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 09, 2021

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.

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

List of Figures

Contents

1 Introduction

QUIC is a transport-layer protocol developed with the intent of improving HTTPS traffic performance. It does that by replacing most HTTPS' components: HTTP/2, TCP, and TLS. Hence, it's able to deal with most of the Internet's traffic [22].

QUIC was designed to allow updates to transport protocols without disrupting the existing networking stack. It achieves that by being implemented in user-space on top of UDP [22]. Therefore, it allows it to be improved faster than it would be if it was implemented in a lower level.

Initially, the focus of QUIC's implementation was performance and not efficiency, resulting in CPU usage that was 3.5 times higher when compared with TCP+TLS [23]. Since then, optimizations were made to improve compute resources usage. Nonetheless, it's predicted that there will always be higher compute resources consumption than.

Different studies show the improvements of QUIC when used on user-facing applications. However, it's rare to find studies regarding its use on back-end services. Therefore, this study analyzes the use of QUIC for internal communications between services on a cloud environment. Checking if its benefits and drawbacks still holdup in this scenario.

To accomplish the objective of this study, a service will be written using QUIC and more traditional communication protocols. This benchmark service will be deployed on a Kubernetes (K8s) Cluster to simulate a real production environment. Metrics regarding latency, throughput, and usage of CPU and memory will be collected. With the metrics in hand, the protocols will be compared with each other. Explaining in detail the results found. Concluding which protocol would be better for use case analyzed in this study.

1.1 Document Structure

This document is structured in seven chapters, starting with this introduction.

The second chapter gives a general background in protocols analyzed throughout this study.

The third chapter gives a general background in distributed systems on cloud, which is the environment used throughout this study.

The fourth chapter explains how experiments setup was made. It describes the benchmark service, how it is experimented in different scenarios and how metrics are collected.

The fifth chapter analyzes the results of ephemeral and persistent clients experiments. Transport-layer and application-layer clients latency, throughput, and usage of CPU and memory are compared.

The sixth chapter is similar to the fifth. However, it analyzes the results of sequential and parallel clients experiments.

The last chapter finishes this study with a conclusion regarding QUIC's use on cloud.

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 [25]. 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 [9]. 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 [9]. 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 [26].

TCP must be able to deal with incorrect, lost, duplicate, or delivered out of order data to provide reliability [26]. 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 [26]. 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 [13].

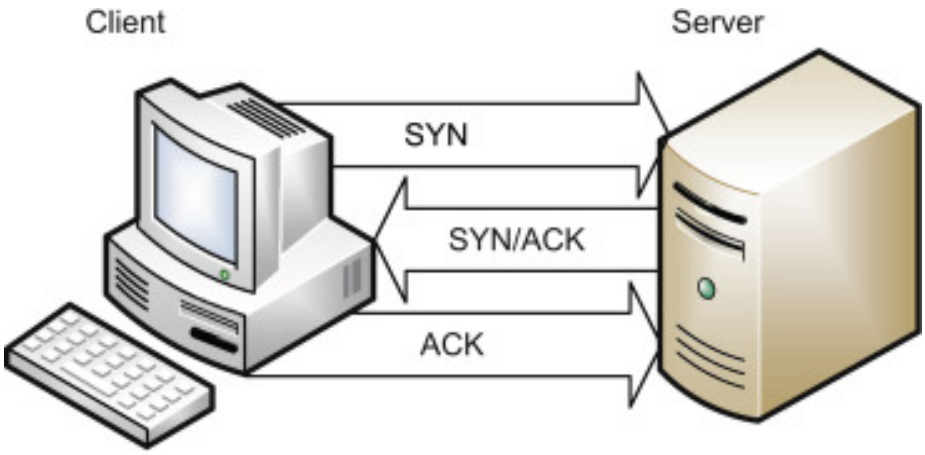

Figure 1: TCP Three-Way Handshake
Source: [13]

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 [13].

TCP is stream oriented, effectively meaning it has the ability to send or receive a stream of bytes [26]. 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 [9].

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 [1].

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 [15]. 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 [17]. 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.

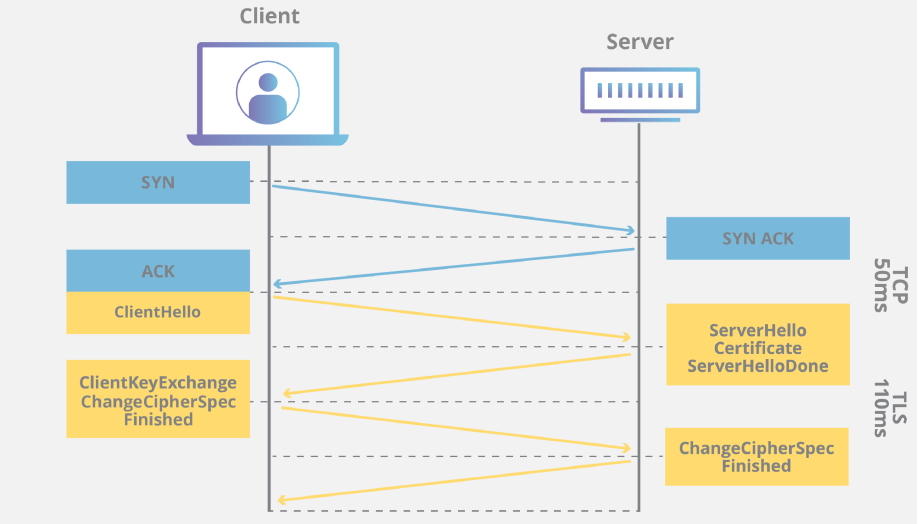

Figure 2: TLS Handshake
Source: [11]

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 [6, 18, 12].

It is a generic, stateless protocol which uses a predefined set of standards and rules for exchange of information through a request-response process [6, 18]. 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 [18].

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 [18]. 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 [18]. 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 [12]. 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 [5]. 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 [5]. 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 [5].

2.7 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is an extension of HTTP, conceptually equivalent to HTTP over TLS [27]. 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 [27].

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 [23]. 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 [22]. 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 [23].

A middlebox is when a device adds functionality other than packet forwarding to an IP router, such as filtering, altering, and manipulating traffic [7]. 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

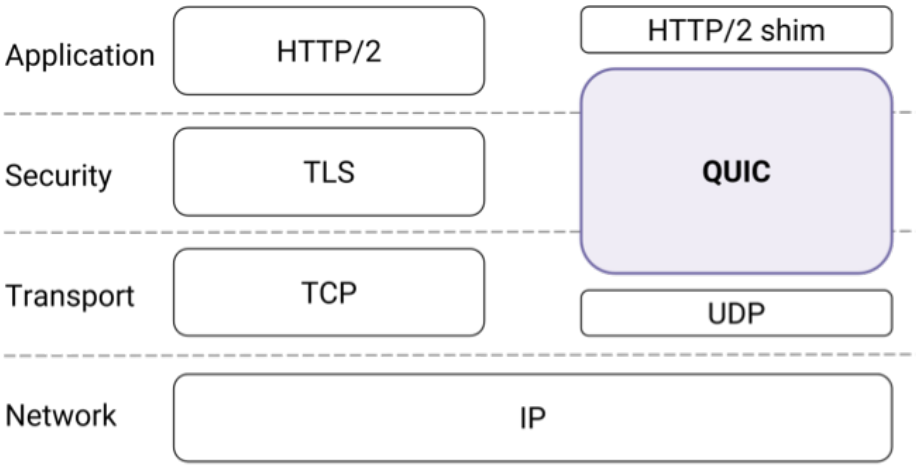

Figure 3: QUIC in the traditional HTTPS stack
Source: [23]

[7]. QUIC addresses this issue by encrypting its packets, therefore avoiding middlebox dependency and data tampering [23].

HTTP/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 [8].

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) [23, 22]. 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.

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

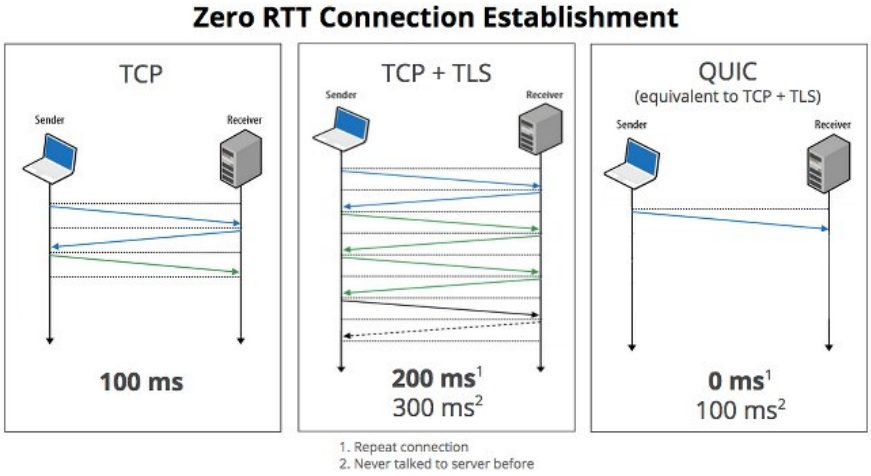

Figure 4: Zero RT Connection Establishment
Source: [20]

[23, 24, 22, 31].

QUIC eliminates TCP's retransmission ambiguity problem by including a new packet number to all packets, even those including retransmitted data [22]. 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 [23].

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 [23]. Optimizations were made after this assessment, decreasing CPU usage difference to 2 times TCP+TLS'.

2.9 HTTP/3

HTTP/3 is effectively HTTP over QUIC. It provides a transport for HTTP 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% of running web browsers, Google Chrome being one of them [33].

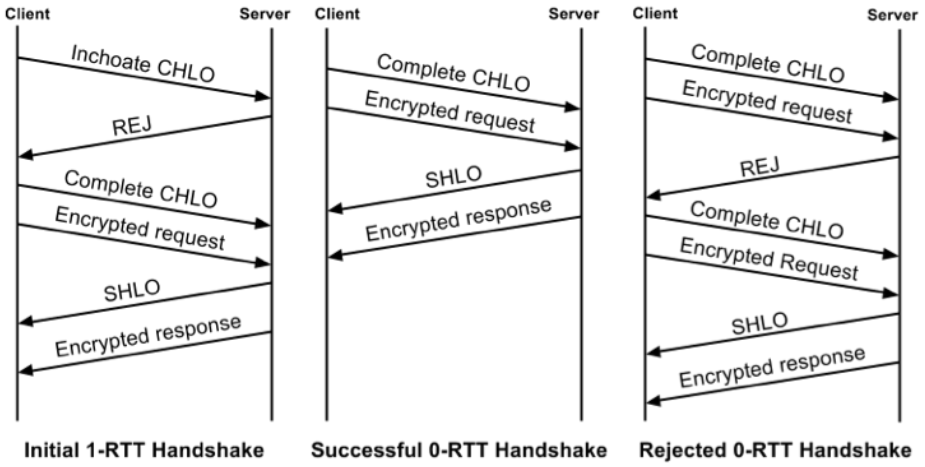

Figure 5: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake
Source: [23]

2.10 Summary

This section went over the important aspects of each transport-layer and application-layer protocols experimented throughout this study. Therefore, their differences should be clear, as well as what they try to improve.

The next section will explore a few aspects of distributed systems on cloud environment. These systems makes extensive use of the observed protocols as a way of communicating with each other. Without them, however, they would not be able to talk to each other and achieve cooperation.

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 [28]. 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: ephemeral and persistent.

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 [3].

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 [4]. 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 [4].

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.

Multi regional 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 [4]. 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

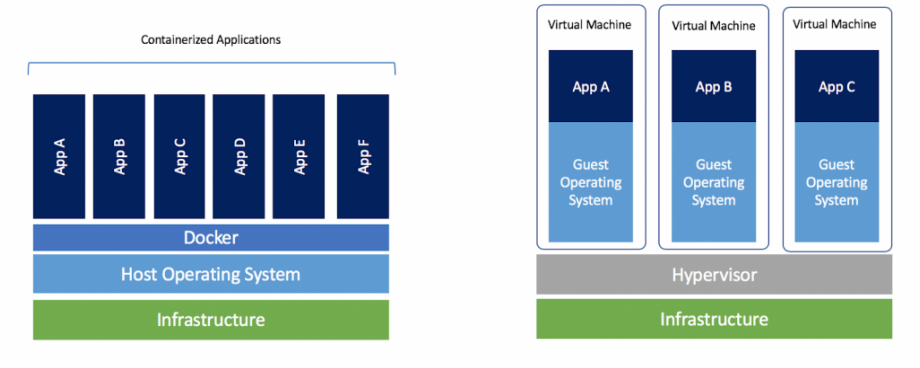

Figure 6: Containers versus Virtual Machines
Source: [16]

environment that contains all of the dependencies of any environment whether it's development, testing, or production.

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), K8s is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications [29]. Additionally, it's considered the most widely used container orchestration platform by CNCF's K8s Project Journey Report from 2019 [10].

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

This section finished all required aspects of managing distributed systems on cloud. Therefore, their challenges and benefits should be clear, as well as their importance to the computing world.

The next section will bring details on how the implemented Benchmark Service will perform experiments. They are going to put previously described protocols to the test in a cloud environment. Consequently, being able to assess their results.

4 Experiments

Up until now, protocols and cloud background were discussed. These backgrounds serves as a way to understand the decisions taken during experiments in the following sections.

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.

4.1 Objectives

These experiments have the objective to compare QUIC with other transport-layer protocols: UDP, TCP, and TCP+TLS. Their results are going to make assessments about each transport-layer protocol performance and resource efficient use. Therefore, latency, throughput, and CPU and memory usage metrics are going to be collected.

Furthermore, to be able to observe QUIC's influence on a production environment, they have the objective to compare HTTP/3 with other application-layer protocols: HTTP/1, HTTP/1+TLS, HTTP/2, and HTTP/2+TLS. Their results are going to make assessments about each application-layer protocol performance and resource efficient use. Therefore, latency, throughput, and CPU and memory usage metrics are also going to be collected.

4.2 Preparation

This subsection explains what kind of preparations were made to be able to experiment on protocols. This includes how the benchmark service was implemented, where it was deployed and what kind of scenarios its trying to simulate.

4.2.1 Benchmark Service

The benchmark service was developed to be able to experiment with each protocol. Their impact on performance and resource usage are analyzed by collecting metrics about latency, throughput, and CPU and memory usage. Therefore, it was developed in the most efficient way possible to avoid any external noises that might pollute the results.

Google's QUIC experimentations paper refers *lucas-clemente/quic-go*'s QUIC implementation as being one the ones that provided invaluable feedback [23]. Therefore, the benchmark service was implemented in Go and uses this library on QUIC's and HTTP/3's experiments.

This implementation took into consideration three types of clients: ephemeral, sequential persistent, and parallel persistent clients. By separating clients into these three categories, experiments are able to simulate three 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. It represents 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 sequential persistent connections. It represents 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 and performing one request at a time.

The third and final scenario explores parallel persistent connections. It represents a service similar to the previously explained. However, it can perform multiple requests at the same time.

Metrics are collected through two different measurements. The first uses the *time* command to get information about CPU time, memory usage and wall clock time as seen in Figure 7. The second uses the benchmark service's logs to calculate latency and throughput. These logs are the time each request took to receive its response.

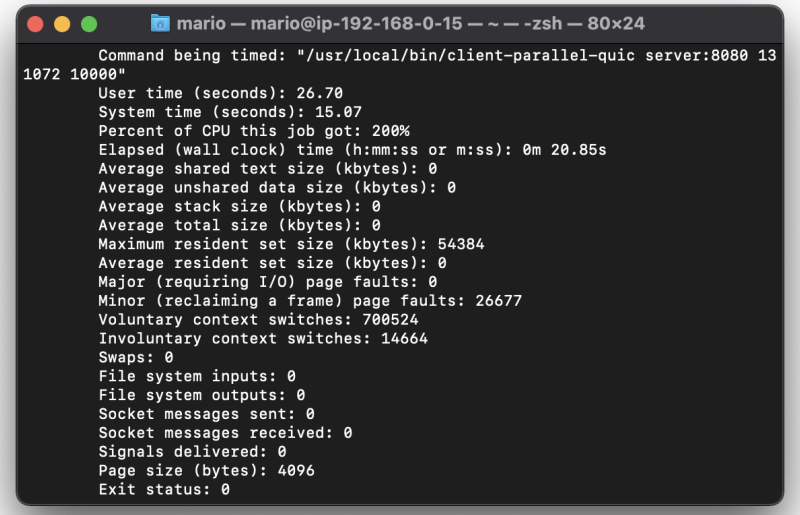

Figure 7: *time* command Result

The benchmark service itself is not enough to simulate a production environment, however. Therefore, all experiments are made inside a K8s Cluster.

4.2.2 K8s Cluster

K8s is a production-grade container orchestrator, which is a common production environment used by most companies that have to deal with container manage-

ment in the cloud. By choosing to use this environment allows easy networking setup for nodes running on different AZs.

As every data center is always under the danger of downtime due to a natural disaster, running nodes on different AZs is a must have for companies that requires a disaster recovery plan. Therefore, enabling them to offer high availability services.

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 on multiple AZs, since, depending on the configuration, applications may be running in any node.

K8s allows pod affinity and taint to be added to pod's configurations. This allows complete control over which pods are going to be scheduled to a specific node. Therefore, enabling all scenarios above to be possible.

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 *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. Finally, during the *Multi-AZ* scenario, 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.

The K8s cluster was deployed through the AWS EKS service. AWS allows easy setup with *eksclt* command-line tool. Therefore, being able to quickly provide a cluster for experimenting.

4.2.3 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.

Pricing was also taken into account when preparing for experiments. AWS not only charges for the Elastic Cloud Computing (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, performing requests and responses with 512KiB payloads 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, sequential persistent, and parallel persistent clients, all Multi-AZ experiments with 512KiB payloads costs approximately $5.28 of data tranfer.

The payloads size is predetermined in compilation time. This value varies in the following manner: 2KiB, 8KiB, 32KiB, 128KiB, and 512KiB. These values were chosen due to cost and since 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].

As each K8s node is an EC2 instance, it's important to choose carefully. The amount of CPU and memory can affect experiments. Therefore, it's required to choose a host with resources to avoid experiments to throttle.

Initial experiments were performed with the smallest EC2 instance type *m6i.large* and K8s pod's CPU limit was set to 1. However, QUIC ended up being throttled and required more resources to be able to perform correctly.

To avoid throttling, the CPU limit was removed and the EC2 instance type was changed to *m6i.xlarge*, which contains 2 CPUs instead of 1. This assured pods to have all computational power it needs to perform correctly.

During QUIC's experiments, QUIC's Go implementation issued a warning about the UDP receive buffer size (Figure 8). It contained a link recommending it to increase this value, since Linux places restrictions to UDP decreasing its throughput [30]. Therefore, it was increased from 128KiB to 26MiB, more than the recommended amount.

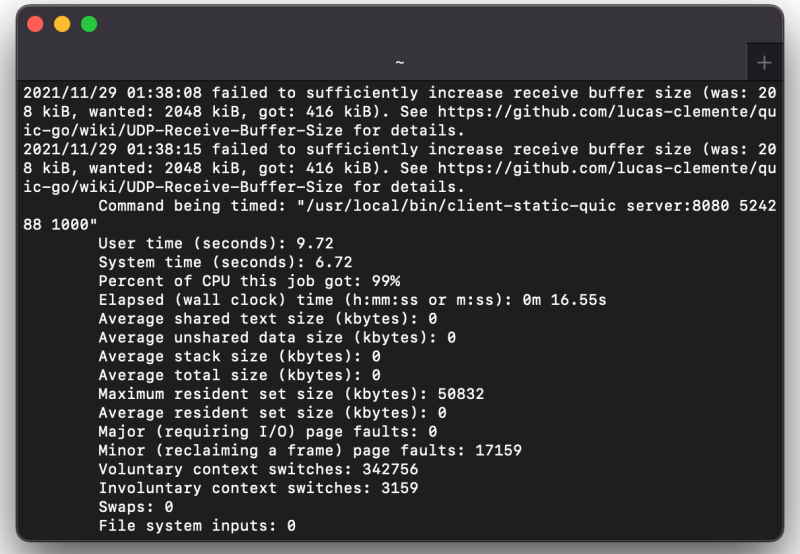

Figure 8: *quic-go* UDP Receive Buffer Size Warning

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

other experiments only used use1-az1.

4.3 Execution

Experiments execution is pretty straight forward. An even groups of nodes are created, and then a set of manifests is applied to the K8s Cluster. Thus, creating all necessary pods.

Clients and servers pods were allocated depending on the scenario being tested. *Local* scenario requires one node, *Single-AZ* requires two nodes in the same AZ, and *Multi-AZ* requires two nodes in different AZs. As this logic was incorporated into pods manifests, it's only a matter of patience to finish all experiments.

After experimentation and collecting metrics, node groups were deleted to make way to completely new node groups. This guarantees experiments are independent and don't affect one another.

4.4 Summary

This section explained the benchmark service, what kind of experiments were performed and how metrics were collected. It should be clear what kind of scenarios each experiment is trying to test and their objectives.

The following sections describes experiments results. It compares ephemeral and sequential persistent clients, and sequential persistent and parallel persistent clients. Consequently, putting each protocol to its limit to be able to show their benefits and drawbacks.

5 Ephemeral and Persistent Clients Experiments

As experiments were described in the *Experiments* section, this section contains all results regarding ephemeral and sequential persistent clients, referred, respectively, as ephemeral and persistent clients.

It's divided into two subsections: transport-layer protocols and application-layer protocols. Each subsection is divided into three subsections: Latency & Throughput, CPU Usage, and Memory Usage. Consequently, all metrics collected are analysed.

5.1 Ephemeral and Persistent Transport Clients

These experiments performs requests sequentially with transport-layer protocols in two different ways. While the ephemeral experiment always creates a new client for each request, the persistent experiment creates only one client that performs all requests. Consequently, client creation and connection establishment overhead can be observed.

5.1.1 Latency & Throughput

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

Scenarios Pattern

Throughout experiments it is possible to observe a pattern. Local scenarios usually had better performance when compared to Single-AZ and Multi-AZ scenarios. This happens because, by using localhost, the traffic never leaves the host machine, it bypasses any local network interface hardware, being executed exclusively via software [21].

All other scenarios are limited by the networking capabilities of the AWS' EC2 instance, which is around 12.5Gb/s [2]. Thus, Local-AZ and Multi-AZ scenarios can only reach a theoretical maximum of 12.5Gb/s of throughput, while Local scenario is not affected by such limits.

Additionally, the Single-AZ scenarios were better than Multi-AZ scenarios due to the overhead from sending the datagram to another data center. Even though remaining in the same region, it results in a higher latency.

UDP Unreliability

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.

TCP is more optimized than UDP

Local TCP experiment was by far the most efficient, with almost zero latency when connection was persistent, reaching approximately 19Gb/s of speed (Figure 9) 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.

TCP is considered a stream oriented protocol, it receives a stream of bytes from the application-layer and it's his responsibility to divide it into segments before passing it on. UDP on the other hand, is considered a message oriented protocol, it defers the burden of dividing the streams of data into messages to the application-layer. While TCP's data division is implemented in the kernel, UDP's depends on the application. Thus, TCP is able to have a higher througput than UDP.

Connection Establishment Impact

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 (Figures 9 and 10). Both results are still better than other TCP and all TCP+TLS' experiments, but brings them to a similar level.

TLS Overhead

As expected, TLS encryption adds a slight overhead on TCP during persistent clients experiments (Figures 9 and 11), as it needs to perform a TLS handshake, consequently adding couple of extra RTTs to be able to establish a connection.

This difference is amplified during ephemeral client experiments (Figures 10 and 12) due to having to perform a TLS handshake before every request.

QUIC & UDP

Though UDP have a higher throughput than QUIC, they share a common pattern. This can be better observed during Local experiments with persistent clients (Figure 11). This similarity exists since QUIC uses UDP as its underlying transport mechanism, therefore it shares the same limitations as UDP.

QUIC's Handshake Efficiency

Although QUIC did not perform nearly as well as TCP+TLS, it managed to be less impacted by ephemeral clients.

During the Multi-AZ scenario, while TCP+TLS' throughput went from 3160 Mb/s with persistent clients to 668 Mb/s with ephemeral clients, equivalent to a decrease of 79%, QUIC's throughput went from 487 Mb/s to 231 Mb/s, equivalent to a decrease of 52% (Figures 11 and 12). This demonstrates the

impact of QUIC's handshake since it only needs 1-RTT with unknown servers, while TCP+TLS' handshake requires, in general, 3-RTTs.

QUIC's Bad Performance

QUIC's experiments were the worst, with latency and throughput almost 6 times worse than TCP+TLS' during Multi-AZ with persistent client experiment (Figures 9 and 11).

As this is a cloud environment, packet loss rate is pretty low, resulting in a reliable network. QUIC's main purpose is to improve performance of devices operating in an unreliable network, such as wireless and mobile networks. Thus, QUIC's benefits can only be seen when it's used in an environment with a high rate of packet loss.

One of the main reasons why QUIC is less efficient than TCP+TLS, it's possibly because TCP's implementation is more efficient than UDP's. As QUIC has UDP as its bottleneck and UDP is worse than TCP on a reliable network, QUIC can only be as efficient as UDP. Consequently, it will not be able to surpass TCP's performance with UDP's current limitations.

QUIC's results show it's not meant to be used on distributed systems. These applications are usually contained in environments with reliable networks, meaning that TCP+TLS will, for the time being, be a better fit.

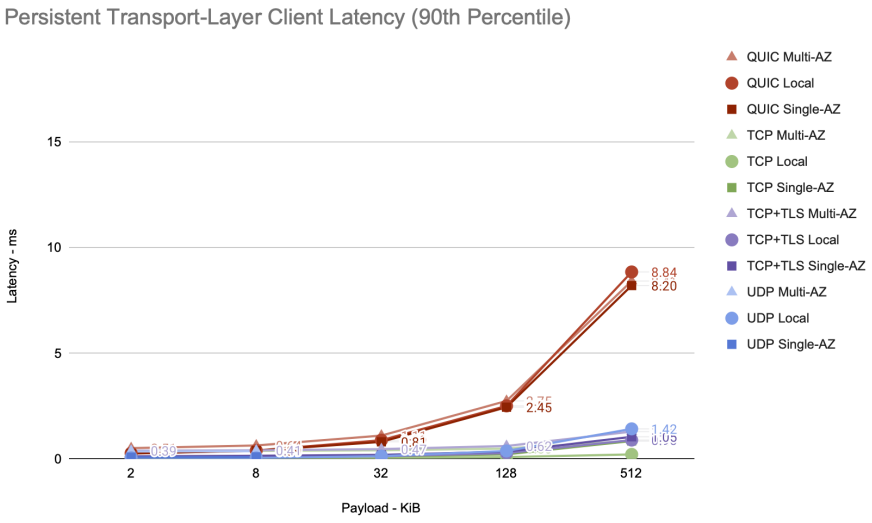

Figure 9: Persistent Transport-Layer Client Latency (90th Percentile)

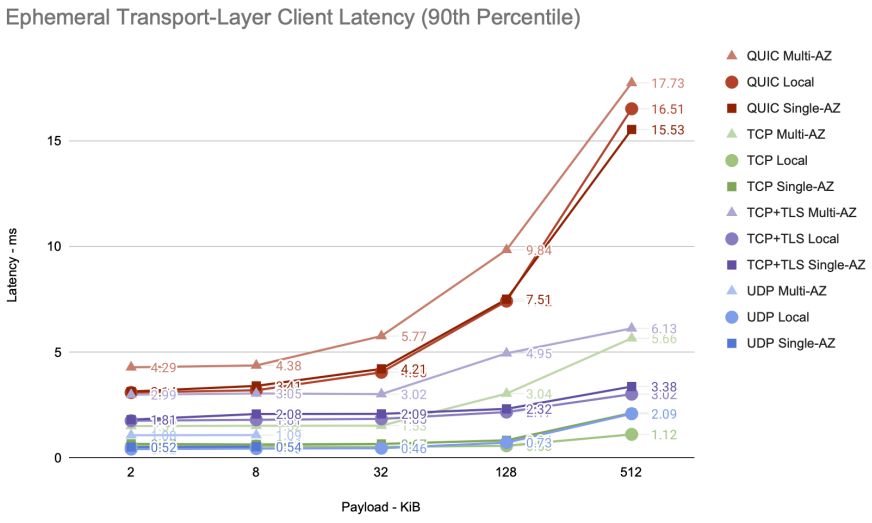

Figure 10: Ephemeral Transport-Layer Client Latency (90th Percentile)

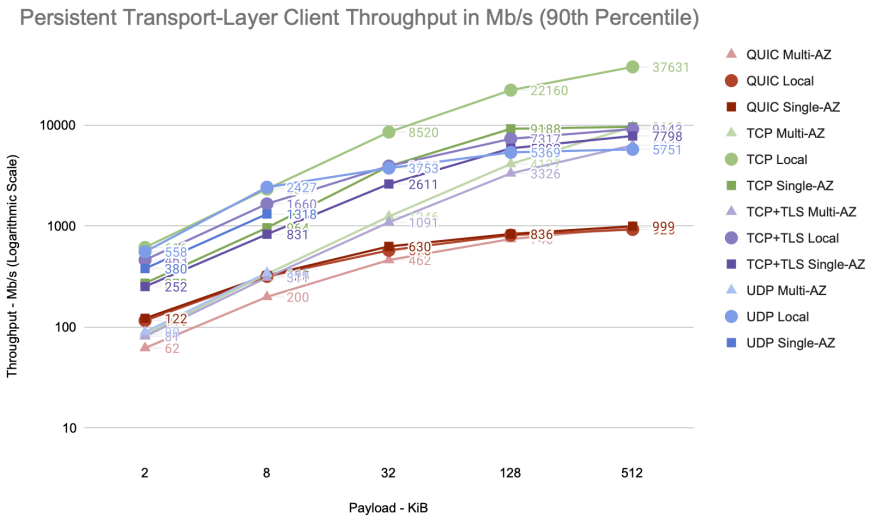

Figure 11: Persistent Transport-Layer Client Throughput in Mb/s (90th Percentile)

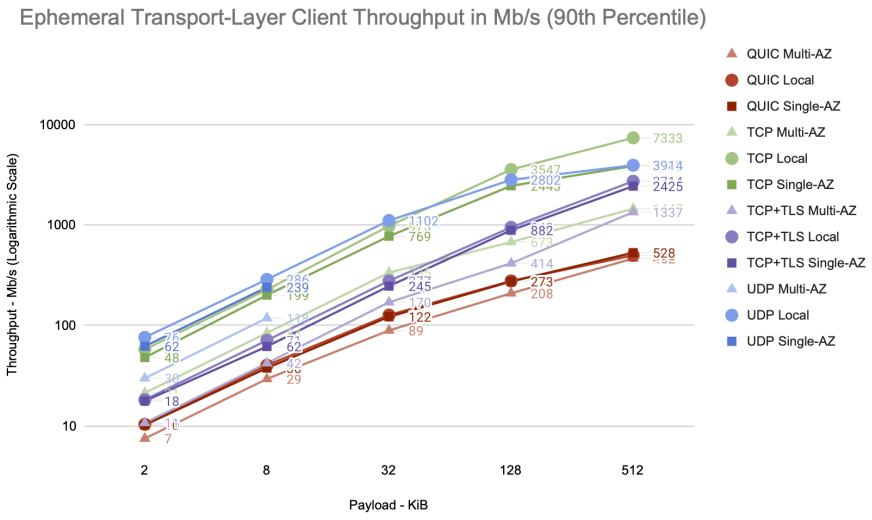

Figure 12: Ephemeral Transport-Layer Client Throughput in Mb/s (90th Percentile)

5.1.2 CPU Usage

Charts on Figures 13, 14, 15, and 16 represent the CPU usage of clients and servers during ephemeral and persistent experiments.

CPU Client and Server Similarities

Even though CPU usage pattern is very similar between client and server, each kind of protocol resulted in different characteristics.

While TCP and UDP client CPU usage were greater than the server's, TCP+TLS and QUIC's server usage were greater. This happens since the latter have to deal with performing TLS' requirements, which appears to have higher requirements for the server.

TCP is more optimized than UDP

During the first two packet sizes, UDP appears to be more efficient. However, from 32KiB onward, it increases CPU usage and surpasses TCP costs. This further shows that TCP's streams oriented implementation is more efficient when dealing with a larger amount of data than UDP's message oriented implementation.

Other Protocols

In the case of the other protocols, TCP is the most efficient. TCP+TLS has a higher cost than the latter due to the fact that it has to deal with the extra TLS overhead, needing to encrypt data and performing TLS handshakes to establish a connection.

As expected, QUIC is the most costly among all protocols. It has increased CPU usage as a tradeoff for efficiency, since it has to deal with cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state.

Scenarios Differences

CPU also increases between scenario 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 more CPU time.

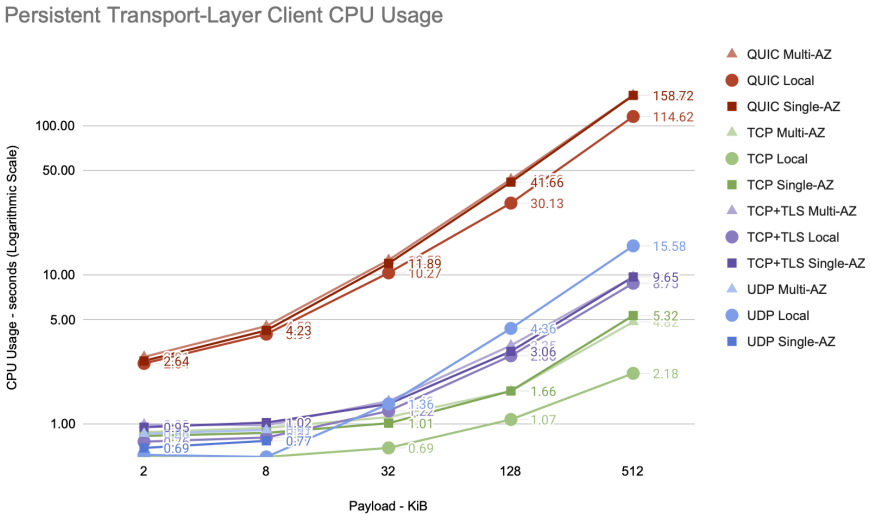

Figure 13: Persistent Transport-Layer Client CPU Usage

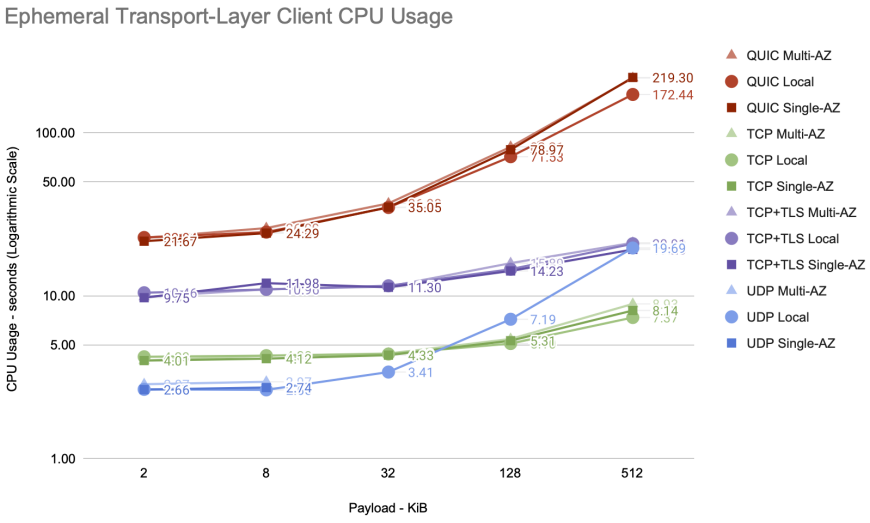

Figure 14: Ephemeral Transport-Layer Client CPU Usage

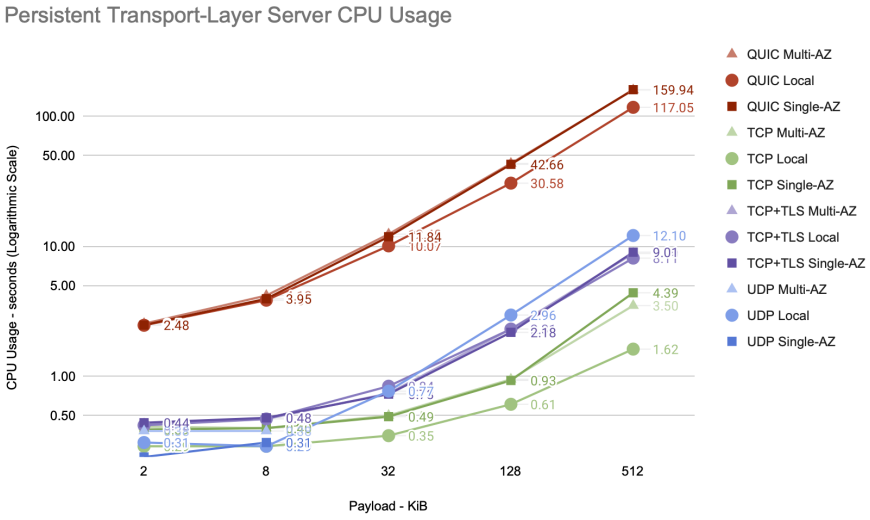

Figure 15: Persistent Transport-Layer Server CPU Usage

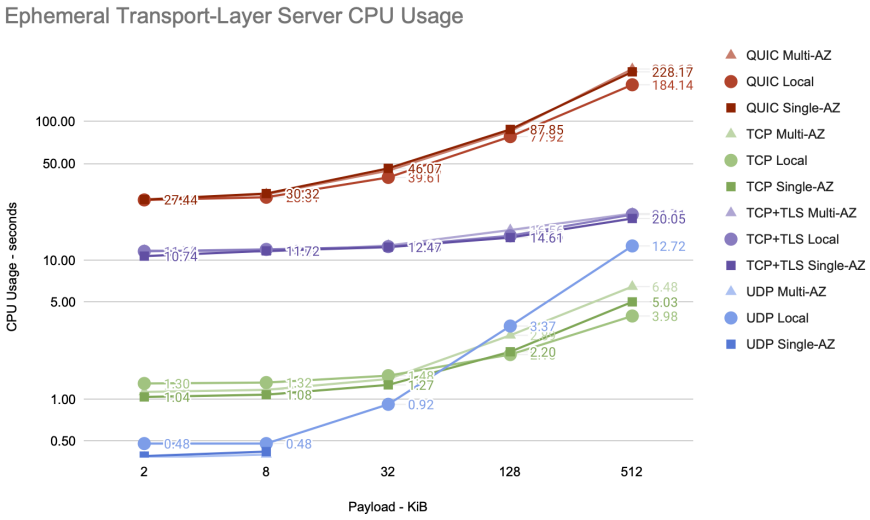

Figure 16: Ephemeral Transport-Layer Server CPU Usage

5.1.3 Memory Usage

Charts on Figures 17, 18, 19, and 20 represent the memory usage of clients and servers during ephemeral and persistent experiments.

Overall Memory Cost

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 Unusual Ephemeral Client Memory Cost

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, during ephemeral experiments it had an interesting behaviour. On smaller payloads it uses a ton of memory, almost 10 times TCP+TLS', while only using half with larger payloads (Figure 18 and 20).

This happens during ephemeral experimentations only because the application is constantly creating and deleting clients, therefore Go's garbage collector is unable to quickly clean unused clients. On smaller payloads, the rate of client creation is higher because the requests end faster, in contrast to larger payloads, which require more time to complete requests, allowing Go's garbage collector more time to do its job. This behaviour is intensified during QUIC's experiments due to its higher cost when creating clients.

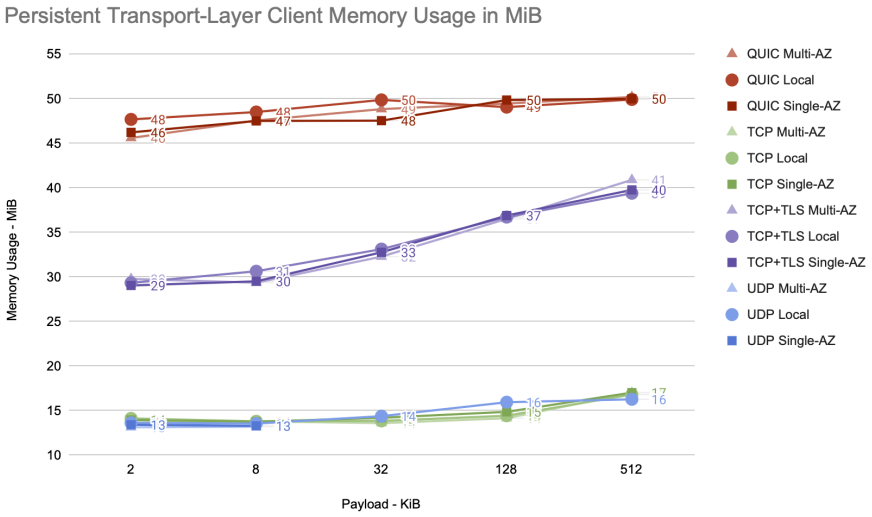

Figure 17: Persistent Transport-Layer Client Memory Usage in MiB

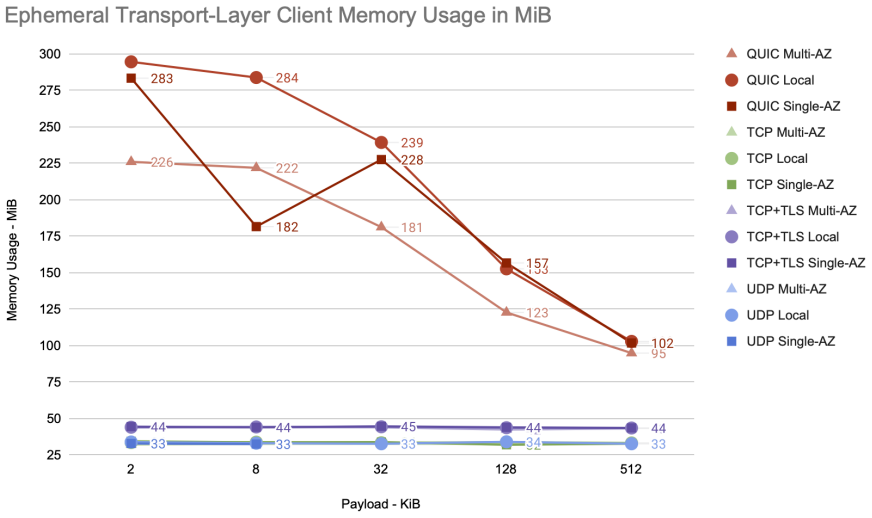

Figure 18: Ephemeral Transport-Layer Client Memory Usage in MiB

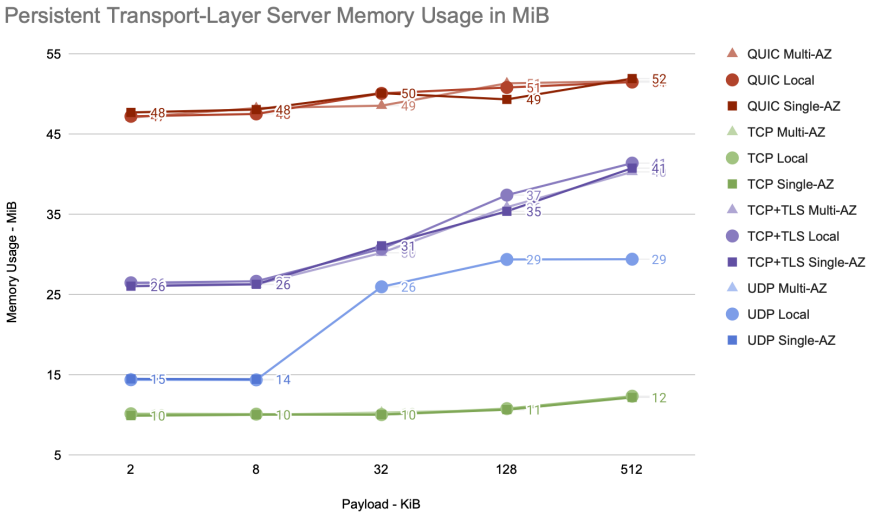

Figure 19: Persistent Transport-Layer Server Memory Usage in MiB

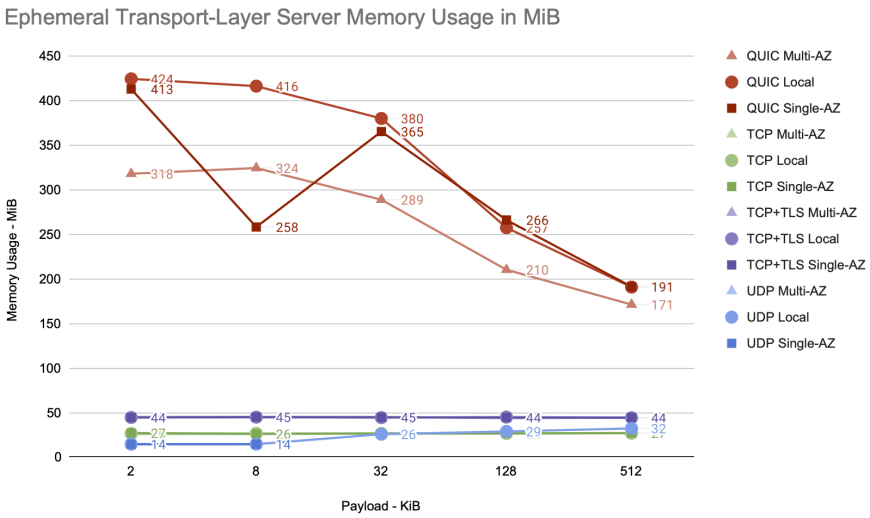

Figure 20: Ephemeral Transport-Layer Server Memory Usage in MiB

5.2 Ephemeral and Persistent Application Clients

These experiments performs requests sequentially with application-layer protocols in two different ways. While the ephemeral experiment always creates a new client for each request, the persistent experiment creates only one client that performs all requests. Consequently, client creation and connection establishment overhead can be observed.

5.2.1 Latency & Throughput

Charts on Figures 21, 22, 23, and 24 represent the P90 of Latency and Throughput of all 10000 requests made by the persistent and ephemeral clients during the experiments.

HTTP/1 and HTTP/2

Local HTTP/1 experiment had the best results overall, with lowest latency and highest throughput. 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.

TLS Impact

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/2+TLS and HTTP/3

HTTP/3 starts with similar results to HTTP/2+TLS, but starting from the 32KiB payload it starts widening the difference between them both. As payload increases, QUIC only gets worse, further demonstrating it performs poorly on a reliable network when compared with HTTP/2+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.

QUIC's Handshake Efficiency

Although HTTP/3 did not perform as well as HTTP/2+TLS, it managed to maintain the same behaviour as transport-layer experiments, since it was less impacted by ephemeral clients.

During the Multi-AZ scenario, while HTTP/2+TLS' throughput went from 1600 Mb/s with persistent clients to 536 Mb/s with ephemeral clients, equivalent to a decrease of 66.5%, HTTP/3's throughput went from 425 Mb/s to 224 Mb/s, equivalent to a decrease of 47% (Figures 23 and 24). This demonstrates the impact of QUIC's handshake since it only needs 1-RTT with unknown servers, while TCP+TLS' handshake requires, in general, 3-RTTs.

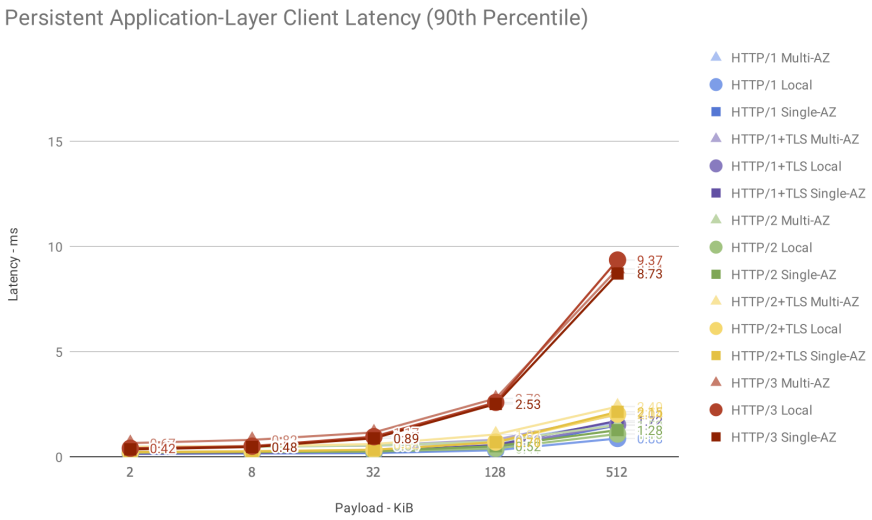

Figure 21: Persistent Application-Layer Client Latency (90th Percentile)

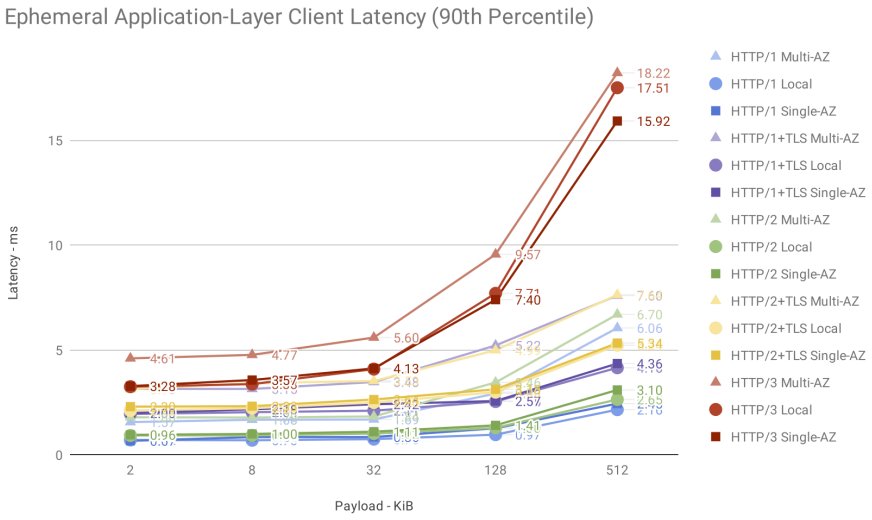

Figure 22: Ephemeral Application-Layer Client Latency (90th Percentile)

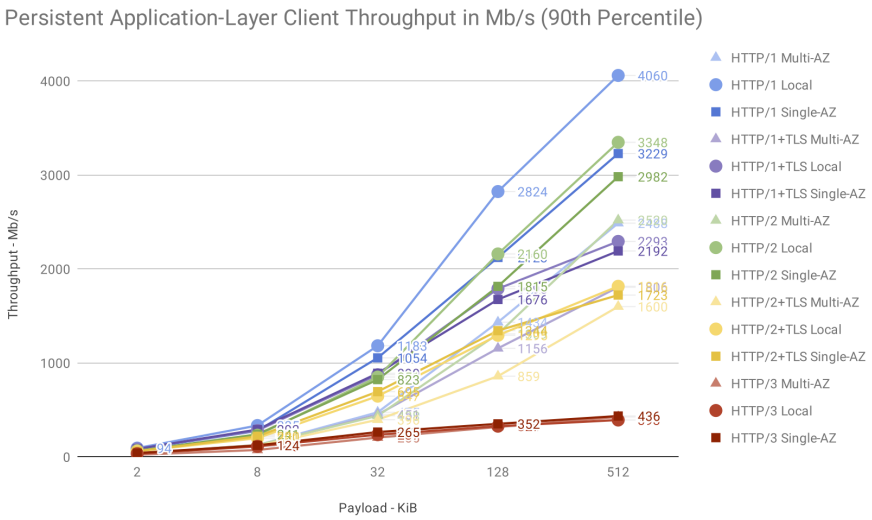

Figure 23: Persistent Application-Layer Client Throughput in Mb/s (90th Percentile)

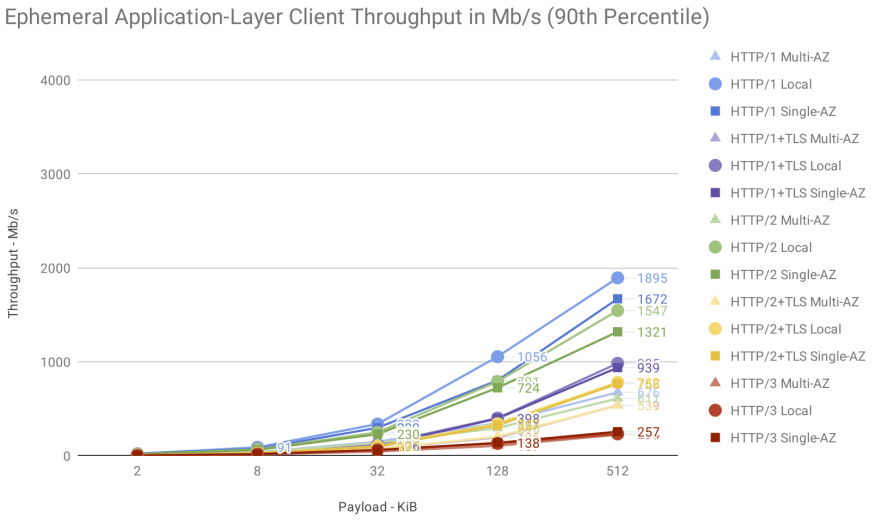

Figure 24: Ephemeral Application-Layer Client Throughput in Mb/s (90th Percentile)

5.2.2 CPU Usage

Charts on Figures 25, 26, 27, and 28 represent the CPU usage of clients and servers during ephemeral and persistent experiments.

Overall CPU Cost

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.

Servers CPU Cost

Servers CPU usage were usually mirrored with its clients. 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.

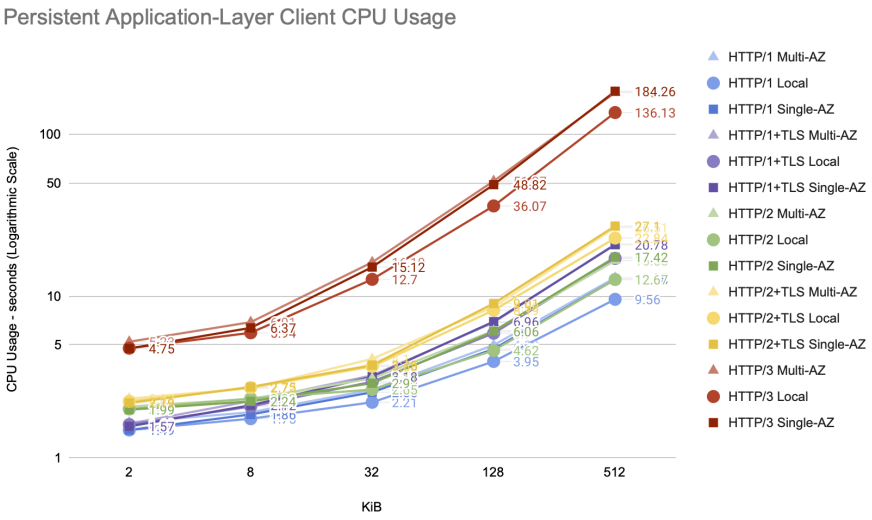

Figure 25: Persistent Application-Layer Client CPU Usage

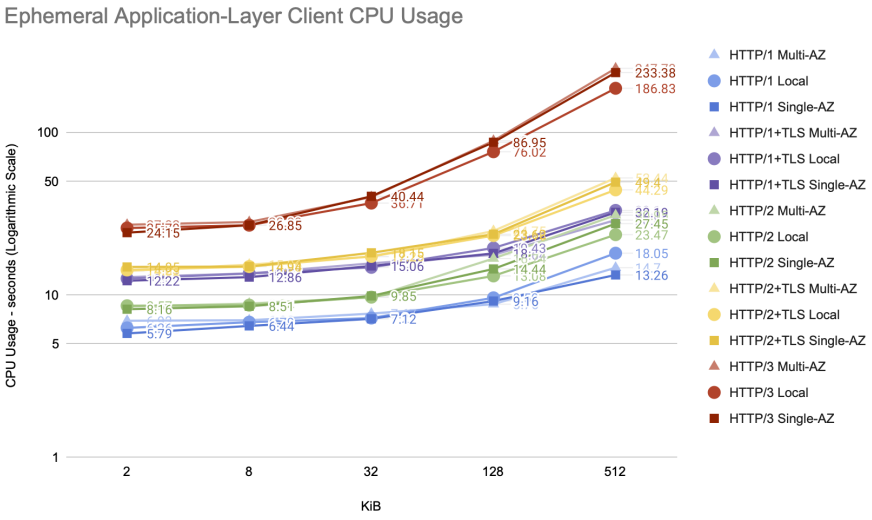

Figure 26: Ephemeral Application-Layer Client CPU Usage

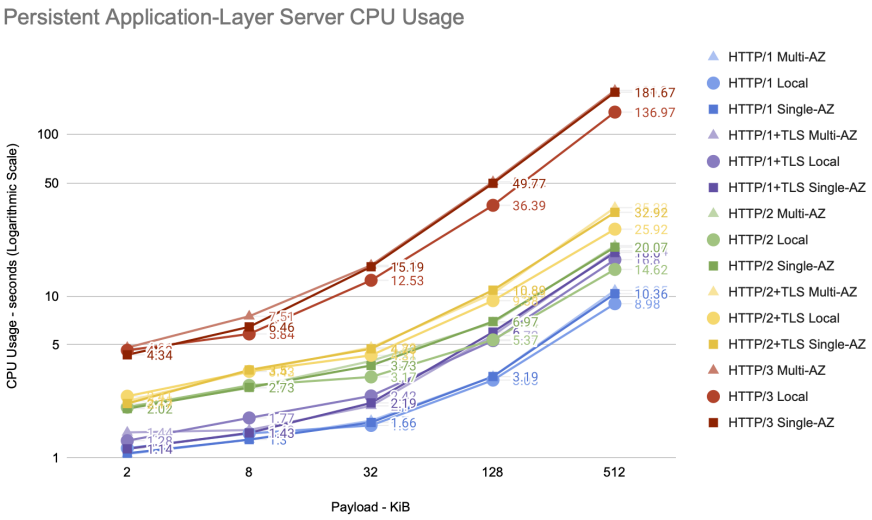

Figure 27: Persistent Application-Layer Server CPU Usage

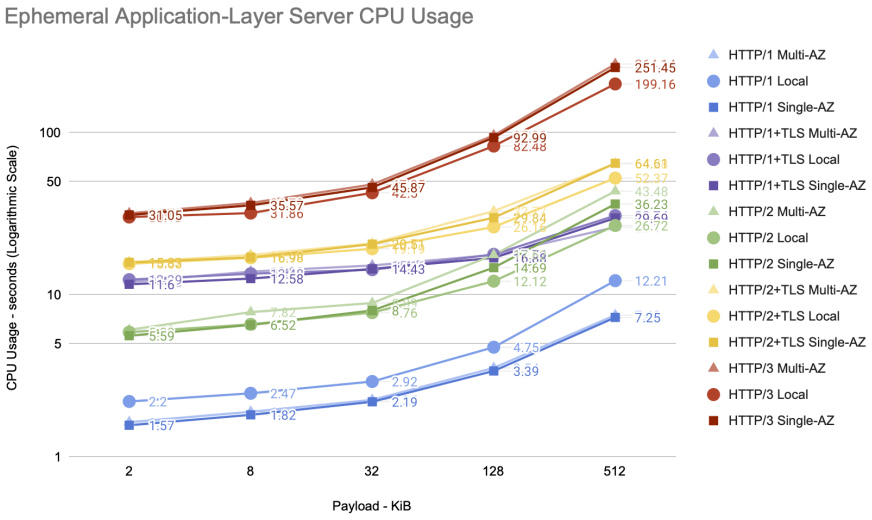

Figure 28: Ephemeral Application-Layer Server CPU Usage

5.2.3 Memory Usage

Charts on Figures 29, 30, 31, and 32 represent the memory usage of clients and servers during ephemeral and persistent experiments.

Overall Memory Usage

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.

HTTP/3 Unusual Ephemeral Client Memory Cost

During ephemeral experiments, Go's garbage collector also had problems dealing with the high rate of created clients, similar to QUIC's results during transport-layer experiments. Problem that begins worse with small payloads, 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.

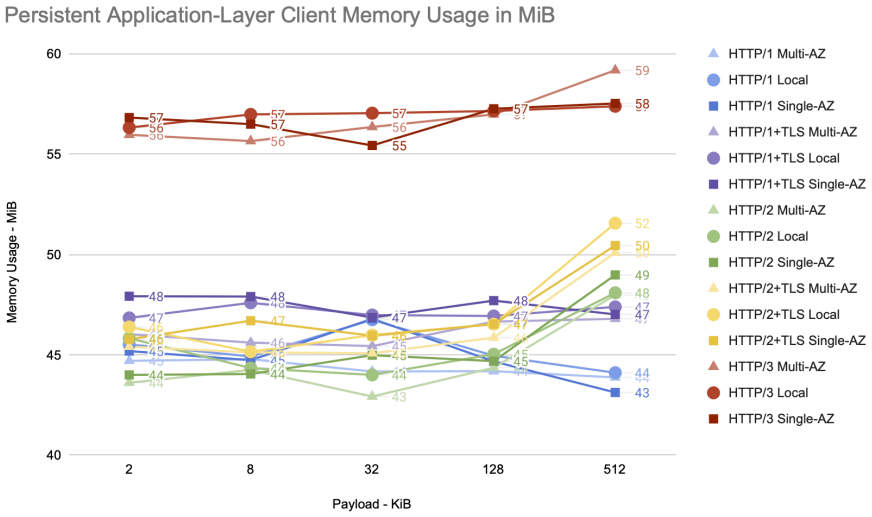

Figure 29: Persistent Application-Layer Client Memory Usage in MiB

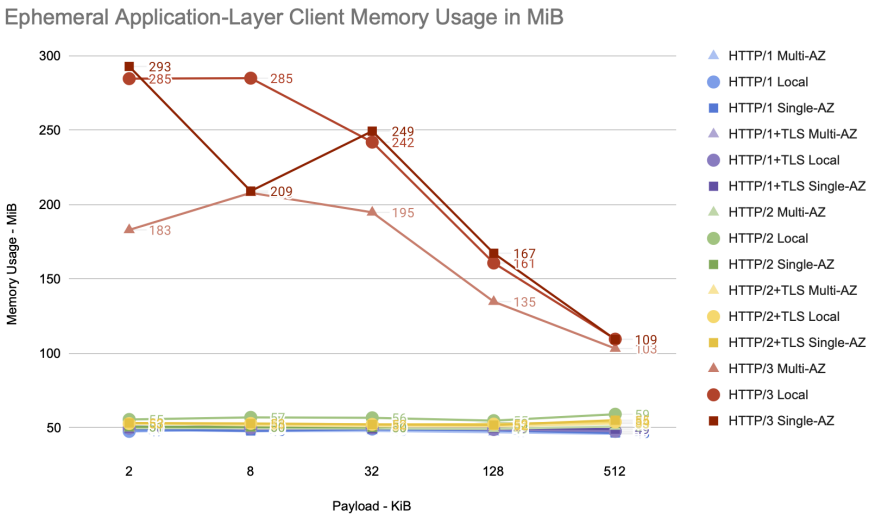

Figure 30: Ephemeral Application-Layer Client Memory Usage in MiB

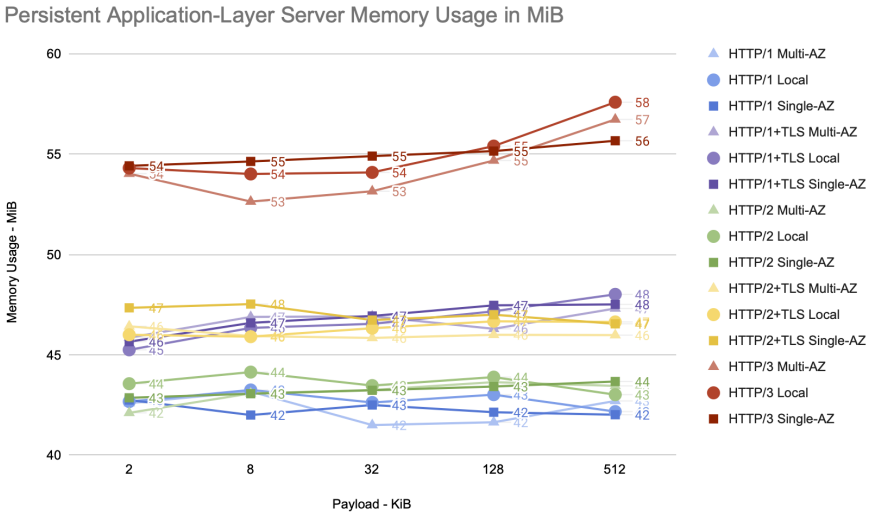

Figure 31: Persistent Application-Layer Server Memory Usage in MiB

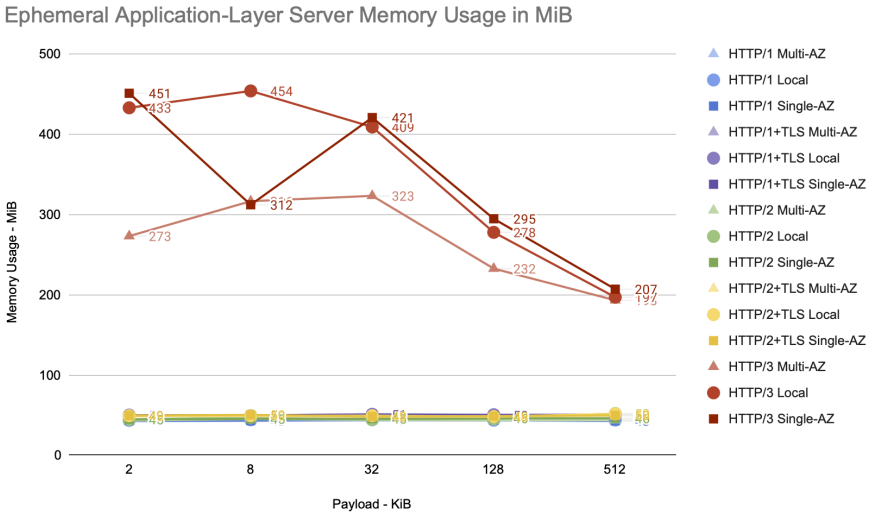

Figure 32: Ephemeral Application-Layer Server Memory Usage in MiB

5.3 Summary

This section explained the results obtained from ephemeral and persistent clients. It showed how each protocol performed when having to create multiple clients and establish multiple connections. Thus, the overhead related with creating connections is exposed.

The following section explores parallelism. Instead of creating new clients for each connection and performing requests sequentially, it performs multiple requests at the same time. Therefore, it experiments with each protocol concurrent features.

6 Parallel and Sequential Clients Experiments

As experiments were described in the *Experiments* section, this section contains all results regarding sequential persistent and parallel persistent clients, referred, respectively, as sequential and parallel clients.

It's divided into two subsections: transport-layer protocols and application-layer protocols. Each subsection is divided into three subsections: Latency & Throughput, CPU Usage, and Memory Usage. Consequently, all metrics collected are analysed.

6.1 Parallel and Sequential Transport Clients

All previous transport-layer protocols experiments performed requests sequentially, undermining the possible gain in efficiency of protocols that contain improvements to performing concurrent requests. Therefore, these experiments tries to explore this scenario by performing all 10000 requests within 100 goroutines.

6.1.1 Latency & Throughput

Charts on Figures 33, 34, 35 and 36 represent the P90 of Latency and Throughput of all 10000 requests made by the parallel and sequential clients during the experiments.

High Parallel Client Latency

As seen in the chart on Figures 33 and 34, both parallel and sequential clients possess a similar behavior. Their scale is differ, however.

Parallel clients results show a hundredfold difference. They run 100 goroutines at the same time, increasing the time each request takes to complete since segments are send in sequence due to only having one connection with the server. Consequently, latency increases 100 times since each goroutine needs to wait for the other 99 to complete their requests before sending any actual data.

Multi-AZ Higher Throughput

As performing Multi-AZ requests have increased latency when compared to others, sequential clients had their throughput affected during experiments. Nonetheless, parallel experiments were not affected by higher latency. Parallel requests do have a higher latency when compared individually, but latency is not the bottleneck.

As many requests are made at the same time, the throughput is defined by how fast requests leave the client and server's response time. Therefore, latency does not impact as much as it did during sequential experiments. Consequently, all three scenarios, Local, Same-AZ, and Multi-AZ, have a very similar result in their respective protocol.

Throughput Parallel Upper Limit

During Parallel clients experiments TCP and TCP+TLS protocols throughput results were very similar. While sequential experiments throughputs showed these protocols can have a better performance, parallel requests seem to have a upper limit to how well they can perform.

Sequential experiments requests always have the client and server completely available to them, resulting in a immediate action when the client wants to send a request and when the server wants to send a response. Parallel experiments requests don't have that luxury, however. Multiple requests are made at the same time, requiring the client to queue requests coming out and responses coming in, the same happens to the server. Due to the amount of parallel requests, this queuing delay is enough to degrade throughput.

UDP Linux Restrictions

QUIC's performance improved a bit overall when requests were made in parallel. Apart from the improvement in the Multi-AZ scenario, it remained roughly the same, however. This happens due to Linux restrictions to UDP performance [14].

The UDP receive buffer is responsible for holding packets that have been received by the kernel, but still were not read by the application [30]. Once it fills up, the kernel will drop any new incoming packets. Usually, the UDP receive buffer size is 128KiB, enough for only 1 microsecond of data on a megabit network [14].

Upon receiving a warning from QUIC's Go Implementation about the small buffer size, it was increased to 26MiB, more than the recommended amount, 2.5MB. Even though the buffer was increased significantly, QUIC still reached an upper throughput limit of 1Gb/s, explaining why both sequential and parallel experiments with 128KiB and 512KiB payloads resulted in almost the same value

UDP Failure

During sequential experiments UDP succeeded during Local scenario throughout all packet sizes, failing in Same-AZ and Multi-AZ scenarios from 32 KiB payloads onward. Parallels experiments failed on all 32 KiB payloads onward, however. As 100 requests are performed at the same time, the UDP receive buffer from the server fills up faster than the application can read incoming messages, resulting in the kernel dropping any new incoming messages. Sequential experiments succeeded on Local scenario since they never overflow the UDP receive buffer, as only one request is performed at a time.

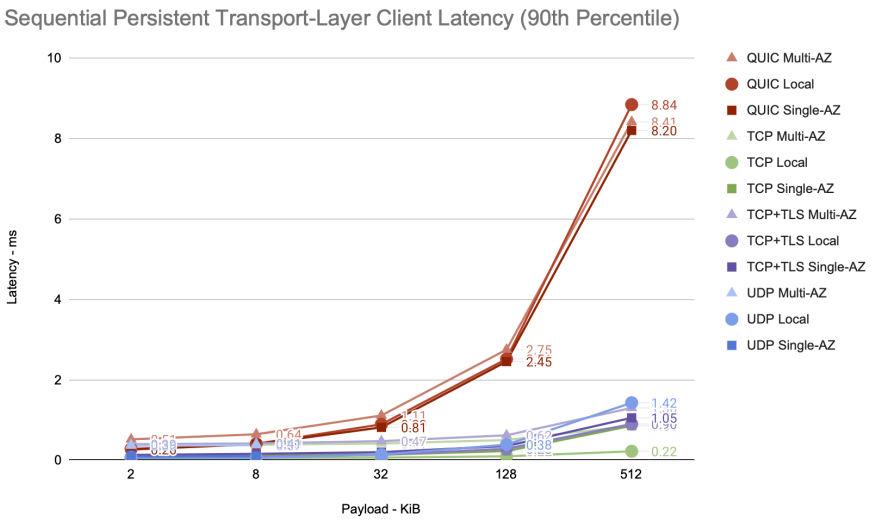

Figure 33: Sequential Transport-Layer Client Latency (90th Percentile)

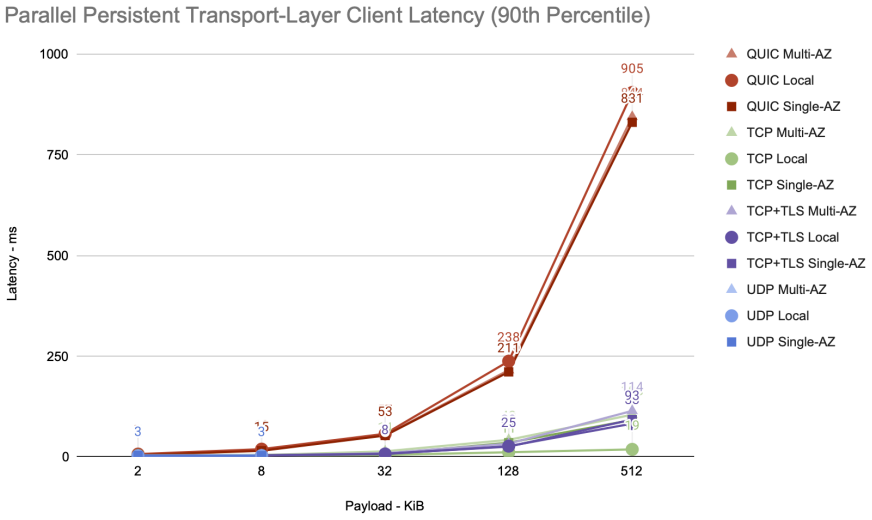

Figure 34: Parallel Transport-Layer Client Latency (90th Percentile)

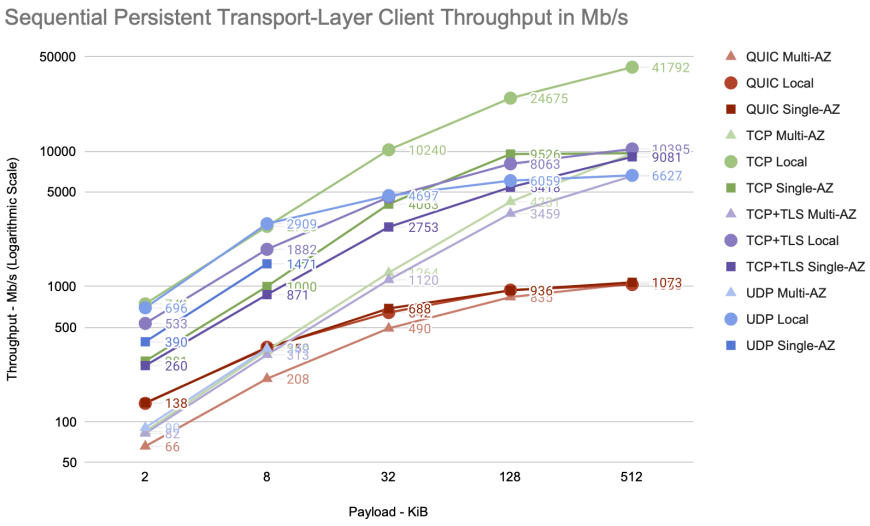

Figure 35: Sequential Persistent Transport-Layer Client Throughput in Mb/s

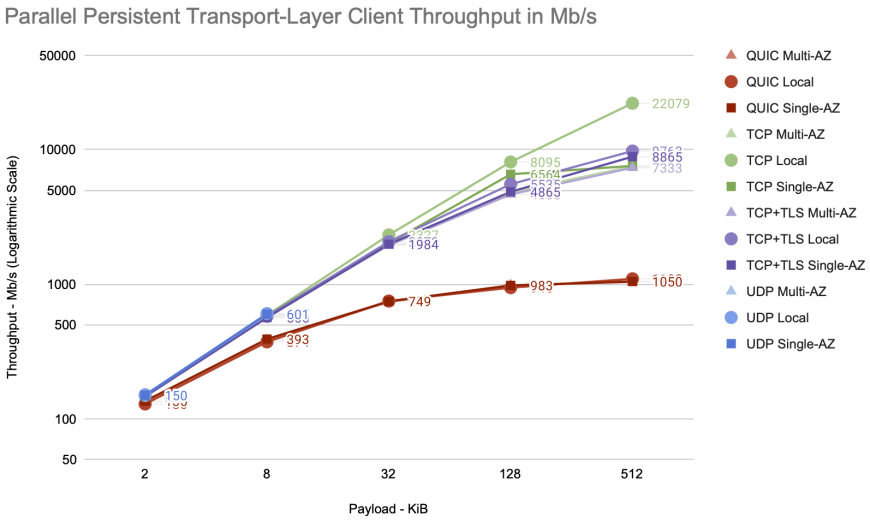

Figure 36: Parallel Persistent Transport-Layer Client Throughput in Mb/s

6.1.2 CPU Usage

Charts on Figures 37, 38, 39, and 40 represent the CPU usage of clients and servers during ephemeral and persistent experiments.

Overall Clients CPU Usage

Parallel requests with 32KiB and smaller payloads had a lower CPU usage when compared with sequential requests. Performing the former waits longer for each response to arrive, requiring more CPU time. Thus, small concurrent requests demands less CPU time.

Requests with 128KiB and larger payloads starts to shift this behavior. As larger payloads needs to be processed at the same time, it requires more CPU time than when processing one at a time. Therefore, when dealing with larger payloads, sequential requests spend less CPU time than parallel.

QUIC's CPU Usage

QUIC's CPU is greater than other protocols when requests are concurrent. It needs to deal with cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state. Thus, requiring more CPU time.

Overall Server CPU Usage

Server CPU usage remained roughly the same as client CPU usage. Requests and responses payloads have the same size. Therefore, clients and servers have to process the same amount of requests and responses, which explains why their CPU usage is so similar.

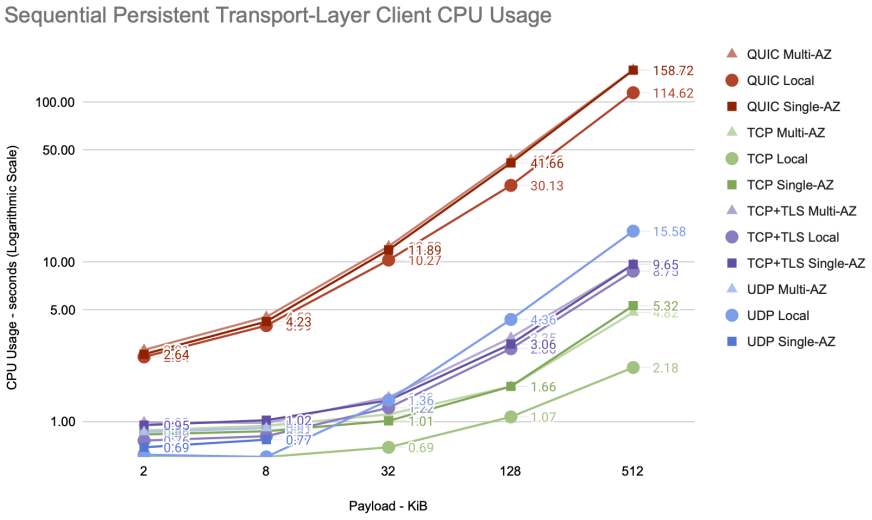

Figure 37: Sequential Persistent Transport-Layer Client CPU Usage

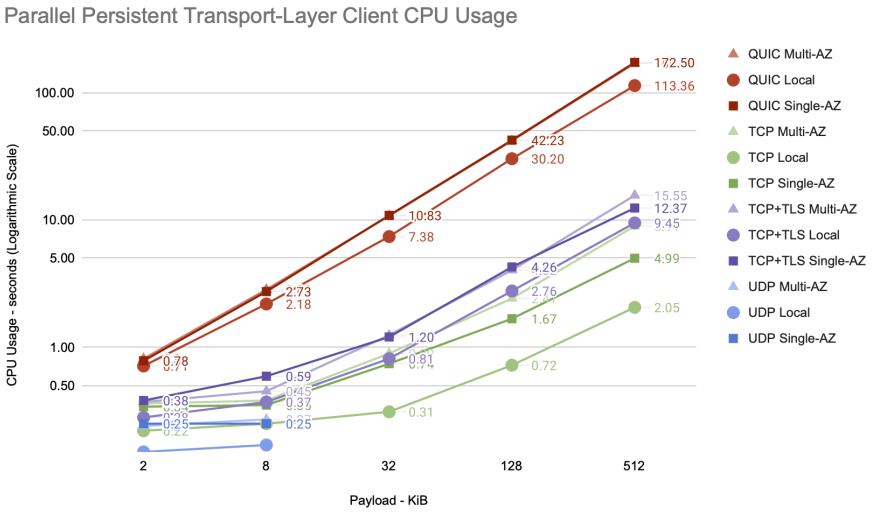

Figure 38: Parallel Persistent Transport-Layer Client CPU Usage

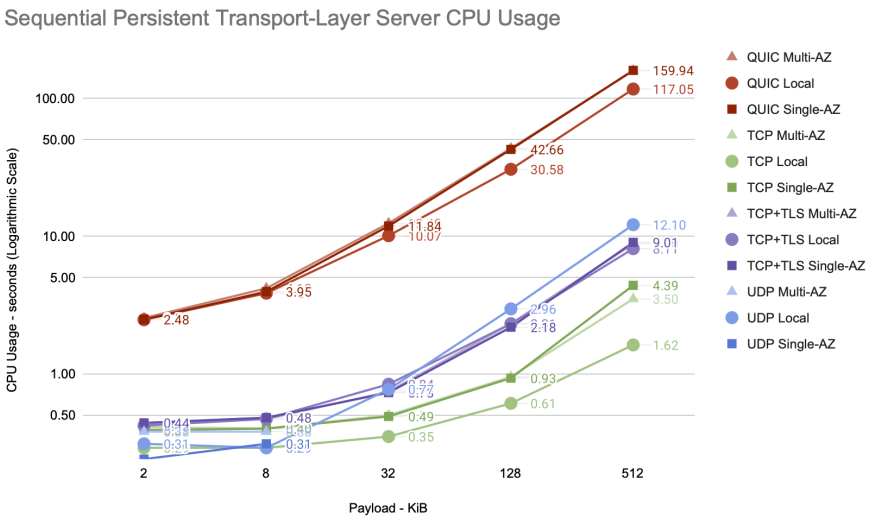

Figure 39: Sequential Persistent Transport-Layer Server CPU Usage

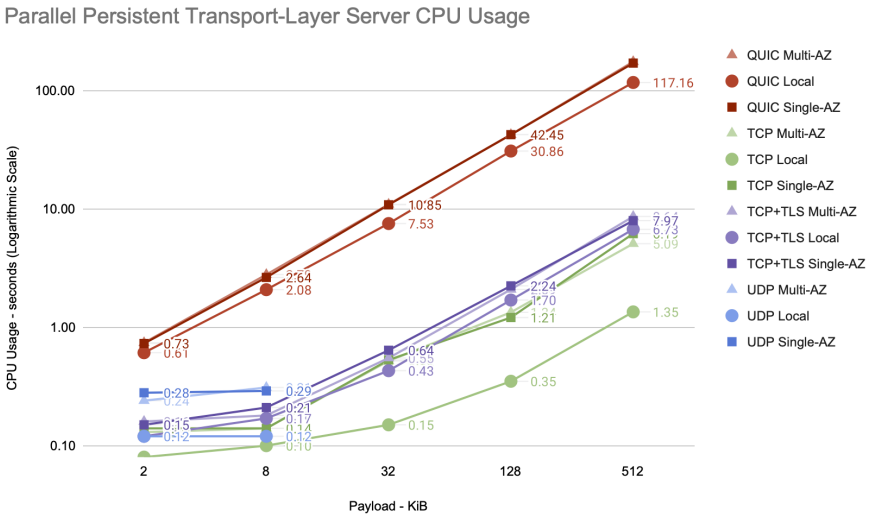

Figure 40: Parallel Persistent Transport-Layer Server CPU Usage

6.1.3 Memory Usage

Charts on Figures 41, 42, 43, and 44 represent the memory usage of clients and servers during sequential and parallel experiments.

Overall Transport Client Memory Usage

Parallel requests demands more memory than sequential requests. While concurrent requests needs to kep more information in memory at the same time, sequential requests only have to deal with one requests at a time. Thus, the slight increase on memory usage by parallel requests.

TLS Memory Usage

TCP+TLS and QUIC used more memory than TCP. They have to implement all features required by TLS, such as TLS handshake and encryption. Thus, demanding more memory to perform these operations in exchange for encrypted payload.

QUIC used more memory when compared to TCP+TLS, however. It has to deal with more features beyond TLS' requirements, such as sending and receiving UDP packets, and maintaining the QUIC state. Thus, it has increased memory usage.

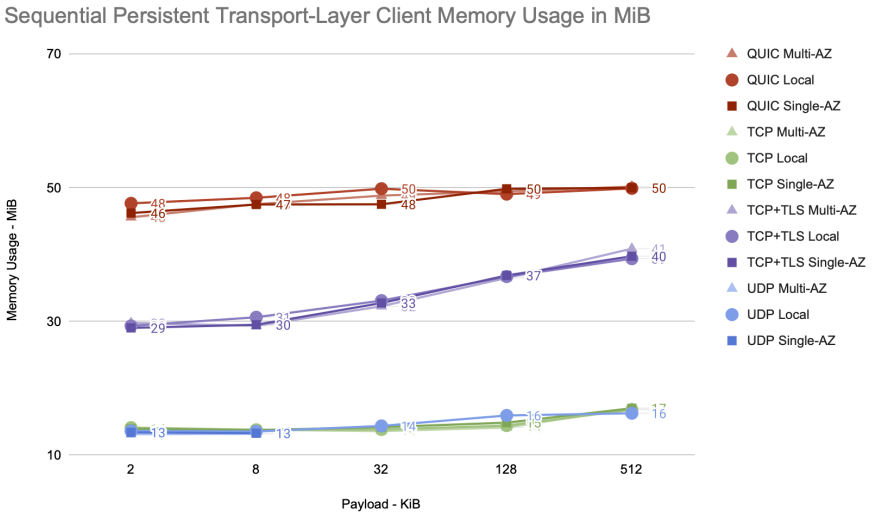

Figure 41: Sequential Persistent Transport-Layer Client Memory Usage in MiB

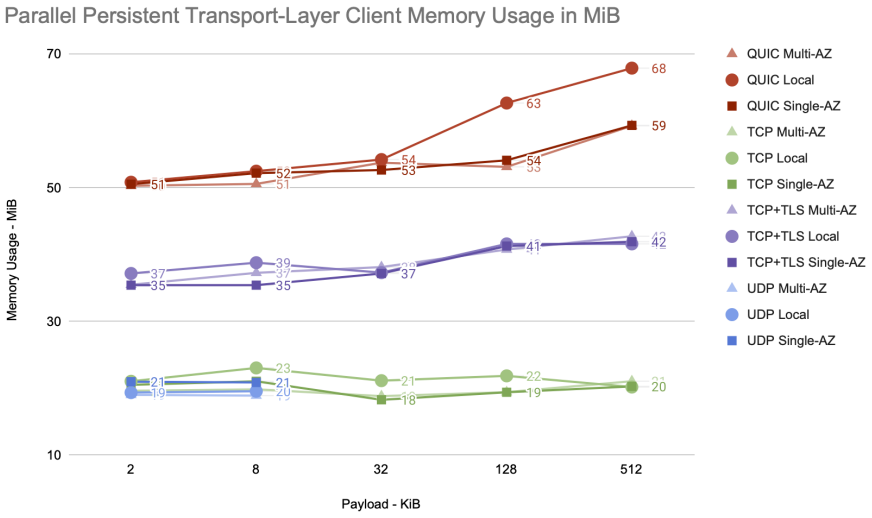

Figure 42: Parallel Persistent Transport-Layer Client Memory Usage in MiB

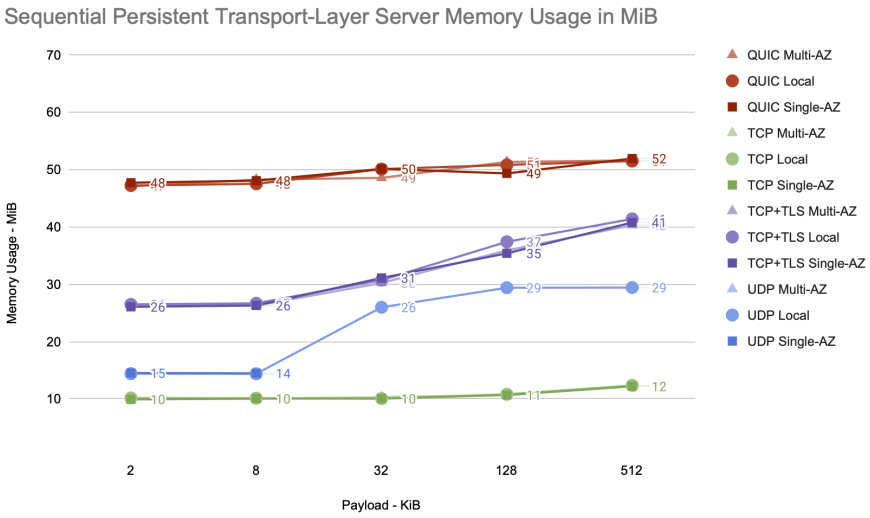

Figure 43: Sequential Persistent Transport-Layer Server Memory Usage in MiB

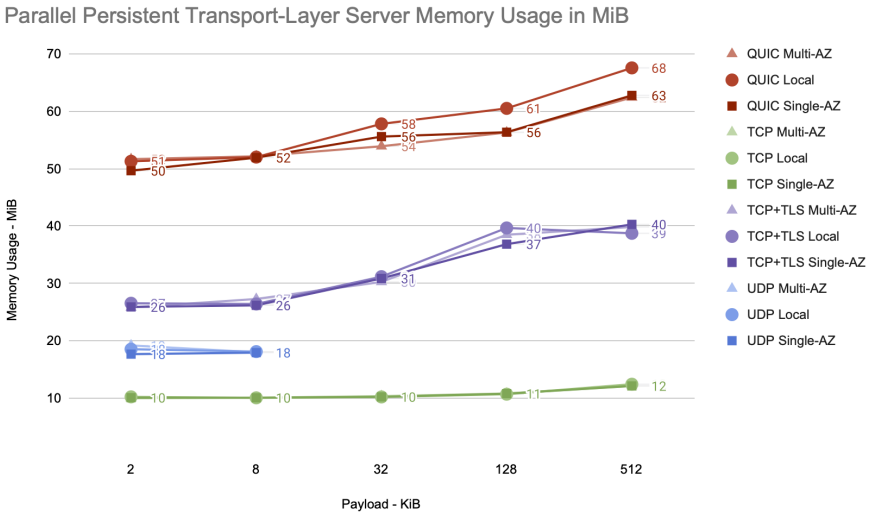

Figure 44: Parallel Persistent Transport-Layer Server Memory Usage in MiB

6.2 Parallel and Sequential Application Clients

All previous application-layer protocols experiments performed requests sequentially, undermining the possible gain in efficiency of protocols that contain improvements to performing concurrent requests. Therefore, these experiments tries to explore this scenario by performing all 10000 requests within 100 goroutines.

6.2.1 Latency & Throughput

Charts on Figures 46, 47, 48 and 49 represent the P90 of Latency and Throughput of all 10000 requests made by the sequential and parallel clients during the experiments.

High Parallel Client Latency

The chart on Figures 46 and 47 shows latency has a similar behavior on both sequential and parallel clients, but their scale differ. Parallel clients show a hundredfold difference, as was also previously observed during transport-layer protocols experiments. This is a reflection of running 100 goroutines concurrently, requests takes longer to finish since each goroutine's request needs to wait for the other 99 requests to be processed by the host before it can be processed itself.

HTTP/1 performed better than HTTP/2

During sequential experiments, HTTP/1 had a better latency and throughput than HTTP/2. Neither HTTP/1 pipelining nor HTTP/2 multiplexing had a big role during these experiments since they only improves parallel requests. Nonetheless, during parallel experiments, HTTP/1 still performed better than HTTP/2.

HTTP/1 pipelining allows clients to perform multiple requests in parallel [18], improving the overall time required to finish more than one request. However, responses must arrive in the same order as requests were made, therefore the first request must receive its response before the second does. This limits its performance since a slow response can delay all other requests. HTTP/2 multiplexing solves this by allowing responses to arrive in any order (Figure 45).

As all request and response payloads had the same size during experiments, HTTP/2 multiplexing did not have an impact on throughput. Thus, HTTP/1 kept being the protocol with best performance due to its simplicity, while HTTP/2 needs to deal with compressing its requests headers.

Multi-AZ Higher Throughput

As previously observed during Transport-Layer protocols, while latency affects sequential clients, it does not impact parallel clients as much. Parallel requests

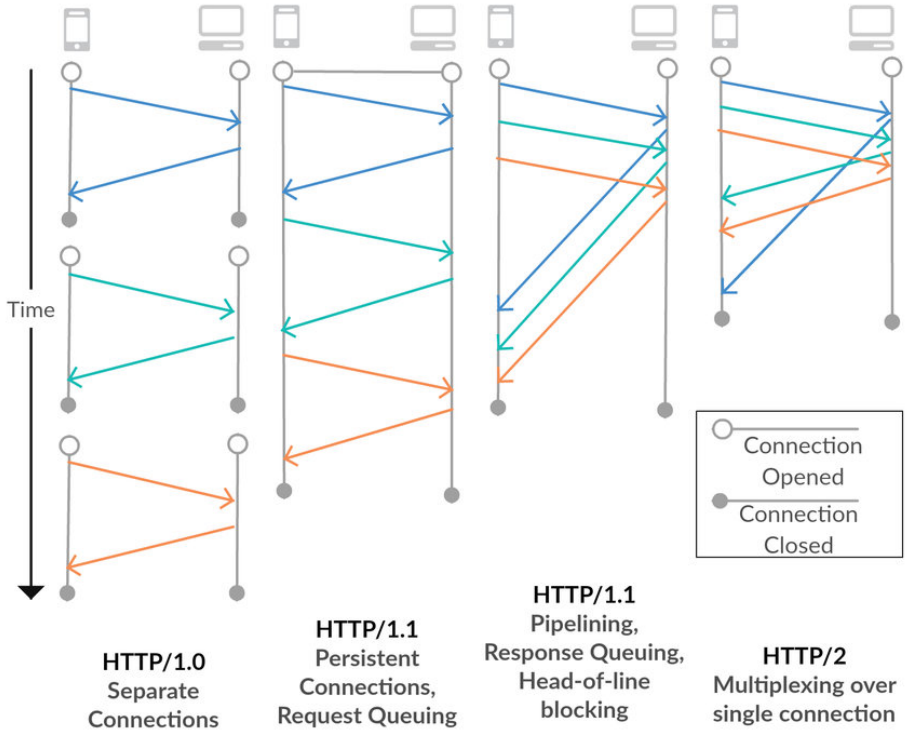

Figure 45: Comparison of HTTP Versions
Source: [19]

improved all application protocols' throughput since requests are performed simultaneously. Therefore, the bottleneck remains how fast requests leave the client and server's response time, and not latency.

Throughput Parallel Upper Limit

The throughput upper limit seen during transport-layer protocols experiments remained during application-layer experiments for all protocols but HTTP/3. Sequential client experiments also had client and server completely free, resulting in immediate request processing by the client and a faster response by the server. However, parallel requests queuing delay problem remained, degrading throughput due to the time requests and responses needs to wait to be processed.

While the previous stated behaviour was true to 128KiB payloads and smaller, parallel requests with 512KiB payloads had a better throughput than sequential requests. Sending larger payloads concurrently benefits surpasses the queuing delay drawbacks. Thus, processing larger payloads results in increased throughput.

HTTP/3 Limits

HTTP/3's throughput had an overall increase when performing parallel requests. It was not significant, however. The UDP receive buffer performance degradation still impacts HTTP/3 as it did with QUIC, preventing HTTP/3 to perform beyond 1Gb/s.

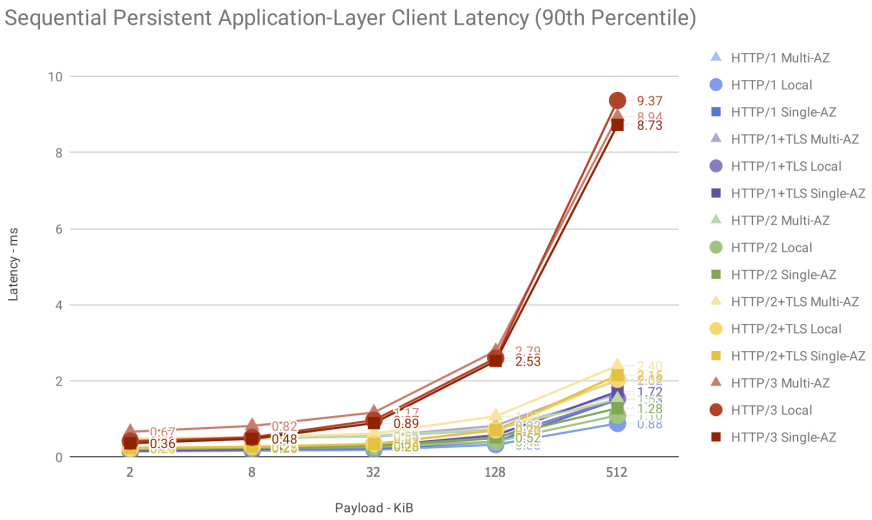

Figure 46: Sequential Persistent Application-Layer Client Latency (90th Percentile)

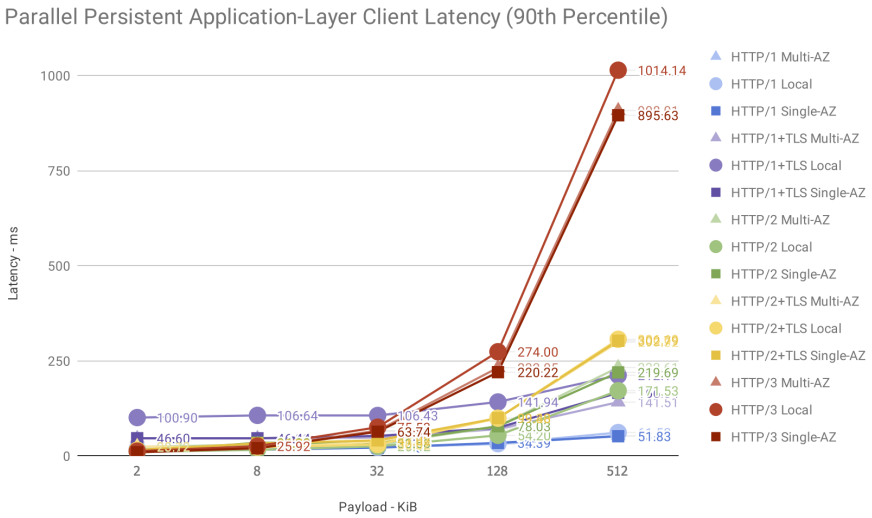

Figure 47: Parallel Persistent Application-Layer Client Latency (90th Percentile)

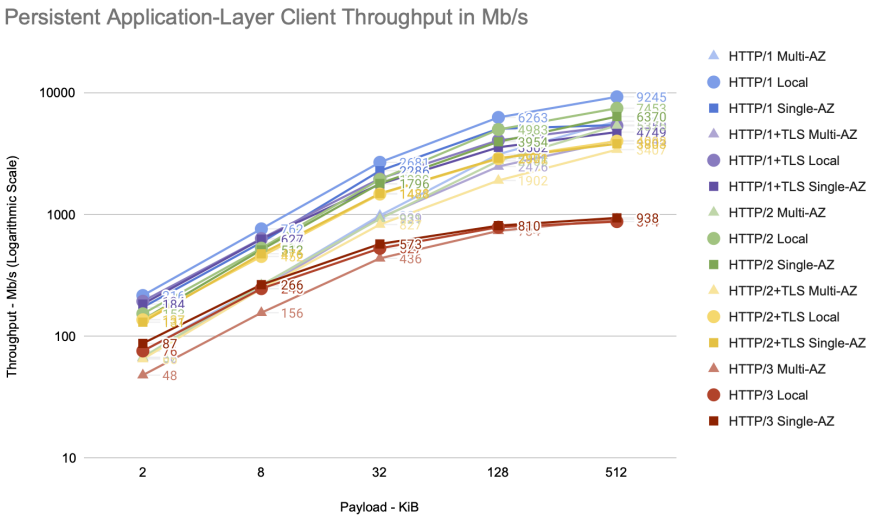

Figure 48: Persistent Application-Layer Client Throughput in Mb/s

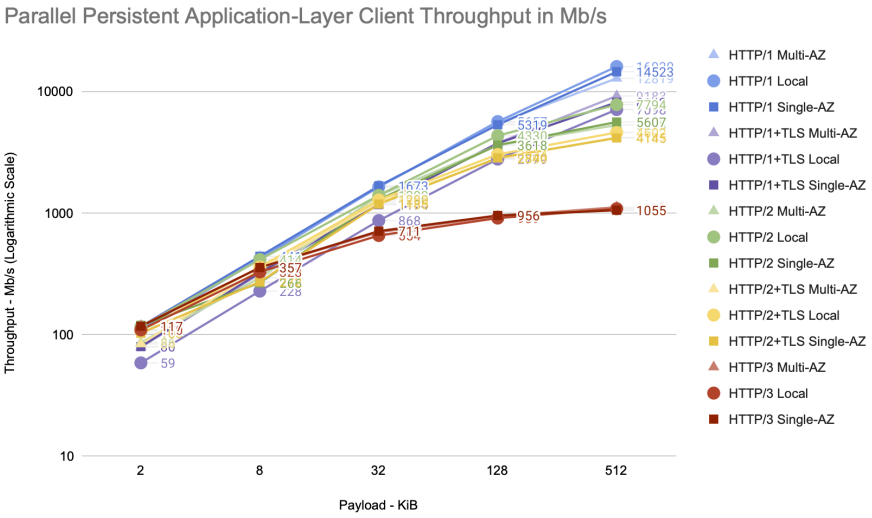

Figure 49: Parallel Persistent Application-Layer Client Throughput in Mb/s

6.2.2 CPU Usage

Charts on Figures 50, 51, 52, and 53 represent the CPU usage of clients and servers during ephemeral and persistent experiments.

Overall Clients CPU Usage

Application-layer protocols CPU Usage results behavior is similar to transport-layer's. Parallel requests with 32KiB and smaller payloads had a lower CPU usage when compared to sequential requests. And as requests payloads reaches 128KiB size, parallel requests CPU usage surpasses sequential requests.

HTTP/3's CPU Usage

QUIC's CPU Usage (Figure 38) is almost the same as HTTP/3's (Figure 51). As QUIC performs shift features that were usually implemented in the application-layer, it does most of the work necessary to manage traffic. Therefore, HTTP/3 is only an interface so applications can still use it as any other HTTP protocol.

Overall Server CPU Usage

As other experiments, Server CPU usage remained roughly the same as client CPU usage. Client and servers still need to process the same amount of requests and responses, which explains why their CPU usage is so similar.

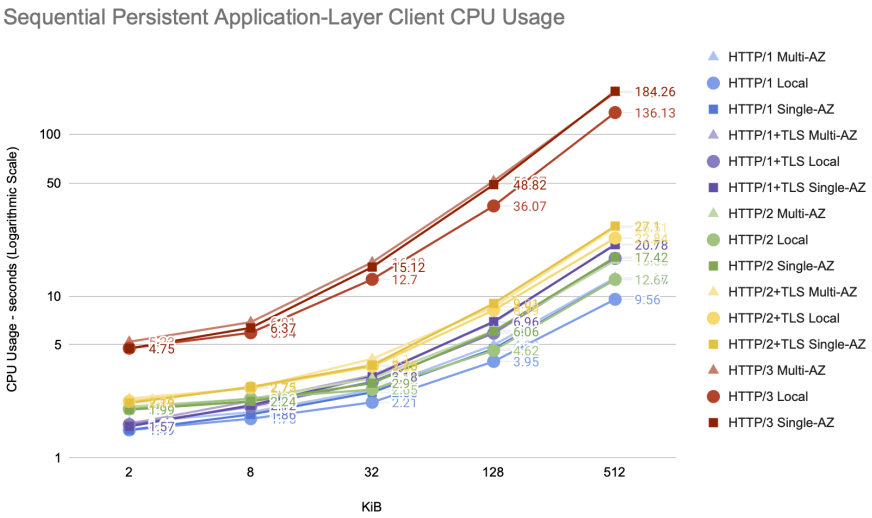

Figure 50: Sequential Persistent Application-Layer Client CPU Usage

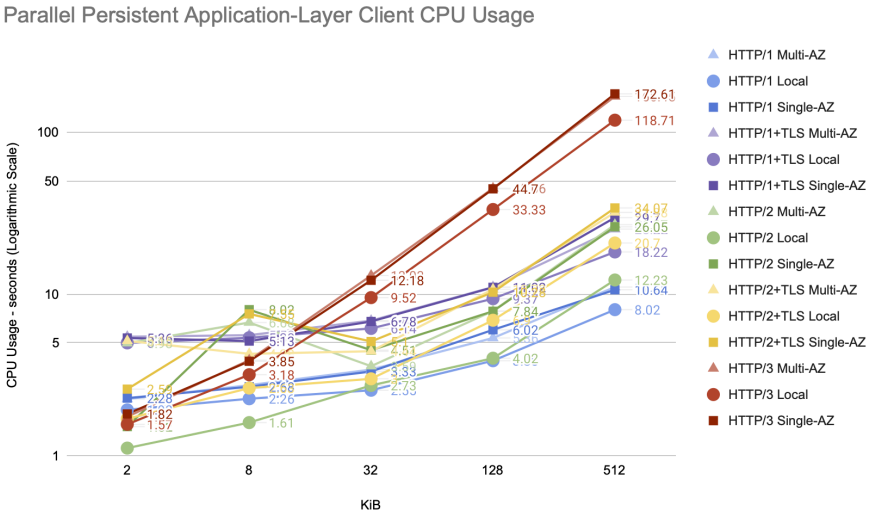

Figure 51: Parallel Persistent Application-Layer Client CPU Usage

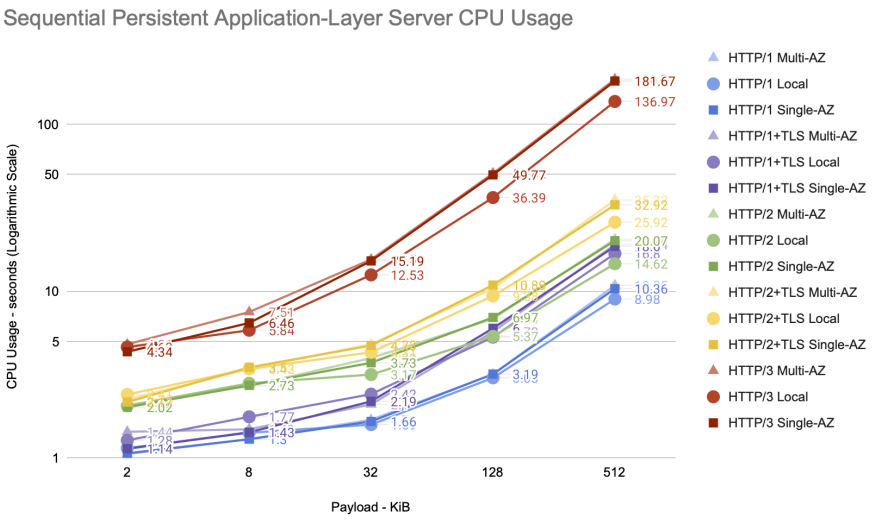

Figure 52: Sequential Persistent Application-Layer Server CPU Usage

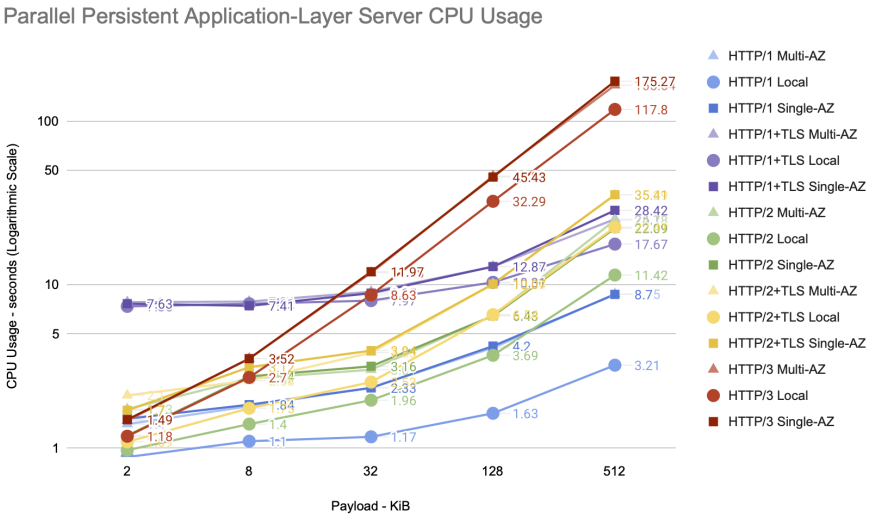

Figure 53: Parallel Persistent Application-Layer Server CPU Usage

6.2.3 Memory Usage

Charts on Figures 54, 55, 56, and 57 represent the memory usage of clients and servers during sequential and parallel experiments.

Overall Application Client Memory Usage

HTTP/1+TLS, HTTP/2 and HTTP/2+TLS managed to require more memory than HTTP/3 during parallel experiments. The latter remained as most costly during sequential experiments, however.

HTTP/1+TLS Memory Usage

HTTP/1+TLS required more memory when performing parallel requests. HTTP/1 relies on TCP to manage its HTTP requests, resulting in a low memory consumption. Therefore, HTTP/1+TLS' memory usage is related to TLS' required features to be able to deliver encrypted data.

HTTP/2 Memory Usage

HTTP/2 parallel requests managed to use a lot of memory when compared to sequential requests. HTTP/2 multiplexing control flow allocates buffers for each one of its streams [5]. Thus, resulting in a large amount of allocated memory.

This behavior gets worse as payload size increase. It demands more space to be able to maintain all requests in the buffer while they are sent to the server. Therefore, this happens as the server cannot keep up with the data being transferred, taking longer to remove the HTTP request from the buffer. Consequently, the buffer grows even more, requiring more memory.

Overall Application Server Memory Usage

While sequential server's memory usage remained similar to its client, parallel server's differs from its client.

HTTP/2 buffer did not impact as much as it did on its client. This buffer is used by the client to send data to the server. Thus, since the server only needs to respond requests that were sent by the client, it results in it never having to perform a large amount of concurrent requests.

Other than HTTP/2's memory usage, other protocols required approximately the same as its clients.

HTTP/3's Memory Usage

HTTP/3's memory (Figure 55) usage was very similar to QUIC's (Figure 42). Unlikely HTTP/2, QUIC only have UDP receive buffer, which in our case was 26MiB in size. Therefore, all of its memory usage is related cryptography, handling UDP packets, and maintaining its internal state.

Parallel requests memory usage was a bit higher when compared to sequential requests. Concurrent requests requires QUIC to maintain status and encrypt more requests than usual. For that reason it requires a bit more of memory.

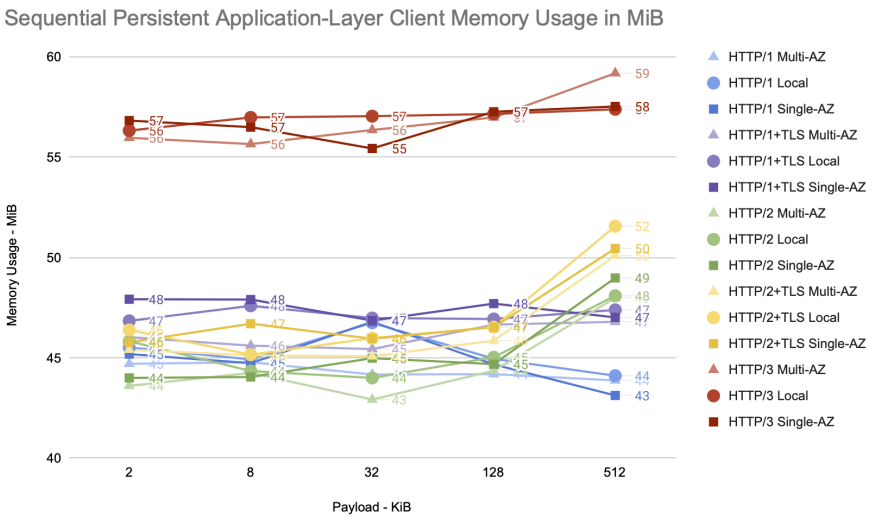

Figure 54: Sequential Persistent Application-Layer Client Memory Usage in MiB

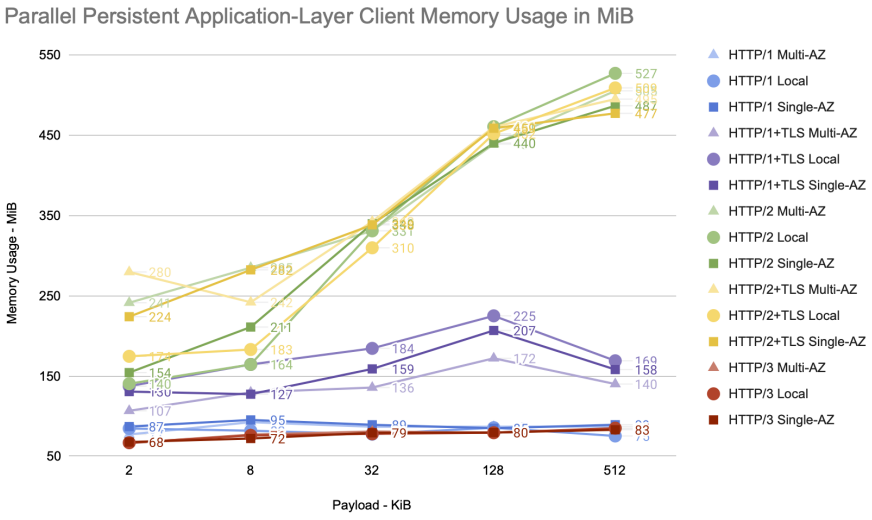

Figure 55: Parallel Persistent Application-Layer Client Memory Usage in MiB

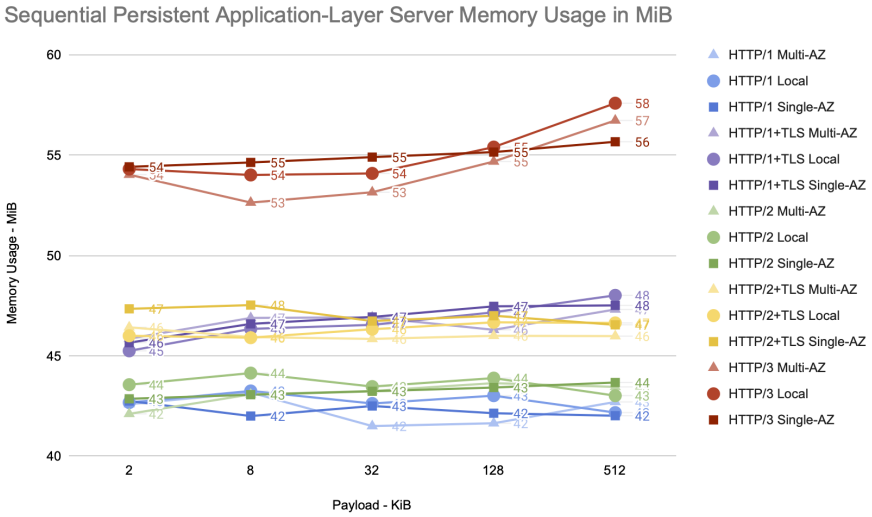

Figure 56: Sequential Persistent Application-Layer Server Memory Usage in MiB

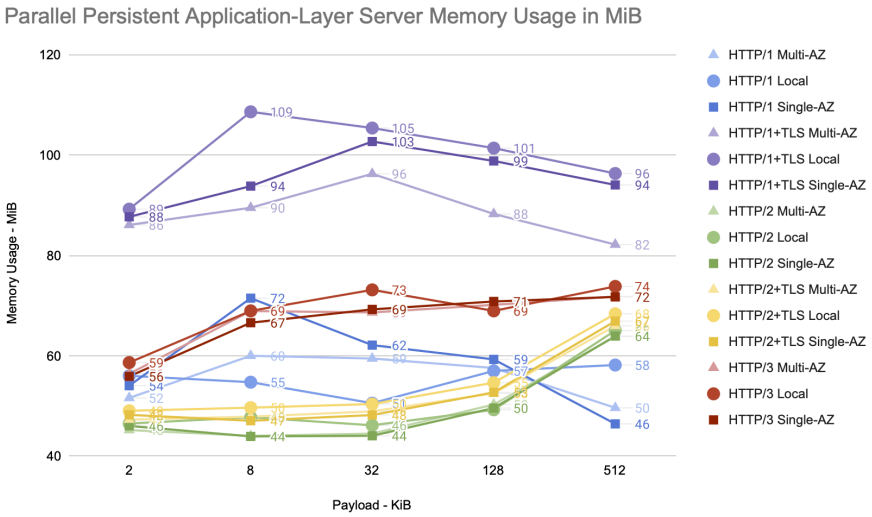

Figure 57: Parallel Persistent Application-Layer Server Memory Usage in MiB

6.3 Summary

This section explained the results obtained from parallel and sequential clients. It showed how each protocol performed when having to perform multiple requests at the same time with a persistent client. Thus, each protocol concurrent optimization is exposed.

7 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] Fahad Al-Dhief, Naseer Sabri, Nurul Muazzah Abdul Latiff, Nik Noordini Nik Abd Malik, Mohd Khanapi Abd Ghani, Mazin Mohammed, R.N. Al-Haddad, Yasir Dawood, Mohd Ghani, and Omar Ibrahim Obaid. Performance comparison between tcp and udp protocols in different simulation scenarios. *International Journal of Engineering Technology*, 7:172–176, 01 2018.

[2] Amazon Web Services. "Amazon EC2 M6i Instances". `https://aws.amazon.com/ec2/instance-types/m6i/`.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[4] AT&T. "What Is The Cloud". `https://youtu.be/_a7hK6kWttE`.

[5] M. Belshe. "Hypertext Transfer Protocol Version 2 (HTTP/2)", May 2015. `https://rfc-editor.org/rfc/rfc7540.txt`.

[6] T. Berners-Lee. "Hypertext Transfer Protocol – HTTP/1.0", May 1996. `https://rfc-editor.org/rfc/rfc1945.txt`.

[7] B. Carpenter. "Middleboxes: Taxonomy and Issues", February 2002. `https://rfc-editor.org/rfc/rfc3234.txt`.

[8] Y. Cheng. "TCP Fast Open", December 2014. `https://rfc-editor.org/rfc/rfc7413.txt`.

[9] Martin P. Clark. *Data Networks, IP and the Internet: Networks, Protocols, Design and Operation*. John Wiley & Sons, Inc., USA, 2003.

[10] Cloud Native Computing Foundation. "CNCF Kubernetes Project Journey Report". `https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Kubernetes_Project_Journey_Report.pdf`.

[11] Cloudflare. "What is a TLS Handshake?". `https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/`.

[12] Cloudflare. "What is HTTP?". `https://www.cloudflare.com/en-ca/learning/ddos/glossary/hypertext-transfer-protocol-http/`.

[13] Eric Conrad, Seth Misenar, and Joshua Feldman. Chapter 3 - domain 2: Telecommunications and network security. In Eric Conrad, Seth Misenar, and Joshua Feldman, editors, *CISSP Study Guide (Second Edition)*, pages 63–141. Syngress, Boston, second edition edition, 2012.

[14] Data Expedition. "Linux Performance Tuning". `https://www.dataexpedition.com/support/notes/tn0035.html`.

[15] T. Dierks. "The Transport Layer Security (TLS) Protocol Version 1.2", August 2008. https://rfc-editor.org/rfc/rfc5246.txt.

[16] Docker. "Are Containers Replacing Virtual Machines?". https://peering.google.com/#/learn-more/quic.

[17] Gregorio Díaz, Fernando Cuartero, Valentín Valero, and Fernando Pelayo. Automatic verification of the tls handshake protocol. pages 789–794, 01 2004.

[18] R. Fielding. "Hypertext Transfer Protocol – HTTP/1.1", June 1999. https://rfc-editor.org/rfc/rfc2616.txt.

[19] Gen Murasaki. "HTTP Version Comparison". https://blog.caoyu.info/http-version.html.

[20] Google. "Google Edge Network". https://peering.google.com/#/learn-more/quic.

[21] R. Hinden. "IP Version 6 Addressing Architecture", February 2006. https://rfc-editor.org/rfc/rfc4291.txt.

[22] J. Iyengar. "QUIC: A UDP-Based Multiplexed and Secure Transport", May 2021. https://rfc-editor.org/rfc/rfc9000.txt.

[23] Adam Langley, Al Riddoch, Alyssa Wilk, Antonio Vicente, Charles 'Buck' Krasic, Cherie Shi, Dan Zhang, Fan Yang, Feodor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Christopher Dorfman, Jim Roskind, Joanna Kulik, Patrik Göran Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, and Wan-Teh Chang. The quic transport protocol: Design and internet-scale deployment. 2017.

[24] V. Paxson. "Computing TCP's Retransmission Timer", November 2000. https://rfc-editor.org/rfc/rfc2988.txt.

[25] J. Postel. "User Datagram Protocol", August 1980. https://rfc-editor.org/rfc/rfc768.txt.

[26] J. Postel. "Transmission Control Protocol", September 1981. https://rfc-editor.org/rfc/rfc793.txt.

[27] E. Rescorla. "HTTP Over TLS", May 2000. https://rfc-editor.org/rfc/rfc2818.txt.

[28] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., USA, 2006.

[29] The Kubernetes Authors. "Jobs". https://kubernetes.io/docs/concepts/workloads/controllers/job/.

[30] The quic-go Authors. "UDP-Receive-Buffer-Size". `https://github.com/lucas-clemente/quic-go/wiki/UDP-Receive-Buffer-Size`.

[31] L Zhang. Why tcp timers don't work well. *SIGCOMM Comput. Commun. Rev.*, 16(3):397–405, aug 1986.

Acronyms

ACK Acknowledge 7, 12, 13

AWS Amazon Web Services 18, 21

AZ Availability Zone 18, 21–24, 26, 29, 35

CA certificate authority 9

CNCF Cloud Native Computing Foundation 17

CPU Central Processing Unit 13, 16, 22, 29, 39, 50, 61

EC2 Elastic Cloud Computing 21, 22

EKS Elastic Kubernetes Service 18, 21

HTML HyperText Markup Language 10

HTTP Hyper Transfer Protocol 10–13, 35, 39, 42

HTTPS Hypertext Transfer Protocol Secure 11, 12

IaaS Infrastructure as a Service 4, 15, 16

K8s Kubernetes 6, 17, 18, 20–22

MIME Multipurpose Internet Mail Extensions 10

OS Operating System 11, 16

P90 90th Percentile 24, 35, 46, 56

PaaS Platform as a Service 15

RFC Request for Comments 13

RTT Round-Trip Times 9, 12, 13, 25, 26, 36

SaaS Software as a Service 16

SYN Synchronize 8

SYN/ACK Synchronize/Acknowledge 8

TCP Transmission Control Protocol 7–9, 11–13, 19, 25, 26, 29, 32, 35, 42

TLS Transport Layer Security 9, 11–13, 19, 25, 26, 29, 32, 35, 39, 42

UDP User Datagram Protocol 2, 7, 8, 11, 19, 22, 24, 25, 29, 32, 39, 50

VM Virtual Machine 16

VoIP Voice over Internet Protocol 7