: Ephemeral Application-Layer Client Throughput in Mb/s (90th Percentile)

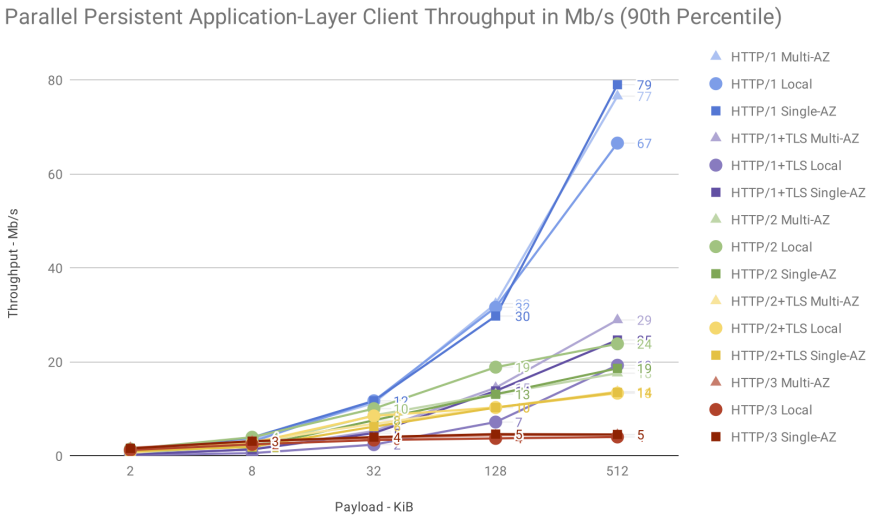

Figure 29: Ephemeral Application-Layer Client Throughput in Mb/s (90th Percentile)

7.2 CPU

These charts represent the CPU usage of clients and servers during ephemeral and persistent experiments.

They follow a similar pattern to the latency and throughput results, HTTP/1 has the lowest CPU usage, while HTTP/3 has the highest.

HTTP/3 is the worst since it has to deal with QUIC's cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state. HTTP/2+TLS and HTTP/1+TLS is better than HTTP/3, since they perform better in a reliable network, as stated before. Finally, HTTP/1 and HTTP/2 are the most performing since they do not have to deal with data encryption and TLS handshake overhead.

Server CPU usage is mirrored with client's. This can be explained by the fact that the server response has the same size as the client request, resulting in the same usage of CPU since they have to perform the same operations to be able to send data.

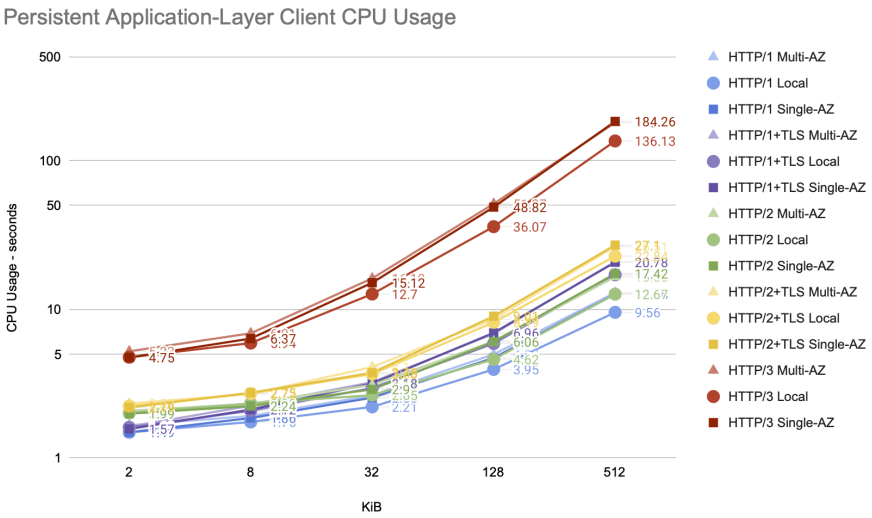

Figure 30: Persistent Application-Layer Client CPU Usage

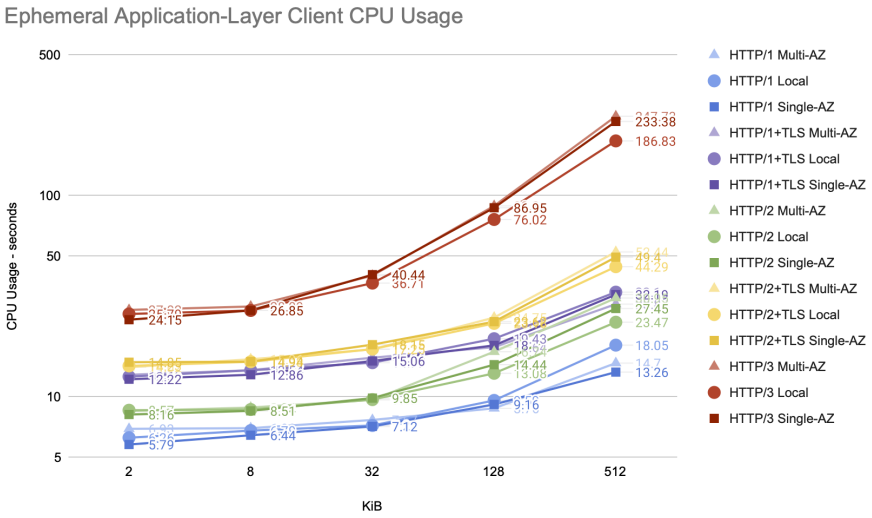

Figure 31: Ephemeral Application-Layer Client CPU Usage

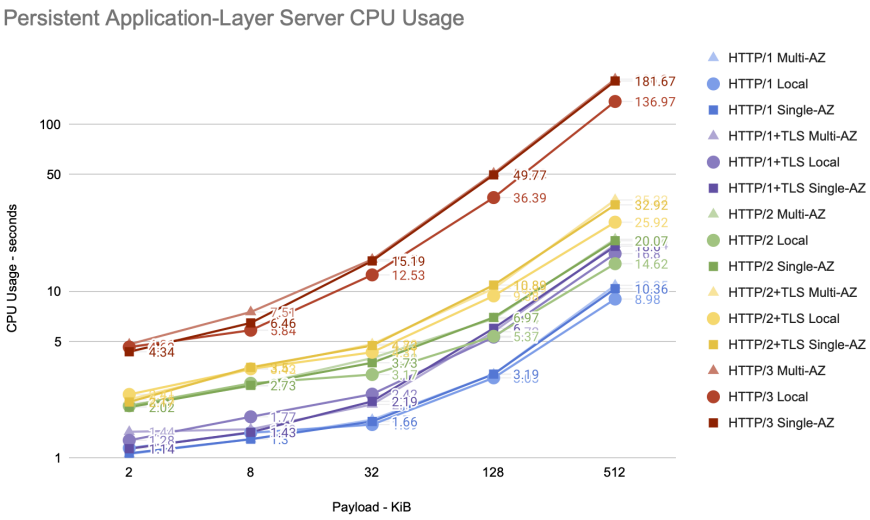

Figure 32: Persistent Application-Layer Server CPU Usage

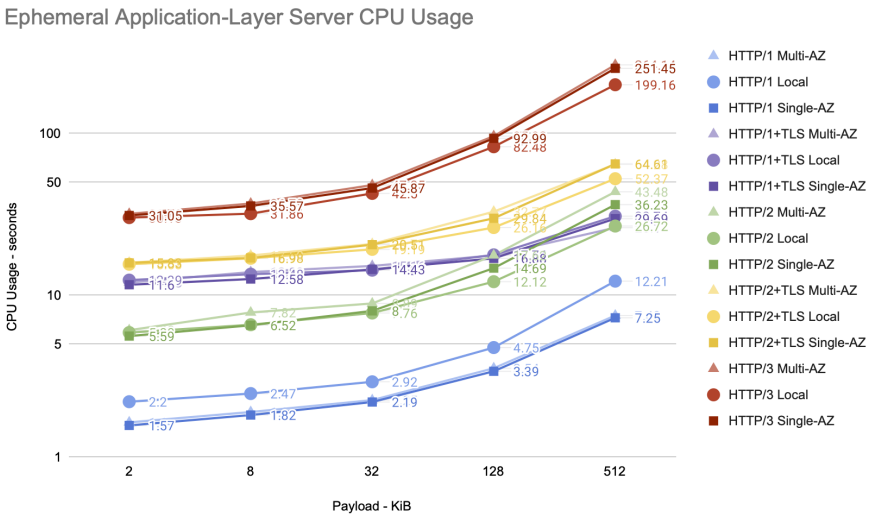

Figure 33: Ephemeral Application-Layer Server CPU Usage

7.3 Memory

These charts represent the memory usage of clients and servers during ephemeral and persistent experiments.

These results were very similar to transport-layer experiments. HTTP/1 and HTTP/2 were the most memory efficient due to their simplicity and use of TCP as transport protocol, which had similar results. HTTP/1+TLS and HTTP/2+TLS had increased memory usage due to TLS requirements for data encryption and TLS handshake. Finally, HTTP/3 had to use more memory due to QUIC's overhead, which trades memory for efficiency.

During ephemeral experiments, Go's garbage collector also had problems dealing with the high rate of created clients. Problem that begins worse with small data sizes, but it gets amortized as data sizes gets larger, which results in slower responses from the server, allowing Go's garbage collector time to do its job.

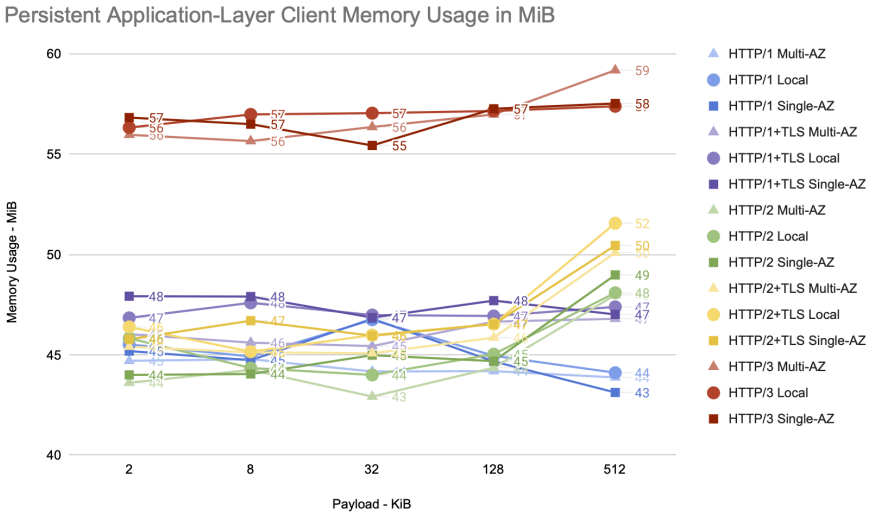

Figure 34: Persistent Application-Layer Client Memory Usage in MiB

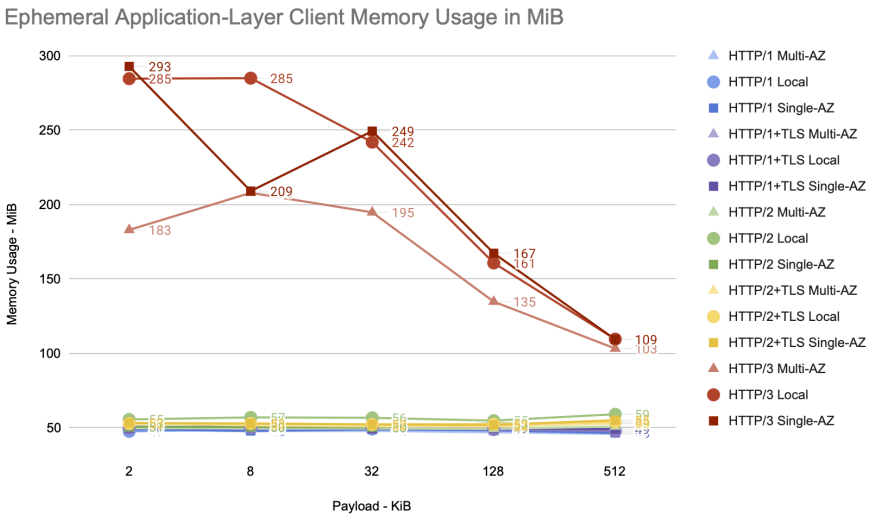

Figure 35: Ephemeral Application-Layer Client Memory Usage in MiB

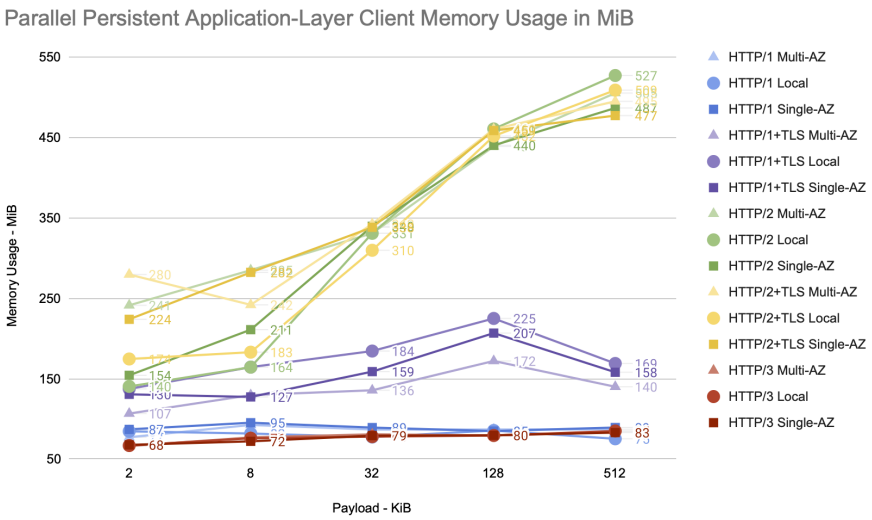

Figure 36: Parallel Persistent Application-Layer Client Memory Usage in MiB

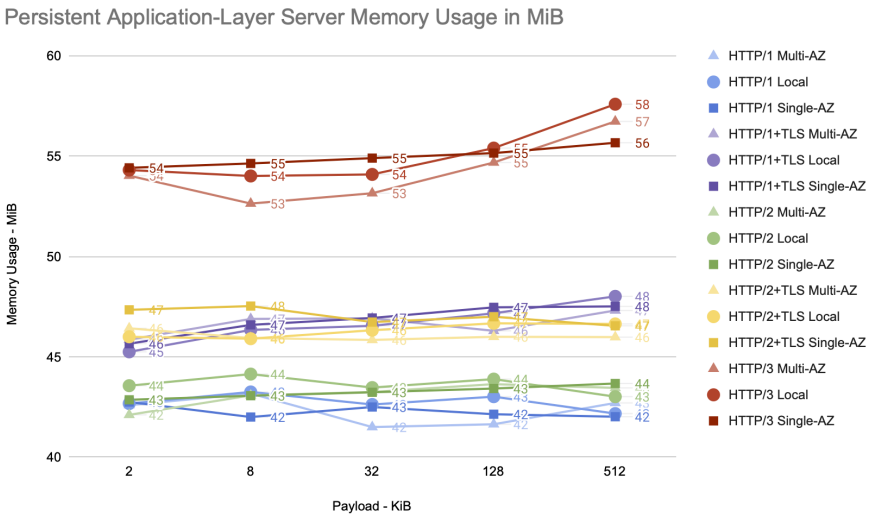

Figure 37: Persistent Application-Layer Server Memory Usage in MiB

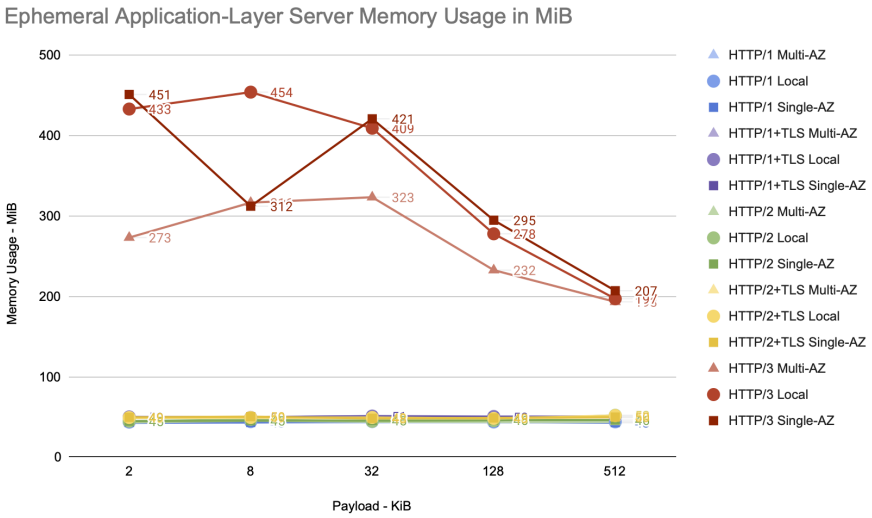

Figure 38: Ephemeral Application-Layer Server Memory Usage in MiB

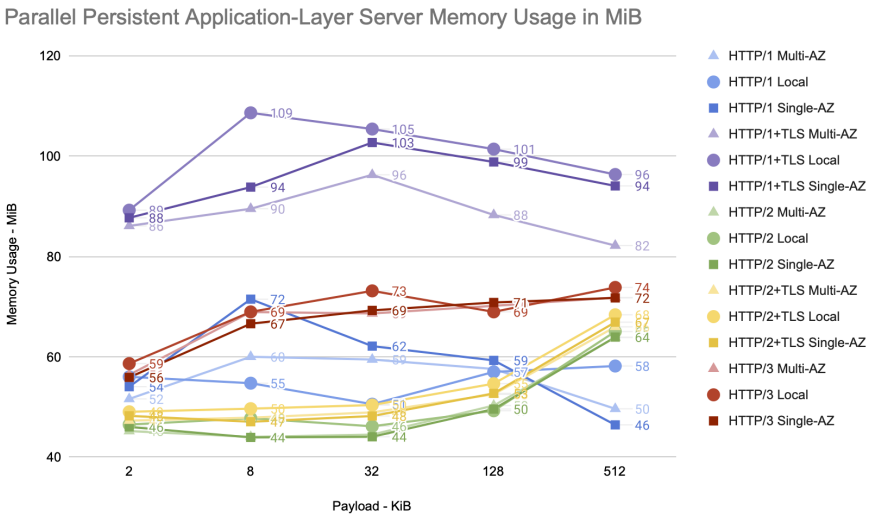

Figure 39: Parallel Persistent Application-Layer Server Memory Usage in MiB

8 Cost

¡INSERT¿

8.1 Summary

¡INSERT¿

9 Conclusion

QUIC has proven to be a better alternative to TCP on unreliable networks, addressing multiple problems of TCP when handling packet loss. Another advantage is the incorporation of the TLS protocol, forcing all connections to be encrypted. Relying on UDP and running on user-space, makes it compatible with existing network equipment and can be implemented by any application.

However, the experiments showed that on a cloud environment, where the network is extremely reliable, QUIC performed worse than TCP. Tuning kernel parameters to improve UDP traffic did not bring a significant advantage to QUIC. Additionally, it was also observed that QUIC requires more compute resources when compared to other protocols, increasing the cost of applications that use it.

HTTP/3, which is based on QUIC, suffers from similar problems and showed poor performance for interservice communication. It was not possible to push the protocol to its full potential, as most of its features are better used by a browser rather than in a cloud environment.

QUIC and HTTP/3 are still a great solution for user-facing applications, improving the experience for the end-user with an extra cost for the server. But it doesn't have a great fit with internal networks, where packet loss is extremely low. In that case it is better to use more traditional protocols that rely on TCP, even with TLS as an additional layer.

The protocol is relatively new and it's possible that in the future it becomes more competitive in reliable networks. In the meanwhile, TCP/TLS is a great solution that has been supporting the majority of the Internet for years, and it is not going away anytime soon.

References

[1] auto verification tls handshake.

[2] CNCF.

[3] data networks ip.

[4] distributed.

[5] kubernetes.

[6] os distributed systems.

[7] quic protocol.

[8] rfc1945.

[9] rfc2616.

[10] rfc2818.

[11] rfc2988.

[12] rfc3234.

[13] rfc5246.

[14] rfc7413.

[15] rfc7540.

[16] rfc768.

[17] rfc793.

[18] rfc9000.

[19] tcp udp comparison.

[20] telecom network security.

[21] what is cloud.

[22] what is http.

[23] L. Zhang.

Acronyms

ACK Acknowledge 9, 15

AWS Amazon Web Services 5, 20, 21, 24

AZ Availability Zone 20–24, 26, 31, 41

CA certificate authority 11

CNCF Cloud Native Computing Foundation 19

CPU Central Processing Unit 15, 18, 21, 23, 24, 31, 45

EC2 Elastic Cloud Computing 24

EKS Elastic Kubernetes Service 20, 21, 24

HTML HyperText Markup Language 12

HTTP Hyper Transfer Protocol 12–14, 16, 21, 22, 41, 45, 48

HTTPS Hypertext Transfer Protocol Secure 2, 13, 14, 21

IaaS Infrastructure as a Service 5, 17, 18

K8s Kubernetes 19–22, 24

MIME Multipurpose Internet Mail Extensions 12

OS Operating System 13, 18

P90 90th Percentile 26, 41

PaaS Platform as a Service 17

RFC Request for Comments 16

RTT Round-Trip Times 11, 14, 15, 26

SaaS Software as a Service 18

SYN Synchronize 10

SYN/ACK Synchronize/Acknowledge 10

TCP Transmission Control Protocol 2, 9–16, 21, 23, 26, 31, 34, 41, 48

TLS Transport Layer Security 2, 11–16, 21–23, 26, 31, 34, 41, 45, 48

UDP User Datagram Protocol 9, 10, 13, 21, 22, 26, 31, 34, 45

VM Virtual Machine 18, 24

VoIP Voice over Internet Protocol 9