When Latency Matters: Measurements and Lessons Learned

When Latency Matters: Measurements and Lessons Learned

Marco Iorio
Politecnico di Torino (DAUIN)
Torino, Italy
marco.iorio@polito.it

Fulvio Risso
Politecnico di Torino (DAUIN)
Torino, Italy
fulvio.risso@polito.it

Claudio Casetti
Politecnico di Torino (DAUIN)
Torino, Italy
claudio.casetti@polito.it

ABSTRACT

Several emerging classes of interactive applications are demanding for extremely low-latency to be fully unleashed, with edge computing generally regarded as a key enabler thanks to reduced delays. This paper presents the outcome of a large-scale end-to-end measurement campaign focusing on task-offloading scenarios, showing that moving the computation closer to the end-users, alone, may turn out not to be enough. Indeed, the complexity associated with modern networks, both at the access and in the core, the behavior of the protocols at different levels of the stack, as well as the orchestration platforms used in data-centers hide a set of pitfalls potentially reverting the benefits introduced by low propagation delays. In short, we highlight how ensuring good QoS to latency-sensitive applications is definitely a multi-dimensional problem, requiring to cope with a great deal of customization and cooperation to get the best from the underlying network.

CCS CONCEPTS

• **Networks** → **Network measurement**; **Network performance analysis**;

KEYWORDS

Network Measurement, End-to-end Performance, Latency, TCP, Edge computing

1 INTRODUCTION

To face the increasing demands for low-latency communication, the edge computing paradigm, which brings powerful, cloud-like infrastructure closer to the end-users, is gaining momentum. Similarly to the driving reasons for the deployment of Content Delivery Networks (CDN), this paradigm allows to keep the traffic local, reducing the physical distance that needs to be traveled by network packets [11, 20]. Still, the ongoing shift from cloud to edge computing mostly targets only one of the many components of the overall E2E latency, the propagation delay, as it is strictly proportional to the proximity between the two endpoints. While being definitely a necessary precondition to meet strict latency goals, potentially decreasing also the probability of packet congestion and the processing overhead introduced by routers along the path, this E2E distance shortening, alone, may turn out not to be sufficient.

Indeed, focusing on the bigger picture, it is possible to pinpoint four additional classes of overhead to be considered. First, the complementary components of the network latency, mainly including transmission, forwarding (classical switches and routers, as well as more complex middle-boxes such as load balancers and firewalls) and queuing delays, both in case of wired and wireless access networks. Second, the overhead ascribable to the transport layers, and in particular to the behavior induced by the congestion control algorithms leveraged by the TCP protocol to modulate the transmission rate. Third, the design of the (likely distributed) application, including the high-level communication protocols (e.g. HTTP, WebSockets [10], MQTT [2], ...), security, as well as the impact of the complex orchestration platforms, such as OpenStack and Kubernetes, which are becoming the de-facto standard inside data-centers of any sizes [8]. Although bringing the isolation and the agility typical of the cloud-native world, these orchestrators come along with various abstractions (e.g. in terms of service exposition), which could impact the communication latency. Forth, the computation time, i.e. the actual time spent by the back-end server to execute the desired task and generate the results returned to the client. Although completely independent from the network performance and strictly related to the business logic of the specific application, it nonetheless accounts in the total latency budget. To this end, it is worth remembering that offloaded computations are typically resource-intensive (e.g. machine-learning oriented), hence requiring non-negligible time even if specialized hardware, such as GPUs, is available [5]. Table 1 summarizes the main factors contributing to the E2E latency, assessing to what extent each one is expected to be influenced by different network and application-related aspects.

In this paper we perform a wide range of latency measurements to comprehensively assess whether the edge-computing promises of extremely low network latency do actually translate into similar application-level performance (i.e. whether the propagation delay dominates the other latency factors), hence meeting the stringent requirements set forth for computation offloading. Unlike most previous investigations (e.g. [4, 7]), which compared the performance of cloud and edge computing in terms of network latency only (i.e. by means of *pings*), we focus on the E2E transaction time. In other words, we consider the entire task offloading process, i.e., from the transmission of the request message to the reception of the corresponding results, to get a broader picture of how much the different components in the communication chain impact on the final performance. Overall, this paper makes two major contributions.

Table 1: The degree of influence of different network and application aspects on the various latency components.

Delay factor	Distance	Network topology	Network bandwidth	Msg. size	App design
Propagation	high	high	low	low	low
Transmission	low	low	high	high	low
Forwarding	med	high	low	med	low
Queuing	med	high	high	med	low
TCP-related	high	low	med	high	med
App-related	low	low	low	med	high

First, it presents the main results of the latency investigation, analyzing the E2E latency while changing multiple variables, including application behavior, access network, servers location and service exposition. Second, it shares the main lessons learned, describing the unexpected pitfalls that can drastically reduce the overall performance and the possible solutions for each of them.

The remainder of the paper is organized as follows. Section 2 details the experimental setup and motivates the metrics adopted for the evaluation. Section 3 presents the outcome of the main measurements, highlighting the pitfalls possibly impairing the performance of delay-sensitive applications. Section 4 focuses on access networks, evaluating buffering-related problems and the latency overhead introduced by wireless networks. Section 5 reviews previous related investigations, while Section 6 draws the main conclusions.

2 MEASUREMENT SETUP

In this section, we describe the experimental setup leveraged for the measurements, detailing the considered task offloading scenario, the characteristics of the test application and the different back-end locations. Second, we present the three main latency metrics considered during the evaluation, motivating the differences and justifying the focus on the application-level flow completion time. Finally, we show a comparison between stateless and persistent connections, and briefly analyze the MQTT messaging protocol.

2.1 Scenario

In our investigation, we focused on a classical computation offloading scenario characterized by a client/server architecture. The former, as an example, may represent an autonomous robot, which periodically sends messages, including sensor readings and camera frames, to a back-end server. This, in turn, performs the appropriate computations and returns the results to the requester. In order to evaluate the latency in a wide range of situations, as well as simplify the configuration and the reproducibility of the measurements, we developed a custom application[1] that periodically sends ping-pong frames over a WebSocket and performs the appropriate measurements. We leveraged WebSockets as they provide persistent, full-duplex communication channels compatible with the traditional HTTP protocol, hence passing through reverse proxies (i.e. L7 load-balancers), which are increasingly common in data-centers. At the same time, we selected Protocol Buffers[2] as data serialization mechanism, since they experimentally proved to introduce lower overhead compared to classical solutions such as XML and JSON (Section 2.3). Although focusing on the E2E transaction time, the application does not simulate the actual computation, as dependant on the specific task and unrelated from the network performance. In other words, as soon as the server receives a request, it immediately replies back with a response of the proper size.

The test application is characterized by two main degrees of configuration. First, in terms of transmission interval, i.e., the time elapsing between the dispatch of two subsequent requests. In our evaluation, we mainly experimented with intervals ranging from 10 ms to 1 s, hence considering both high-frequency transmissions and slow-paced requests. Second, in terms of message sizes, both concerning the requests issued by the client and the responses returned by the server. Sticking to the peculiarities of the scenario, we simulated asymmetric flows characterized by inputs bigger than the corresponding output [23]. Specifically, we considered the following three representative configurations: (i) *Command and control*, characterized by symmetric 1 kB requests and responses. This setup simulates a remote control system and represents a lower bound in terms of latency, given that it introduces very limited transmission delay due to the dispatch of a single packet per message. (ii) *Telemetry*, encompassing 10 kB requests and 1 kB responses emulating the periodic transmission of a set of sensor readings (e.g. originating from radars and lidars 2D) to the back-end. (iii) *Image recognition*, considering 100 kB requests and 1 kB responses to represent the cyclical dispatch of camera frames for processing.

Although the precise numbers can definitely vary based on the specific situation, these arrangements are deemed to represent the three main classes of frame sizes, hence achieving the desired coverage during the investigation to assess the effects of message size variations on the E2E latency. Bigger message sizes have not been considered as, complemented with fast sending intervals, would lead to extremely high bandwidth requirements (e.g., 1 MB frames sent every 10 ms would roughly correspond to 1 Gbps), as well as the total latency would be dominated by the transmission delay.

Focusing on the actual deployments, to probe the different factors influencing the communication latency, we repeated the measurements considering multiple configurations summarized in Table 2. As for the server, we leveraged a lightly-loaded Kubernetes cluster located at the university premises (Turin, Italy), as well as a subset of popular cloud providers. More in detail, we selected three increasingly distant Microsoft AKS instances (Switzerland North, France Central and UK South), and three Amazon EC2 instances (Milan, Paris and London). Although farther data-centers may not be appropriate to fulfill very strict latency requirements, they have nonetheless been considered to assess whether the performance could be acceptable in case of less demanding applications. On the other hand, the client-side of the measurement tool has been hosted first on one server at the university premises (located in a different part of the campus with respect to the Kubernetes cluster), as well as another one inside the corporate network of a partner company also based in Turin. Both represent best-case access networks for task offloading, being characterized by wired connections and 1 Gbps links. Second, we hosted the client on a laptop connected to a domestic network served by a broadband, fiber-to-the-cabinet (FTTC) Internet access (100 Mbps / 20 Mbps). This fostered the investigation of issues afflicting less performing networks, as well as the impact of wireless access. Although the various deployments

Table 2: An overview of approximate distance (as the crow flies) between the client and server locations considered.

	On-prem 1 Turin	AWS Milan	AKS CH South	AKS/AWS France	AKS/AWS UK South
On-prem 1	<1 km	125 km	275 km	575 km	925 km
On-prem 2	5 km	125 km	275 km	575 km	925 km
Home Net.	35 km	150 km	300 km	600 km	950 km

[1]The latency-tester developed for these measurements is open-source, and freely available on GitHub: https://github.com/richiMarchi/latency-tester

[2]https://developers.google.com/protocol-buffers

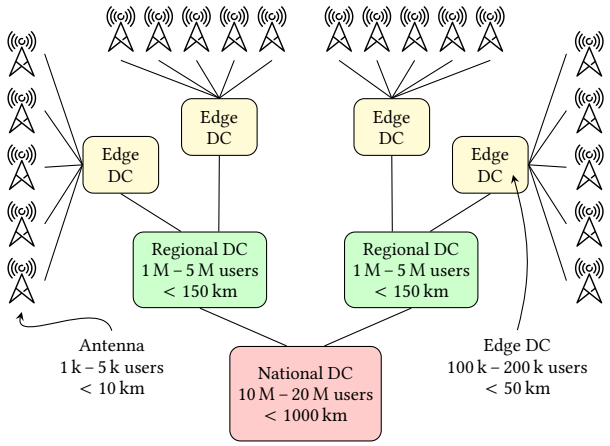

Figure 1: A schematic abstracting a possible organization of the data-centers leveraged by a telecom operator.

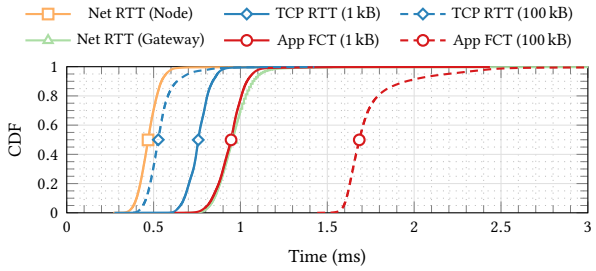

Figure 2: Latency distribution at network, TCP and app-level, considering the dispatch of 1 kB and 100 kB requests.

are possibly characterized by different computing power, they are deemed not to significantly impact the measurements, as no resource intensive task is performed by the end-points.

The matrix of client and server locations, complemented by the different combinations of parameters, is expected to reflect the wide range of scenarios that can be encountered in the task-offloading landscape. To this end, Fig. 1 presents an abstraction of the possible topology of the data-centers leveraged by a telecom operator, characterizing for each one the range of users and the expected E2E distance. Our measurements exploiting public cloud data-centers definitely fall between the regional and the national DC distance range, while the on-premise tests represent a lower bound for the edge data-center scenario. All measurements were repeated multiple times, to obtain statistically relevant results and evaluate the latency stability during the day and across multiple days. All in all, we gathered hundreds of hours of measurements, corresponding to millions of raw latency samples, thus paving the way for accurate analysis even when focusing on high percentiles.

2.2 Latency Metrics

Three main metrics can be considered with respect to latency measurements. First, the *network-level round-trip-time* (RTT), which is typically evaluated by means of the *ping* or *traceroute* tools. Extremely simple, this measurement represents an estimation of the bare performance of the underlying network, thus without including the overhead introduced by higher-level protocols and transport delays. Still, it is associated with different caveats, as ICMP and UDP probes are often blocked by the firewalls of corporate networks and cloud providers, as well as they may be discarded by data-center load balancers. Additionally, these probes are typically characterized by lower forwarding priority compared to other traffic, hence leading to possibly inconsistent results [15]. Second, the *transport-level RTT*, hence focusing on the performance of the actual TCP stream, in terms of the time required to send one segment and receive the corresponding acknowledgement. This metric, which can be evaluated using the analysis tools provided by network sniffers such as *tskark* and *wireshark*, may not represent a complete E2E latency measure (e.g., in case a reverse proxy is used, which terminates the client TCP connection), as well as it has to deal with the TCP optimizations, such as delayed ACKs, which may slightly alter the outcome. Still, it proved to be of extreme value to discern TCP-level (e.g. buffering-related) issues from application-level latency. Third, *application-level RTT*, to assess the entire flow completion time (FCT) from the beginning of the request transmission to the complete reception of the corresponding response from the server. Unlike previous metrics, this E2E measurement includes both the network delays strictly speaking, as well as the time required to transmit the messages. Yet, it allows to evaluate the actual QoS that is perceived by the final application, taking into account also the specificity of the infrastructure and the effects of middle-boxes such as load balancers and proxies. It is worth noting that all the three previously mentioned metrics represent RTT measurements, as they relax the requirement for precisely synchronized clocks imposed by the evaluation of one-way delays.

Fig. 2 presents a two-hour long evaluation of the latency distribution considering the three previously mentioned metrics. Client and server are located in distant areas of the campus network and interconnected by means of a 1 Gbps L3 network. Request messages are dispatched every 10 ms, while pings are issued every 1 s. The Nagle's algorithm is enabled (although we obtained equivalent results regardless of its status) and the application sets the `PSH` TCP flag. Looking first at the ping results, it is possible to observe two main aspects: first, the extremely good performance achieved at the network level, with the median latency (P 50) equal to 0.47 ms and the 99th percentile (P 99) at 0.62 ms only. Second, the relevant divergence between the responses from a physical server and those returned by its preceding router (i.e., its default gateway), with the latter reporting twice as much delay. Focusing now on 1 kB application messages, the bare network performance is mostly translated into the actual flow completion time (P 50 = 0.95 ms, P 99 = 1.13 ms), with a limited overhead introduced both at TCP and application-level. Finally, considering the *Image recognition* scenario (i.e. 100 kB requests), it becomes evident the latency contribution ascribable to the transmission of the messages (even though relatively small), and theoretically accounting for approx. 0.85 ms. Although intuitive, this aspect needs obviously to be considered when designing latency-sensitive applications, given also the resulting longer tail at higher percentiles, which makes it harder to provide QoS guarantees. At the same time, the TCP-level RTT is *lower* than in the 1 kB message case, given the mitigation of the effect introduced by delayed ACKs due to the transmission of multiple network packets.

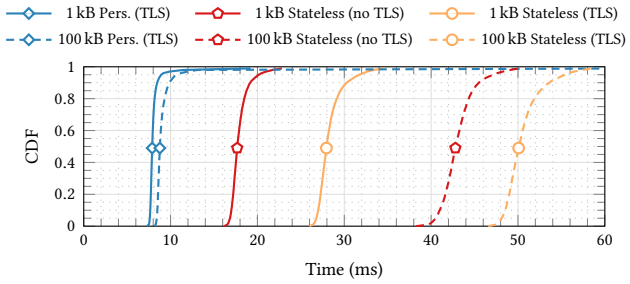

Figure 3: App-level FCT comparison between stateless (w/ and w/o TLS) and persistent connections (w/ TLS). Solid lines refer to 1 kB messages, while dashed ones to 100 kB requests.

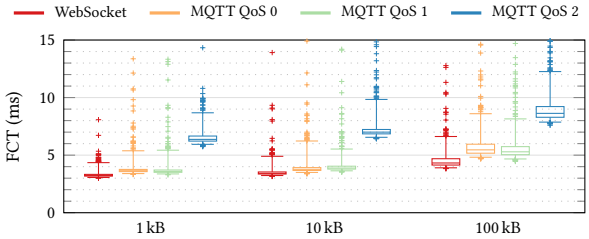

Figure 4: App-level FCT comparison between WebSocket and MQTT-based communication, for a client hosted on-premise and a server (along with the broker), on AWS Milan.

2.3 Stateless vs Persistent Connections

Let focus now on a preliminary comparison between two widely adopted communication approaches for HTTP-based distributed applications. First, we consider a completely stateless approach, characterized by the establishment of a new connection for each different transaction.[3] Second, we use a single, persistent, and full-duplex communication channel, as the one provided by the WebSocket abstraction. Hence, avoiding the overhead required to re-establish a new TCP connection for each request.

Fig. 3 presents the distribution of the application-level FCT in these two situations, simulating a task offloading scenario from the on-premise client to an AKS instance located in North Switzerland, and characterized by a median TCP-level RTT (P50) of 7.2 ms. It is evident at a first glance the significant difference in terms of performance. Focusing on 1 kB frames, the FCT is dominated by the propagation time, accounting for a single RTT in case of persistent channels (P50 $\approx$ 7.9 ms). Yet, stateless interactions are more demanding, requiring an additional iteration to complete the TCP handshake (P50 $\approx$ 17.7 ms) and, if security is required, one more to setup the TLS session[4] (P50 $\approx$ 28.0 ms). Moreover, new connections need to be established through L7 proxies, if any, further increasing the perceived latency. Still, even more relevant differences emerge when considering bigger request sizes, a fact boiling down to the underlying behavior of the TCP protocol. Indeed, newly established TCP connections incur in the *slow start* phase, hence progressively probing for the bandwidth available by sending increasing amounts of segments. In turn, this causes the 100 kB message to require multiple RTTs to be transmitted, compared to the single iteration when the congestion window (*cwnd*) has already reached the steady-state. Finally, it is worth noting that the additional latency associated with the stateless approach is proportional to the network-level RTT, hence increasing the farther the two endpoints are.

Learning 1: Latency-sensitive distributed applications should leverage persistent connections as much as possible, to avoid the establishment overhead of stateless TCP and benefit from the increased TCP *cwnd* values and throughput at steady state.

[3]To fully highlight the connection establishment overhead, we disabled *HTTP keep-alive*, hence effectively leveraging different underlying TCP connections.

[4]The interactions necessary to establish a TLS connection depend on the version of the protocol: TLS 1.2 and earlier require two RTTs, while TLS 1.3 reduced the overhead to one RTT only. Furthermore, it also supports a 0-RTT configuration, to allow resumption of existing connections with no additional latency [19]. Fig. 3 refers to TLS 1.3 with a default configuration.

This result justifies the adoption of persistent connections for the rest of the evaluation. Similarly, switching from JSON to Protocol Buffers for payload serialization reduced the associated transmission and reception processing overhead by 1 ms to 2 ms in case of 100 kB messages, complemented by a more compact FCT distribution even with smaller sizes. All in all, this highlights how every single design choice can have an extremely significant impact on the QoS offered by the application. Hence, raising the need for cautious decisions throughout the entire development of latency-sensitive applications, to avoid impairing the underlying network and losing the advantages offered by close-by data-centers.

Learning 2: When dealing with low-latency requirements, it is necessary to adopt efficient data serialization mechanisms and libraries, to achieve good performance both in terms of resulting message size and, most importantly, encoding/decoding time.

2.4 Publish/Subscribe IoT Messaging Transport

Concluding our preliminary analysis, we now focus on a brief comparison between persistent WebSocket connections and MQTT, a lightweight publish/subscribe messaging transport that is becoming increasingly common in many industrial and IoT scenarios [17]. As for the latter, we leveraged once more a ping-pong approach, with a client publishing to a first topic and measuring the delay before the response is echoed back by the server on a second one.

Fig. 4 presents the results of the evaluation, considering a client deployed on-premise and the server on AWS Milan. For the sake of fairness between the two approaches, we co-located the message broker (Eclipse Mosquitto[5]) along with the server, hence avoiding the introduction of additional network overhead. Furthermore, we evaluated all three QoS levels supported by the MQTT specifications about message delivery [2], namely *at most once* (0), *at least once* (1) and *exactly once* (2). Focusing on the actual numbers, it is possible to draw two main observations. First, the relatively limited latency difference between WebSocket and MQTT-based communication (QoS 0 and 1), with the latter abstraction accounting from 0.3 ms up to 1 ms in case of bigger request sizes. Second, the twice as much delay associated with MQTT QoS 2, given the more complex interaction required to ensure the desired guarantees. Finally, it is worth mentioning we needed to explicitly disable the Nagle's algorithm on the message broker, to prevent high latency issues experienced with MQTT QoS 1 and 2.

[5]https://mosquitto.org/

Learning 3: MQTT and the associated message broker introduce limited overhead compared to TCP, at least when QoS 0 or QoS 1 and the proper configuration are used. Conversely, the additional delay becomes relevant when the highest QoS is required.

Learning 4: Latency-sensitive applications shall disable the Nagle's algorithm to prevent unexpected delays if small packets, either data or control, are transmitted over a TCP connection.

3 LATENCY MEASUREMENTS AND PITFALLS

This section presents the outcome of a set of close-up investigations analyzing specific aspects deemed to negatively influence the delays experienced by latency-sensitive applications, highlighting at the same time the pitfalls encountered during the measurement campaign. Specifically, we start focusing on a comparison between different common approaches to host and expose a back-end service in a data-center, before characterizing an application behavior that turned out to have unexpected effects on the FCT. Finally, we move on and analyze the impact of possible core-network inefficiencies. All the measurements refer to wired networks, while leaving for the subsequent section the investigation in case of wireless access.

3.1 Hosting the Back-end Service

Focusing on a task offloading scenario, it is reasonable to assume the back-end computations to be performed in a data-center, either public or private. Still, multiple applications, or even instances of the same component, typically need to coexist on the same infrastructure, to better leverage the resources available and avoid reserving entire bare-metal servers to single tasks. Hence, raising the need for orchestration solutions, which essentially take care of the execution, resilience and isolation of multiple services.

Sticking to cloud-native, open-source solutions, Kubernetes is gaining momentum and has therefore been chosen for our investigation. Still, many choices are possible when it comes to make the desired HTTP services reachable from the clients. Supposing an n nodes cluster, with the target application running in a container hosted on node x, it is possible to consider two main alternatives. First, the back-end can be exposed leveraging a Kubernetes *LoadBalancer* service[6], which takes care of assigning a routable IP address to the service and appropriately announcing it to the network infrastructure (e.g. through the BGP protocol or gratuitous ARPs)[7]. If a specific flag of the service, namely *ExternalTrafficPolicy*, is set to *Local*, the incoming requests are directly forwarded to the node hosting the back-end, x in this case. Conversely, when *ExternalTrafficPolicy* is set to *Cluster* (i.e., the default one), the traffic may incur in a second hop, reaching first node y and then being forwarded to node x, which actually hosts the application. This allows better overall load-spreading in presence of multiple replicas, at the cost of possibly increased latency. The same situations can also be reproduced using a *NodePort* service (as done in these measurements for additional control), though this requires the nodes to be assigned routable addresses and it imposes limitations on the ports that can be used. The second alternative involves a reverse proxy, which consists of an application-level load balancer in charge of terminating the TCP and TLS connections, possibly exposed through a Kubernetes *LoadBalancer* service. This represents the single entry point for many services and it forwards each request to the appropriate back-end. In this study we selected the *NGINX Ingress Controller* (community edition – v0.44) with default settings, being it by far the most used solution in Kubernetes today [8].

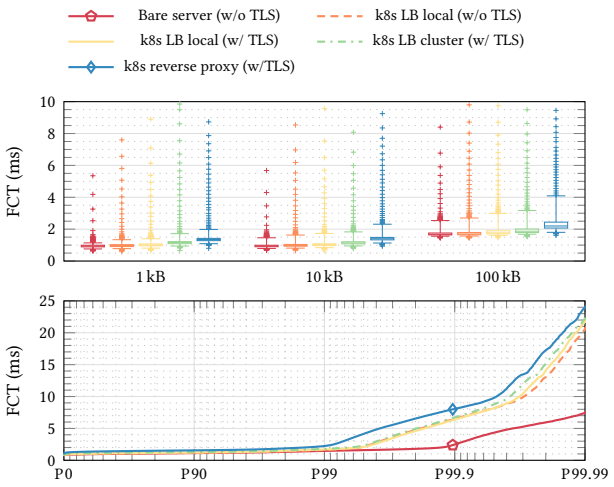

Figure 5: App-level FCT comparison varying the approach used to host and expose the back-end service. The box-plot refers to three scenarios characterized by increasing request sizes, while the lower graph represents a 110-hours long close-up analysis at high percentiles in case of 10 kB frames.

Fig. 5 presents the outcome of a twelve-hours long evaluation, comparing the three alternatives considered above (i.e. load balancer in *Local* and *Cluster* modes, and reverse proxy), complemented by different combinations in terms of security. Finally, the application is also hosted on a bare-metal server, to assess the overhead introduced by the container abstraction and orchestration (i.e. Kubernetes). Client and server are both executed on-premise, leveraging the low network latency offered by the campus network to better highlight the actual differences. The box-plot represents the extension of the quartiles, as well as of the 1st and the 99th percentile (a representative set of outliers is shown as individual points) of the FCT for messages of three different sizes. Results are very similar both for the application executed on the bare-metal server and the ones exposed through the load balancer, with no relevant variations introduced by TLS when the connection is already established. Still, the load balancing *Cluster* policy is associated with slightly higher latency (0.1 ms – 0.2 ms), ascribable to the additional hop. Differently, the usage of a reverse proxy caused a more relevant overhead (0.5 ms – 1 ms at P99), confirming the impact introduced by the additional software component.

Learning 5: In the majority of cases, the different Kubernetes service-exposition approaches introduce a negligible delay, thus requiring extremely low propagation delays to be of any relevance.

Learning 6: The adoption of TLS to secure network traffic has practically no impact at run-time with powerful enough devices,

[6]https://kubernetes.io/docs/concepts/services-networking/service/

[7]It is worth noting the LoadBalancer service is a functionality that is either offered by the cloud provider hosting the Kubernetes cluster or enabled by an additional component, such as MetalLB, in case of on-premise deployments.

while it possibly introduces a significant overhead during the connection establishment phase (Fig. 3).

Moving on, the lower part of Fig. 5 shows a 110-hours long close-up analysis of the results associated with 10 kB requests (although the trend is expected to be similar also in the other cases), focusing on higher percentiles to better pinpoint the latency distribution in the context of strongly demanding scenarios. Indeed, mission and safety critical applications may definitely require extremely high reliability [16], as missing even a single deadline could turn out to have catastrophic effects. In an effort to achieve high confidence, we collected almost 40 million latency samples: a new request is dispatched every 50 ms, and all the measurements are executed in parallel to uniform the impact of possible temporary network problems. In a nutshell, while the different alternatives are relatively aligned up to P90, and even P99, they start to diverge at higher percentiles. Given the definitely more stable results obtained in the bare server scenario compared to the other cases, it looks like most of the overhead, especially at high percentiles, is related to the application execution by the end-point devices, both client and server (e.g., temporary load bursts, job scheduling, interrupt handling, ...), likely due to the higher number of software components involved (e.g., application running in containers instead of bare hardware, the introduction of the reverse proxy). Sure enough, part of the bare server outliers has also the same cause, albeit to a lesser extent thanks to the reduced complexity.

Learning 7: Given the nature of the latency long tail measurements and the apparently underloaded conditions of the *enterprise network*, the possible improvements that could be achieved by introducing QoS/slicing technologies in the above infrastructure is questionable. Similar results have been achieved also on some ISP *core networks*, particularly when reaching the closest data-centers.

Learning 8: Distributed approaches leveraging shared infrastructure (both network and compute) are hardly applicable in mission and safety critical scenarios due to the FCT rise after P99. Still, more complex setups (i.e., based on containers and optionally reverse proxies) are characterized by worse performance, thus raising the need for careful analyses about trade-offs between simplicity and performance when requiring high reliability.

3.2 Slowed Down Low-Frequency Applications

Given the necessity to use persistent connections and the use-cases considered during the measurements, the application, and so the underlying TCP connection, alternates active periods characterized by the actual transmission of request data and idle instants, i.e. waiting for the next sending slot. Although apparently irrelevant at first sight, this aspect turned out to have devastating effects on the performance perceived by the application, as represented in Fig. 6.

The graph compares the app-level FCT with the client deployed on-premise and the server hosted on AKS Switzerland North, considering 100 kB request messages to simulate the offloading of an image recognition task. All parameters are kept constant between one measurement and the other, varying only the interval between the transmission of subsequent messages. Numbers show how the FCT starts to grow as soon as the send interval exceeds 200 ms, with the median value boosting from 9 to 31 ms if the application

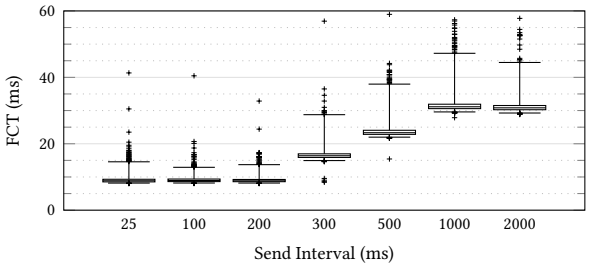

Figure 6: App-level FCT comparison varying the interval between the dispatch of one 100 kB request and the subsequent, while maintaining unchanged all the other variables.

remains idle for more than 1 s between two requests. In other words, the duration of the application duty cycle is indirectly, as well as unexpectedly, impacting the offered QoS.

This three-fold latency increase is due to the algorithm suggested in RFC 2861 [14], which aims at preventing the dispatch of excessive bursts of packets if the sender remains idle for relatively long periods of time (i.e. higher than the retransmission timer — RTO — estimation), by progressively halving the TCP *cwnd* value. Although this behavior has been recognised to strongly damage the performance of periodic communications over persistent channels due to *cwnd* shrinking and the above RFC is obsolete [9], it is nonetheless currently enabled by default in Linux when CUBIC TCP is adopted. Still, while this feature can be turned off modifying an appropriate flag[8], as done in the measurements shown in this paper, as well as adopting a congestion control algorithm which leverages different pacing techniques (e.g. BBR [3]), these modifications typically require the control of the operating system and may not be configured directly by orchestrators such as Kubernetes.[9] Hence, raising the need for careful pre-deployment analysis to ensure application performance is not impaired by incorrect settings, as well as for collaboration between developers and system administrators, to provide the best QoS possible for the given scenario.

Learning 9: Latency reduction requires the collaboration of multiple actors, including network providers, data-center providers (e.g. to configure TCP parameters and tune reverse proxies behavior) and application developers. Hence, possibly involving different people/teams, even when leveraging dynamic platforms such as Kubernetes. Indeed, application tuning, alone, is often not enough.

3.3 Core Network Analysis

Up to now, our analysis focused on the impact of the application behavior (i.e. client side), as well as the exposition of the back-end service (i.e. server side). We present now different measurements to assess which characteristics of the core network could impact the actual QoS, besides propagation and transmission delays.

[8] i.e., `tcp_slow_start_after_idle`, which is exposed through the `/proc` interface: https://man7.org/linux/man-pages/man7/tcp.7.html

[9] At time of writing, the *sysctl* flags used to either disable the "slow start after idle" behavior or change the congestion control algorithm are not included in the Kubernetes *safe* set, and need to be allowed by the cluster administrators on a per-node basis before being configurable when defining the application deployment (https://kubernetes.io/docs/tasks/administer-cluster/sysctl-cluster/).

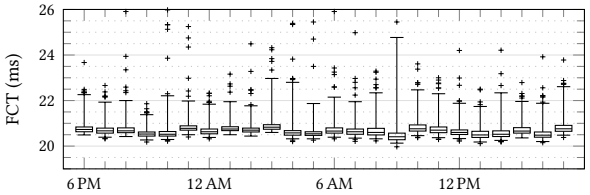

Figure 7: 24-hours app-level FCT evaluation between an on-premise client and AWS Paris, highlighting the stability of the network over one day.

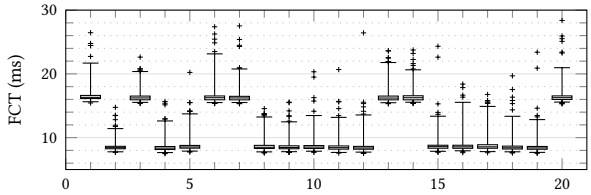

Figure 8: 20-runs FCT evaluation between an on-premise client and AKS Switzerland, showing the effect of inefficient ECMP routing performed by the core network and impacting the path followed by packets of different sessions.

3.3.1 Temporal stability. We carried out multiple 24-hours long measurement campaigns to assess the stability of the backbone network performance between on-premise locations and data-centers, as well as between multiple data-centers located in different regions. While constant and predictable communication delays are fundamental for many latency-sensitive applications, it seemed intuitive to observe varying behaviors due to possible congestion during rush hours. Fig. 7 presents the outcome of one of these evaluations, showing the FCT experienced between a client deployed on-premise and a back-end service hosted on AWS Paris, while exchanging 10 kB requests and 1 kB responses. To our surprise, all results showed extremely stable performance, and no relevant and consistent correlation appeared to exist between delay and time of the day. Indeed, in all cases the medians of 24 runs were contained in the 0.2 ms to 0.5 ms ($\approx$ 1 % to 2 %) range, with only slightly higher differences at P90 ($\approx$ 1 ms). Additionally, inter-data center communications appeared to be even more stable, while on-premise networks introduced slightly higher variability. For the sake of completeness, it is worth mentioning that during one very specific measurement (i.e. client hosted on AKS London and server on AKS West US) we noticed an abrupt 5 ms FCT (and TCP-level RTT) drop at midnight GMT. Most probably, it depended on a scheduled modification of some routing configuration, which caused a shorter path to be followed by the packets.

Learning 10: The latency between a given pair of endpoints is virtually constant throughout the day, and across multiple days.

3.3.2 Routing Instability. While showing stable performance, the long-run measurements uncovered at the same time a definitely less desirable behavior, which is graphically represented in Fig. 8. Specifically, the box-plot details the app-level FCT experienced by a client located at the university premises communicating to a back-end server hosted on AKS Switzerland North (although a similar pattern reproduced also towards the other AKS regions considered). At a first sight, it is immediately evident the extreme difference between multiple runs, with some sessions characterized by a median around 8.5 ms and others almost twice as much, 16.25 ms. This specific behavior turned out to be caused by the network topology[10] and routing policies of the Consortium GARR (i.e. the Italian national network provider for universities and research centers).

In fact, the traffic coming from our university, located in Turin, and directed to AKS data-centers gets forwarded by equal-cost multi-path (ECMP) routing either towards Milan (`MI2`) or Rome (`RM2`). Then, the packets reach the corresponding Internet Exchange, `MIX` or `NAMEX` respectively, before entering the Azure network. Still, two paths may be equivalent from the routers point of view (e.g. based on hop count or links speed), but characterized by completely different performance considering other metrics (e.g. the E2E latency, which depends also on the physical distance). ECMP routing is typically performed at session level, to prevent the occurrence of packet reordering: indeed, configuring a fixed source port (as the other network parameters are constant across subsequent runs), this variability is no longer present and all runs stabilize around either one of the two values. We leveraged this peculiarity during the other evaluations to achieve consistent results, forcing the packets to flow through the shorter path.

Although these observations stem from a rather peculiar behavior characteristic of a specific network, it is nonetheless reasonable to assume similar problems to be present to different extents in a wide variety of scenarios, possibly impairing latency-sensitive applications. Additionally, being related to the core network, they fall outside the control of system administrators, hence raising the need for different solutions. Indeed, in presence of persistent connections and multiple unequal paths, it may be wise to leverage helpers opening parallel channels to probe for the best available characteristics, and select the ones ensuring the desired QoS.

Learning 11: Uncontrollable core-network routing configurations can unexpectedly and dramatically influence the performance of delay-sensitive applications. Continuous monitoring is fundamental to quickly detect modifications and, when possible, put in place countermeasures (e.g., forcing shortest path selection).

3.3.3 Routing Inefficiencies. We considered different close-by locations suitable for the deployment of the client and server components of an application with relatively loose latency requirements (i.e. < 30 ms). Fig. 9 summarizes the outcome, considering two on-premise hosts located in Turin, respectively in our university and in a partner company, as well as back-ends deployed in Western Europe public-cloud data-centers. Finally, we simulated an increasingly common inter-cloud scenario, hosting the two components of the application on different cloud providers, located in the same region, accounting for resiliency and cost effectiveness demands.

When talking about latency, one typical assumption regards two endpoints physically close-by being also characterized by extremely low propagation delays. Yet, this may not necessarily turn out to be true in all situations, as the path followed by the packets may be much longer due to the interconnection of different network

[10] https://gins.garr.it/xWeathermap/mapgen.php?slice=garrx_top

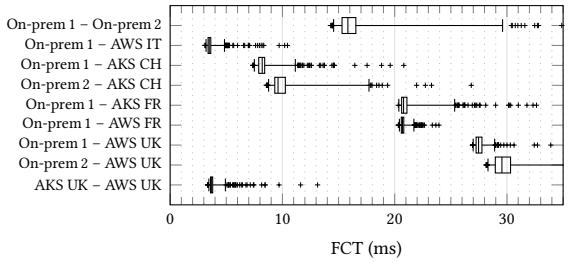

Figure 9: App-level FCT comparison focusing on different client-server locations, issuing 10 kB requests every 100 ms.

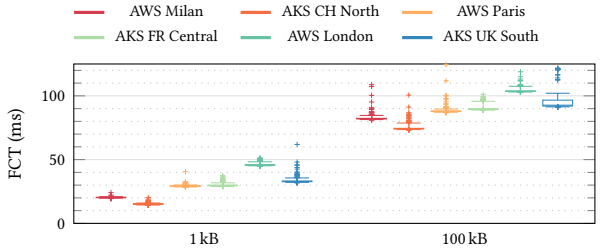

Figure 10: App-level FCT comparison between a domestic client and different servers hosted on public data-center locations, issuing either 1 kB or 100 kB requests every 100 ms.

providers and possible routing inefficiencies. As a representative example, the on-premise to on-premise evaluation displayed a median FCT close to 16 ms, although client and server were effectively located in the same city, only few kilometers apart. Conversely, much better performance has been achieved with the server hosted on AWS Milan (IT), characterized by a P50 FCT as low as 3.5 ms, thanks to the more direct core network interconnection. Even hosting the server farther, on AKS Switzerland (CH), led to definitely better results; still, the two clients experienced quite different QoS, with the former associated with a lower median and an overall more compact distribution. Both AKS Central France (FR) and AWS Paris (FR) showed stable performance, although with a relatively higher base delay ($\approx$ 20 ms) due to the increased physical distance. Moving to farther data-centers, AWS UK was characterized by a median FCT latency between 27 ms and 30 ms, once more with different stability depending on the client access network. Finally, the inter-cloud scenario achieved definitely good results, although the packets had to go through the network of two different cloud-providers.

Learning 12: Physical distance alone does not always represent a good approximation of the propagation delay, given the possible effects of routing inefficiencies. This aspect becomes all the more relevant the more network providers are crossed, given the increased probability of distant interconnections for economic reasons.

4 THE IMPACT OF THE ACCESS NETWORK

This section focuses explicitly on access networks, to present an overview of the overhead introduced by different alternatives. Indeed, the previous measurements were carried out in the best case scenario, encompassing the 1 Gbps wired links typical of corporate networks. Still, task offloading is usually envisioned for mobile devices, hence leveraging less performing wireless networks and leading to the Mobile Edge Computing (MEC) paradigm [1]. More in detail, we initially consider a residential network served by a broadband, FTTC Internet access, measuring the latency towards different public data-centers and analyzing the effects of buffering and parallel transmissions on FCT. Then, we experiment with wireless networks, to characterize the additional latency overhead.

4.1 The Latency towards Public Data-Centers

Focusing on residential networks, Fig. 10 presents an FCT comparison between a domestic client, served by an FTTC Internet access (100 Mbps / 20 Mbps), and multiple servers hosted on public data-centers across Western Europe. Looking back at the performance achieved in corporate networks (Fig. 9), it is immediately evident the latency difference, although the clients are close-by in terms of geographical distance ($\approx$ 35 km). This factor, observed also in [13], is most likely ascribed to the complexity of ISP access networks, introducing a relevant latency overhead due to complex forwarding policies and invasive packet processing.

More in detail, considering 1 kB messages first, AKS CH is unexpectedly characterized by the lowest median latency (15.1 ms), being faster than the server hosted on AWS Milan, which is geographically closer ($P50$ = 20.3 ms). Similarly, it is possible to observe a relevant difference between the two close-by UK end-points, with AWS resulting 13 ms slower than AKS. Thus, showing a limited correlation between the experienced latency and the geographical distance. Additionally, it is also worth mentioning that in several cases, with AWS Paris being the most severe, we observed ECMP-related latency variations, hence requiring to select a fixed source port to achieve stable performance across multiple runs. Moving to the analysis of the results associated with 100 kB messages, the transmission delay becomes clearly predominant, with approx. 60 ms added to the previous measurements, and corresponding to an upload throughput slightly greater than 14 Mbps.

Given the unexpected results, we extended the evaluation to multiple clients located in the Turin area and served by different ISPs, measuring the latency experienced towards the two close-by data-centers. The results are presented in Fig. 11, and elicit two main considerations. First, the relatively aligned performance towards AKS CH, with only the ISP 3 sample being characterized by lower latency. Second, the definitely strange behavior observed by ISP 1 clients towards AWS Milan. Indeed, repeating the measurements a day apart (i.e. ISP $1^{\ddagger}$), the delay decreased by almost an half. Still, all samples were in both cases consistent across consecutive repetitions (even spaced by different hours), as well as among the different clients. No difference was observed either from other ISPs or from the corporate network, hence confirming the unexpected variation to be related to some ISP 1 internal routing variation. Finally, considering different clients located in central and southern Italy (not shown in Fig. 11), we also measured a median FCT in the 15 ms – 20 ms range (with limited dispersion) towards the same two data-centers, showing once more that the experienced E2E latency is currently not dominated by the physical distance.

Learning 13: Residential networks are typically associated with higher base delays compared to corporate networks, due to the complexity of the ISP access network and commercial policies. Still,

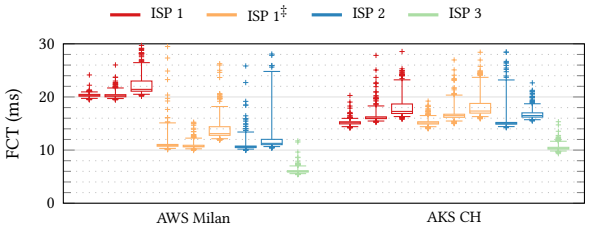

Figure 11: App-level FCT comparison between residential clients located in Turin and its province (served by different ISPs) and close-by data-centers, issuing 1 kB requests.

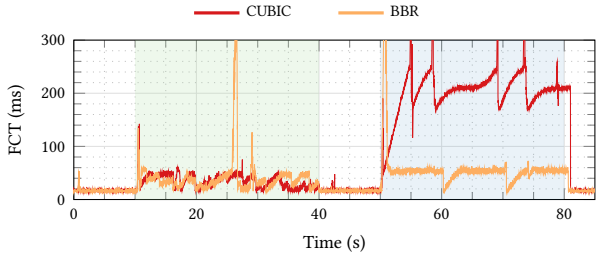

Figure 12: FCT comparison between a domestic client and AKS CH, showing the effects of parallel transfers and buffering with two CC algorithms. The delay-sensitive stream is characterized by 1 kB requests/responses exchanged every 10 ms; the green (10 s – 40 s) and blue (50 s – 80 s) shaded areas represent respectively parallel download and upload flows.

this additional overhead appears to be mostly independent from the physical distance to the destination, hence limiting the possible advantages brought in by close-by data-centers. Additionally, the experienced latency can abruptly and inexplicably change completely, due to internal ISP routing variations.

4.2 When Packets get Stuck in Traffic

Access networks can also lead to subtler issues, especially when delay-sensitive applications and parallel, high-throughput, transfers do coexist. Considering an automotive scenario, the latter may be represented by infotainment traffic or over-the-air updates, as well as the transmission of diagnostic information. Fig. 12 exemplifies this situation, encompassing periodic command and control interactions towards a back-end hosted on AKS Switzerland and characterized by a base latency of 16 ms. Small, 1 kB messages perfectly reflect the TCP-level RTT induced by the parallel transfers, without being affected by the reduced bandwidth. Looking at the graph, it is immediately evident how parallel file transfers, both downstream and upstream, respectively represented by the green (10 s – 40 s) and blue (50 s – 80 s) shaded areas, can dramatically impact the application QoS. This delay increase boils down to the behavior of TCP congestion control (CC) algorithms, which attempt to maximize the throughput and speed-up file transfers.

To this end, classical loss-based approaches, such as CUBIC, have been shown long ago to badly interact with the buffers located before bottleneck links, progressively increasing the queue length and leading, in the extreme cases, to the bufferbloat phenomenon [12]. Fig. 12 shows precisely this behavior, with the FCT experienced by the delay-sensitive application starting to grow immediately at the onset of the parallel *download* stream, reaching up to three times the base latency and displaying extremely high variability. Still, much worse performance is undergone in case of parallel *upload* streams, ascribable to the lower available bandwidth characteristic of asymmetric access links (i.e. about one fifth the downstream direction) and the possibly bigger buffers featured by the home gateway. Switching to BBR, a CC algorithm based on a different approach to estimate the available bandwidth and aiming at reducing the queuing pressure on bottleneck buffers, the measured FCT definitely improves, although remaining three times as high as in absence of any parallel flow (50 s – 80 s).

Learning 14: Network-based bufferbloat mitigation techniques, such as smart queue management algorithms to identify and prioritize latency-oriented flows, as well as possibly dedicated network slices to ensure the best performance in case of strict requirements, may be required particularly in the *access network*. However, this requires a solution to the well-known problem of accurate classification of the incoming traffic, being able to distinguish a device when involved in a real-time communication or in a bulk data transfer (e.g., firmware update), which may happen at any time.

4.3 Wireless Access Links

Concluding our analysis, we finally experimented with wireless access links. Acknowledging the typical usage of low-end devices in many IoT applications, we leveraged a Raspberry Pi 3 Model B+ board to host the client. As for networking, it features both 1 Gbps Ethernet and dual band 802.11ac Wi-Fi, although the former is limited to ≈ 300 Mbps due to internal bus limitations. To better highlight the differences, the server was executed by a laptop connected through a wired link to the access point, represented by a high-end domestic modem/router (FRITZ!Box 7590). Without claiming completeness, we believe these measurements to complement the rest of our analysis, giving an overview of the additional overhead introduced by wireless links.

Fig. 13 presents the outcome of the evaluation, leveraging both wired and wireless connections, the latter both close to the access point (i.e. ≈ 1 m LOS) as well as relatively far (≈ 10 m NLOS), to account for different scenarios. Additionally, we considered three incremental message sizes, highlighting both the overhead ascribable to the wireless medium and the effect related to the possible bandwidth reduction. Finally, we disabled Wi-Fi power management features, to prevent performance degradation. Focusing first on 1 kB messages, it is immediately evident the latency difference between wired and wireless links, with the median value (P50) boosting from as low as 1 ms in the first case, up to more than three times higher (3.4 ms) in the latter. Additionally, wireless samples are associated with higher dispersion and worse performance at high percentiles, hence being possibly inappropriate in case of strict reliability requirements. Moving to bigger sizes, the trend is confirmed and rather intensified, due to the additional impact of the lower bandwidth offered by the wireless medium (≈ 100 Mbps, as measured by the `iperf3` tool). As for wireless measurements, the physical distance from the access point turned out to have limited impact on the P50 value, while displaying increased variability

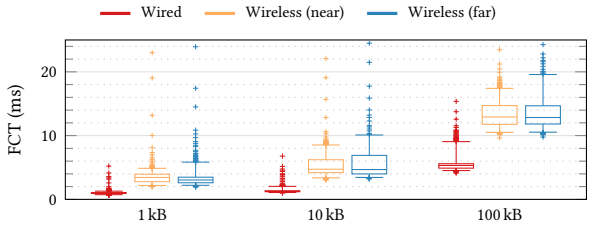

Figure 13: On-LAN App-level FCT comparison switching between wired and wireless access networks.

at higher percentiles when moving the client farther. Still, during preliminary measurements (not presented in Fig. 13), we definitely struggled more to achieve good performance in the latter case, due to the higher sensitivity to external interference, causing latency spikes and accidental negotiation of lower data-rates, impairing the performance especially in case of bigger message sizes.

Learning 15: Latency-sensitive applications should leverage wired links, if possible. If wireless is required, it is necessary to account for the additional delay (and greater dispersion), until low latency medium will be fully available. Power management tuning can improve the performance at the cost of higher consumption.

5 RELATED WORK

The increasing prominence of latency-sensitive applications, as well as the rise of distributed paradigms such as cloud and edge computing, has undoubtedly fostered interest in the community to evaluate the associated performance.

Back in 2012, Choy *et al.* [7] analysed the on-demand gaming scenario, evaluating the delay experienced by 2500 US users to reach different AWS EC2 instances. They assumed 80 ms as an upper bound latency to ensure good QoS, showing less than 70 % of the considered end-users could meet the target. Still, they argued leveraging CDN edge servers equipped with additional processing units could allow 28 % more users to achieve the desired latency. In [21] the authors compared single-cloud and multi-cloud deployments, showing 20 % – 50 % of IP prefixes would benefit from the latter by more than 20 % in terms of lower latency, thanks to the reduced distance to the closest data-center. Yet, users in certain countries would still experience high RTT (i.e. > 100 ms) due to the lack of near-by data-centers and the effects of routing inefficiencies. Recently, Charyyev *et al.* [4] performed a large scale latency evaluation between end-users, represented by *Ripe Atlas* nodes, and cloud and edge locations, the latter constituted by *Akamai* servers. In the end, they observed 82 % of the users could reach the closest edge server in less than 20 ms, while only half obtained comparable performance towards cloud data-centers. Interestingly, this difference shrinks in Western Europe, given the widespread presence of data-centers. These works strictly focused on the bare network latency measured by means of *pings* or TCP-level probes, showing the advantages of physical closeness in terms of reduced latency. Still, low propagation delays represent just one of the different factors influencing the application-level QoS.

Switching to full-stack evaluations, in 2017, Chen *et al.* [6] assessed the performance of different computer vision applications offloaded from a smartphone to edge and cloud back-ends, considering both Wi-Fi and LTE access networks. They observed better performance leveraging edge servers, thanks to the combination of lower latency and higher bandwidth, overall speeding up the communication by 100 ms – 200 ms. Still, a 4G access network was associated with higher latency, partially losing the advantages brought in by physical closeness. Additionally, most application-level FCTs turned out to be dominated by the server-side processing time (i.e. accounting for hundreds of ms to even seconds in the worst case), hence demanding for hardware accelerators and possibly reverting the results in case of uneven computational power. Recently, Gorlatova *et al.* [13] measured the task completion latency of responsive IoT applications, considering on-premise, conventional and serverless cloud back-ends. Most notably, they observed definitely worse performance associated with the latter approach, due to start-up time and higher variability ascribable to prioritization. Generally speaking, as expected, conventional cloud execution points were characterized by increased communication overhead with respect to on-premise devices, although achieving lower FCTs in case of compute-intensive tasks, thanks to the increased available processing power. Moving on, in [18], the authors evaluated the impact of distributing the micro-services composing common applications, such as e-commerce and social network platforms, partly at the edge and partly in the cloud. They showed this approach to potentially hide unexpected pitfalls if the components are not designed properly, leading to worse performance than cloud-only deployments. Indeed, response times can dramatically increase due to the interconnected nature of micro-services, requiring multiple communications back and forth between the edge and the cloud to answer a single request and multiplying the actual network delay.

Finally, Xu *et al.* [22] performed a wide-range measurement study to assess the performance of 5G, given its promises for high bandwidth, ultra reliable and low latency communication. They evaluated physical layer QoS, E2E throughput and latency, as well as smartphone energy consumption and they mostly experimented with the 5G base stations deployed in their campus, leveraging the widely adopted Non-Standalone (NSA) mode: hence, reusing legacy 4G infrastructure for the control-plane to reduce costs. The outcome of their analysis was characterized by both lights and shades: on the one hand, they observed increased bandwidth as well as reduced latency, the latter mostly thanks to the flattened core network architecture. Still, they experienced coverage problems and sharp degradation indoor, along with excessive handover delays due to the shared control plane with 4G and high power consumption. Additionally, traditional loss-based TCP congestion control algorithms turned out to lead to poor throughput (i.e. < 33 %) due to severe packet losses they ascribed to small buffers in the wired network part. All in all, they concluded raising the need for optimizations in the entire ecosystem, from the infrastructure to protocols, in order to fully unleash the potential of 5G for applications characterized by extremely stringent bandwidth and latency requirements.

6 CONCLUSIONS

Ensuring constrained latency to distributed applications is becoming a mandatory requirement in a variety of scenarios, ranging from automation to transportation, from medicine to entertainment.

Learning 16: While edge computing is typically assumed as a viable solution to this need, moving data-centers close to the users turned out to be only a small part of a bigger picture and, probably, not even the most important one. In fact, considering the data-center diffusion at least in Western Europe and the relatively low propagation delays, it is currently questionable the race towards the edge if justified by latency reductions only. Indeed, in all non-trivial scenarios, the E2E delay would much likely be dominated by the computation and transmission time.

This paper highlights how guaranteeing good QoS to delay-sensitive applications is definitely a multi-dimensional problem, requiring to focus on the entire communication stack, from network parameters to transport and application-level aspects, as well as to carefully tune a series of system variables. Still, multi-dimensionality is associated with great complexity, since it requires the cooperation of multiple actors to achieve the best performance. Indeed, even considering the extreme simplicity of our measurement tool, apparently insignificant design choices turned out to drastically impact the latency performance: hence, raising the need to thoroughly weight up every decision during the development process. Focusing on the surrounding environment, the complexity of modern networks, together with the interaction with transport protocols, can possibly conceal unexpected pitfalls dramatically impairing the actual performance. This strongly highlights the importance of preliminary evaluations before deploying delay-sensitive distributed applications, anticipating possible problems and preventing issues from being unnoticed until the system is delivered to the final customers. At the same time, continuous and accurate performance monitoring at multiple network levels is of fundamental relevance even after application deployment, to ensure consistent QoS across time and quickly identify possible variations (e.g., routing) impairing the application latency.

Learning 17: Given the number of involved actors and the difficulties to reach stable numbers beyond P99, applications with strict latency requirements should be engineered to autonomously cope (reliably) with sudden variations and spikes, confirming once more that building clever and robust applications is a better option than counting on a complex and unpredictable infrastructure substrate.

Concluding, we also highlight the difficulty of obtaining FCTs < 10 ms, even considering only on-LAN communications and bare network RTTs of 1 ms or less, which is achievable e.g. in a production factory with proper infrastructure setup. First, even in ideal configurations, it is fundamental to take into account transmission and processing delays, with the latter being particularly relevant (and possibly variable) in case of non-trivial tasks. Second, applications associated with those strict requirements are typically either mission or safety critical, hence demanding for extreme reliability too. Still, our investigation showed how problems tend to emerge when focusing on P99 and higher, questioning the feasibility of leveraging shared infrastructure in these scenarios. Although better reliability could be certainly achieved exploiting isolated network slices to prevent interference, complemented with real-time operating systems to increase predictability, we still believe this goal to be definitely challenging. Leave it alone meeting the 1 ms target, which would undoubtedly require a complete paradigm shift.

ACKNOWLEDGMENTS

The authors would like to thank R. Marchi, for the development of the *latency-tester* tool and the support, together with N. Chiapello and F. Semeraro, during the data gathering phase, L. Camiciotti, F. Ciaccia, A. Doglioli, M. Longo, A. Manzalini, W. Palazzo, G. Perlo, R. Procopio, R. Querio and M. Reineri for the inspiring discussions.

REFERENCES

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. 2018. Mobile Edge Computing: A Survey. *IEEE Internet Things J.* 5, 1 (Feb. 2018), 450–465. https://doi.org/10.1109/JIOT.2017.2750180

[2] A. Banks, E. Briggs, K. Borgendale, and R. Gupta. 2019. *MQTT Version 5.0.* Technical Report. OASIS Open. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf Accessed on: May 31, 2021.

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. 2016. BBR: Congestion-Based Congestion Control: Measuring Bottleneck Bandwidth and Round-Trip Propagation Time. *Queue* 14, 5 (Oct. 2016), 20–53. https://doi.org/10.1145/3012426.3022184

[4] B. Charyyev, E. Arslan, and M. H. Gunes. 2020. Latency Comparison of Cloud Datacenters and Edge Servers. In *Proc. 2020 IEEE Global Communications Conf. (GLOBECOM).* IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/GLOBECOM42002.2020.9322406

[5] J. Chen and X. Ran. 2019. Deep Learning With Edge Computing: A Review. *Proc. IEEE* 107, 8 (Aug. 2019), 1655–1674. https://doi.org/10.1109/JPROC.2019.2921977

[6] Z. Chen et al. 2017. An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance. In *Proc. 2nd ACM/IEEE Symp. on Edge Computing.* ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3132211.3134458

[7] S. Choy, B. Wong, G. Simon, and C. Rosenberg. 2012. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proc. 11th Annual Workshop on Network and Systems Support for Games (NetGames).* IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/NetGames.2012.6404024

[8] CNCF Staff. 2020. *CNCF Survey 2020.* Technical Report. Cloud Native Computing Foundation. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf Accessed on: May 31, 2021.

[9] G. Fairhurst, A. Sathiaseelan, and R. Secchi. 2015. *Updating TCP to Support Rate-Limited Traffic.* RFC 7661. RFC Editor.

[10] I. Fette and A. Melnikov. 2011. *The WebSocket Protocol.* RFC 6455. RFC Editor.

[11] P Garcia Lopez et al. 2015. Edge-Centric Computing: Vision and Challenges. *ACM SIGCOMM Comput. Commun. Rev.* 45, 5 (Sept. 2015), 37–42. https://doi.org/10.1145/2831347.2831354

[12] J. Gettys and K. Nichols. 2011. Bufferbloat: Dark Buffers in the Internet: Networks without Effective AQM May Again Be Vulnerable to Congestion Collapse. *ACM Queue* 9, 11 (Nov. 2011), 40–54. https://doi.org/10.1145/2063166.2071893

[13] M. Gorlatova, H. Inaltekin, and M. Chiang. 2020. Characterizing task completion latencies in multi-point multi-quality fog computing systems. *Comput. Netw.* 181 (Nov. 2020), 1–11. https://doi.org/10.1016/j.comnet.2020.107526

[14] M. Handley, J. Padhye, and S. Floyd. 2000. *TCP Congestion Window Validation.* RFC 2861. RFC Editor.

[15] G. Huston. 2003. Measuring IP Network Performance. *The Internet Protocol J.* 6, 1 (March 2003), 2–19.

[16] X. Jiang et al. 2019. Low-Latency Networking: Where Latency Lurks and How to Tame It. *Proc. IEEE* 107, 2 (Feb. 2019), 280–306. https://doi.org/10.1109/JPROC.2018.2863960

[17] N. Naik. 2017. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *Proc. 2017 IEEE Int. Systems Engineering Symp. (ISSE).* IEEE, New York, NY, USA, 1–7. https://doi.org/10.1109/SysEng.2017.8088251

[18] C. Nguyen, A. Mehta, C. Klein, and E. Elmroth. 2019. Why Cloud Applications Are Not Ready for the Edge (Yet). In *Proc. 4th ACM/IEEE Symp. on Edge Computing.* ACM, New York, NY, USA, 250–263. https://doi.org/10.1145/3318216.3363298

[19] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446. RFC Editor.

[20] W. Shi et al. 2016. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* 3, 5 (Oct. 2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[21] Z. Wu and H. V. Madhyastha. 2013. Understanding the Latency Benefits of Multi-Cloud Webservice Deployments. *ACM SIGCOMM Comput. Commun. Rev.* 43, 2 (April 2013), 13–20. https://doi.org/10.1145/2479957.2479960

[22] D. Xu et al. 2020. Understanding Operational 5G: A First Measurement Study on Its Coverage, Performance and Energy Consumption. In *Proc. Annu. Conf. ACM SIGCOMM.* ACM, New York, NY, USA, 479–494. https://doi.org/10.1145/3387514.3405882

[23] K. Zhang et al. 2017. Mobile-Edge Computing for Vehicular Networks: A Promising Network Paradigm with Predictive Off-Loading. *IEEE Veh. Technol. Mag.* 12, 2 (June 2017), 36–44. https://doi.org/10.1109/MVT.2017.2668838

A REPRODUCTION OF OUR RESEARCH

We aim to enable reproducibility of our results. In the following, we discuss our study in terms of repeatability (i.e., the ability of the same team to obtain the same results upon running the same measurement), replicability (i.e., the research can be performed by a different team with the same experimental setup), and reproducibility (i.e, the ability of independent teams to arrive at the same factual conclusion using their own tools and measurements).[1]

A.1 Repeatability

As the Internet is continuously evolving and data is gathered in live networks, repeated measurements may not necessarely yield the same results. Hence, introducing a natural limit for precise repeatability of our research. Nevertheless, to minimize the risk of errors and one-time effects, we repeated the measurements multiple times and over long periods, while explicitly highlighting in our paper as learnings the relevant and unexpected discrepancies.

A.2 Replicability

To make our research replicable, we release the source code of our measurement tools, our analysis scripts and the raw samples of the results presented in this paper on GitHub:

https://github.com/richiMarchi/latency-tester

A.3 Reproducibility

While strict reproduction of our results is difficult due to the reasons explained in Appendix A.1, as well as, in different cases, the dependency from the underlying network, an independent team is still able to obtain the same type of statistics presented in this work. Specifically, the paper describes the characteristics of the different scenarios considered and the configurations adopted for each measurement (e.g., endpoints distance, message size and frequency). As for the infrastructure, unless explicitly specified in the text, we always leveraged default configurations, both for the Linux Kernel (as supplied in Ubuntu 20.04 LTS), the Kubernetes clusters (version 1.19 and newer) and the AWS EC2 instances.

[1] Dror G. Feitelson. 2015. From Repeatability to Reproducibility and Corroboration. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 3–11. https://doi.org/10.1145/2723872.2723875