Flow control for Latency-Critical RPCs

Marios Kogias
EPFL

Edouard Bugnion
EPFL

ABSTRACT

In today's modern datacenters, the waiting time spent within a server's queue is a major contributor of the end-to-end tail latency of µs-scale remote procedure calls. In traditional TCP, congestion control handles in-network congestion, while flow control was designed to avoid memory overruns in streaming scenarios. The latter is unfortunately oblivious to the load on the server when processing short requests from multiple clients at very high rates. Acknowledging flow control as the mechanism that controls queuing on the end-host, we propose a different flow control mechanism that depends on the application-specific service-level objectives and controls the waiting time in the receivers queue by adjusting the incoming load accordingly. We design this latency-aware flow control mechanism as part of TCP by maintaining a wire-compatible header format without introducing extra messages. We implement a proof-of-concept userspace TCP stack on top of DPDK and we show that the new flow control mechanism prevents applications from violating service-level objectives in a single-server environment by throttling the incoming requests. We demonstrate the true benefit of the approach in a replicated, multi-server scenario, where independent clients leverage the flow-control signal to avoid directing requests to the overloaded servers.

CCS CONCEPTS

• **Networks → Transport protocols**;

KEYWORDS

flow control, TCP, Remote Procedure Call, latency-critical

ACM Reference Format:
Marios Kogias and Edouard Bugnion. 2018. Flow control for Latency-Critical RPCs. In *KBNets'18: ACM SIGCOMM 2018 Afternoon Workshop on Kernel Bypassing Networks , August 20, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229538.3229541

1 INTRODUCTION

The "cloud era" is built on top of at least two nearly ubiquitous paradigms: TCP/IP and remote procedure calls (RPC) [10]. Both paradigms are used to connect to mega-datacenters (*e.g.,* `https`) as well as within datacenters, where they connect multiple tiers of servers with wide fan-in/fan-out flow patterns and strict tail-latency service level objectives (SLO) [7, 14]. TCP has emerged as the main transport protocol on top of commodity Ethernet for latency-critical RPCs. The reliable, ordered byte stream provided by TCP serves as the basis for other higher-level, application abstractions and guarantees, such as exactly-once RPC semantics. Non-commodity alternatives, *e.g.,* RDMA over Infiniband and RDMA over Lossless Ethernet (RoCE) have specific hardware requirements or expose alternative APIs, that limits their wide adoption, despite potentially reduced round-trip times (RTT).

A datacenter, though, differs radically from the assumptions considered during the initial TCP design [13]. Specifically, modern datacenters are uniformly designed and built upon low-latency Ethernet fabrics in Clos topologies. Those fabrics comprise of commodity cut-through switches with shallow buffers that have a few hundreds of nanoseconds of switching latency. Such a design guarantees unloaded RTTs in the scale of µs and a few Pbps of bisection bandwidth [40]. Despite TCP's extensive use within a datacenter, most of its mechanisms remain as originally designed, such as the sliding window. TCP's sliding window is managed by both the congestion and flow control logic. We make the distinction between congestion control algorithms focusing on in-network congestion, while flow control focusing on queueing on the end-hosts. Improvements throughout the networking stack, though, have mostly focused on the in-network congestion. Approaches such as [2–4] reduced congestion and buffer utilisation, nearly eliminated packet drops, improved fabric utilisation, and reduced latency jitter.

In contrast, the endpoints have received less consideration. In particular, the core flow-control mechanism of TCP is used today primarily to prevent packet loss at the end-hosts, but without any particular consideration for end-to-end latency. The initial purpose of flow control is to avoid overwhelming the receiver, which can lead to packet drop. This was a significant concern during the initial design of TCP due to the limited amount of DRAM in computers of a few decades ago. Modern servers nowadays have abundant DRAM, and can potentially accommodate a large number of packets. Applications with strict tail-latency SLOs, though, require minimal queueing on the server side. For a datacenter RPC server that handles µs-level requests from multiple sockets (fan-in), the queuing on the server side will account for a significant proportion of the client end-to-end latency, given that the service time and the RTT combined account for a few µs. Latency-critical RPCs, such as key-value stores, usually depend on short requests and responses of a few bytes [6]. So, an in-memory, latency-sensitive RPC server will violate a strict SLO after queuing only a few dozens of short messages, which collectively require only a tiny fraction of the server's memory.

This paper suggests to revisit the notion of flow-control specifically for latency-sensitive, µs-scale, kernel-bypassed RPC services on top of TCP. According to our proposal, the destination signals its availability back to the source in an application-independent manner, as with TCP's standard flow control. Unlike the standard approach, though, the signal is based on the expected wait and

KBNets'18, August 20, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5909-2/18/08…$15.00
https://doi.org/10.1145/3229538.3229541

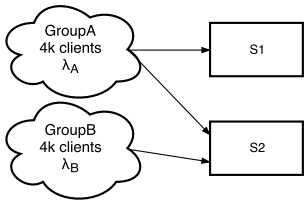


Figure 1: Replicated RPC experiment setup

service times of the server and takes into consideration the specific application-level tail-latency SLOs (rather than available memory).

Our flow-control mechanism uses a token bucket per connection to control the incoming request rate and maintain it at a level so that the application SLOs are not violated. The clients self-pace their requests based on available tokens. The server controls the size and the fill-rate of the bucket based on (a) the trailing estimate of the service time distribution of RPCs, (b) the SLO, and (c) the number of connections.

We implement a proof-of-concept TCP stack with a latency-aware flow control mechanism by simply re-purposing the TCP sliding window. The new mechanism requires no changes to the TCP header format or additional messages. Our implementation on top of Intel's DPDK [15] is suitable to evaluate the effectiveness of the approach for µs-scale tasks. Our evaluation of the mechanism in a series of synthetic microbenchmarks with different service time distributions shows that it can accurately identify the load that will violate the latency SLO and maintain the throughput at that level. Moreover, we show the benefits of using this mechanism in cases of replicated services in order to avoid overloaded servers.

2 MOTIVATION

Flow control is one of the profound examples of Salzer's end-to-end argument [38]. In a latency-aware flow control mechanism, the two ends, client and server, communicate, so that the client adjusts their offered load after the server's directives, based on how the server performs and the agreed SLO. In a scenario with replicated RPC-servers, this mechanism can be used to dynamically adjust the incoming load to the servers, so that the application tail-latency SLOs are not violated. Figure 1 provides a motivating example that applies the end-to-end argument for replicated RPC services: 8K clients are split into two groups, where the first group can interchangeably select between server 1 (*S1*) and server 2 (*S2*), but the second group (for some reason) can only use *S2*. The relative arrival rate of requests from the two groups is unknown and varying, and the service time to process each request is also unknown. By incorporating tail-latency awareness into flow control, each client should be able to independently determine when and where to send requests so that the SLO is never violated.

3 BACKGROUND

Existing TCP Flow Control: TCP, as a connection-oriented transport protocol, implements flow control per connection. The flow control mechanism depends on the size of the available receive socket buffer communicated between the two endpoints with every packet exchanged. The sender uses this information in conjunction with the current congestion window size to decide how many bytes to send, and sends the maximum amount allowed.

From an implementation perspective, flow control is implemented via a 16-bit header field, with window-scale option exchanged during the 3-way handshake [21]. On each endpoint the size of the receive buffer can be configured per connection during runtime through a `setsockopt()` call. However, it can not be configured to less than 4kB. A server with a high client count (high fan-in) must therefore accept potentially multiple RPCs from each client, before the existing flow control starts throttling each client independently.

Figure 2a summarises the existing TCP flow control mechanism. It shows a connection between a server and a client. The server also has second connection to another client. There are buffers on both ends of the connection that are partially occupied. TCP's existing flow control mechanism signals back the available per connection buffer space (B) as part of the packets exchanged.

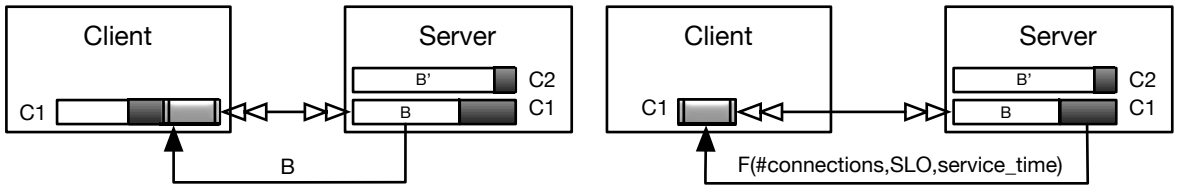
TCP Flow Control in KBNets: Kernel-bypass networking introduces additional issues related to the implied semantics of flow-control. TCP's flow control accounts for the bytes processed by the TCP stack but *not* processed by the application. Thus, it assumes certain asynchrony between network and application processing. This assumption is invalid for kernel-bypass implementations with a symmetric, run-to-completion design inspired by middlebox dataplanes[8, 37]. Because of the tight coupling of network and application processing, there are no buffered data between the two stages. Thus, the semantics of TCP's flow control become vague. It appears as if receiver buffers are never filled up, despite incoming packets being queued or dropped before TCP and application processing. The alternative approach dedicates threads to network processing, other threads to application processing, and the two groups communicate over interprocess communication, *e.g.,* shared memory [22]. Here, the TCP flow control semantics remain unchanged.

4 DESIGN

Our goal is to reduce and control buffering on the path of an RPC on top of TCP. To do so, we design a latency-aware flow control mechanism for TCP, specifically targeting latency-critical RPC services.

Figure 2b summarises the proposed design. On the RPC-client side, buffering is limited to the sliding window, which ensures that requests can be sent without delay to the server. On the server side, we change the semantics of the flow control signal to take into consideration the total amount of connections on the server, and the application SLOs and service time. The flow-control logic on the server should be able to predict the overall incoming load that will violate the latency SLO and maintain throughput below that level across all connections. We set the following design requirements: (a) There should not be extra messages specifically for flow control. (b) The TCP header format should remain intact. (c) The mechanism should be agnostic to the service time distribution.

Predicting the SLO-violating load: Since we want to leverage flow control, a throughput-oriented mechanism, to control the end-to-end request latency, we need to understand and approximate the



(a) TCP Flow Control (b) Latency-Aware Flow Control

Figure 2: One-way Flow Control FeedBack Loop. C_i indicates a connection, dark grey indicates occupied buffer space, light grey indicates the slidiing window, B is the available receive buffer space.

correlation between the incoming rate of requests and the end-to-end latency of each request. To do so, we use some basic queueing theory. Although TCP is a connection-oriented transport protocol, we focus on RPC services and abstract the basic system functionality. Thus, a system can be described by its service time distribution, the incoming distribution of requests, and the number of workers and queues. We use Kendall's notation to describe queuing models, where in the following expression `A/S/n`, `A` is the inter-arrival distribution, `S` is the service time distribution, and `n` is the number of workers. The scheduling policy implied is first-come-first-served (FCFS). The end-to-end request latency is the sum of the propagation delay, the wait time in the queue, and the service time. The propagation delay depends on the request size and the network characteristics. The service time distribution depends on the application. Thus, we focus on how to control the wait time and provide wait time SLOs (WSLOs), namely an upper limit on the time each request might wait to be served.

According to queuing theory, the average wait time in an `M/G/1` system can be expressed in a closed form that depends on the system load, and the average and standard deviation of the service time. This implies that in any system with a Poisson inter-arrival distribution and a single worker, we can control the average wait time by controlling the system load, independently of the service time distribution. Equation 1 expresses the average wait time (T_w), as a function of the system load (ρ), and the service time standard deviation (σ_{T_s}) and average (T_s) [27].

$$T_w = 1/2\frac{\rho T_s}{1-\rho}[1 + (\frac{\sigma_{T_s}}{T_s})^2] \tag{1}$$

SLOs, though, are expressed at some percentile. Since there is no closed form that expresses the wait time percentiles, we will use the central limit theorem for heavy traffic queueing systems [24] to approximate them. According to that, in any G/G/N queuing system under heavy traffic load, the wait time distribution could be approximated by an exponential distribution. Based on that we can approximate any percentile of the wait time distribution, since we know the distribution to be exponential and its average from Equation 1. Vice versa, we can set an upper limit in the wait time for a certain percentile, and predict the system load that will violate this WSLO. To do so, first we need to estimate the average wait time Tw_{target} at the SLO. Equation 2 computes the average of an exponential distribution whose value at the p-percentile is Tw_{slo}. β is the scale parameter of the exponential distribution. If we substitute Tw with Tw_{target} in Equation 1, and solve for ρ, we get the load that will violate the WSLO. For example, for a fixed service time distribution ($\sigma_{T_s} = 0$) with $Ts = 10$, and a 99-th percentile WSLO of 100($Tw_{slo} = 100$ and $p = 0.99$), we predict that the WSLO will be violated when $\rho = 0.81$

$$\begin{aligned} Pr[X \leq Tw_{slo}] &= p \\ 1 - e^{-\frac{Tw_{slo}}{\beta}} &= p \\ Tw_{target} = \beta &= -\frac{Tw_{slo}}{\ln(1-p)} \end{aligned} \tag{2}$$

Flow-control Mechanism: We incorporate the above result in the existing TCP's flow control infrastructure. The operator should only define the WSLO as the amount of wait time allowed at a certain percentile and the system should adjust accordingly. Although TCP connections are symmetric, in the following explanation we focus on the case of an RPC server (receiver) that needs to apply flow control to the incoming requests from different clients (senders).

We first describe the functionality of the receiver running a single RPC service. For simplicity, we consider a single-threaded RCP server and we analyse the multi-threaded case at the end of the section. The receiver calculates the maximum allowed incoming rate according to the WSLO, based on the formulas in Section 4, and prevents the clients from sending faster than this rate. To estimate the average and standard deviation of the service time, every receiver maintains a global moving average of the application service time per request. Moreover, to avoid unnecessary client throttling, every receiver maintains a moving average of the request inter-arrival time across all connections, and throttles only when the overall load is close to the maximum allowed one.

The throttling mechanism depends on a per-connection token bucket algorithm controlled by the receiver. Every token corresponds to a request. If the incoming rate is lower than a certain percentage of the maximum allowed load, the receiver allocates a fixed number of tokens per connection. For our experiments we set the threshold at 80% of the maximum allowed load. After that threshold throttling is necessary. The receiver distributes the maximum allowed load across all connections. Load distribution can consider any connection priority scheme depending on the application logic. The receiver translates the per connection load to a number of tokens per connection that periodically replenishes, and communicates the amount of available tokens to the receiver

through the TCP header. If there are no tokens available, the sender should remain silent till the new replenishment. To do so without the need of extra messages, the receiver notifies the sender about the duration it should remain silent, again through the TCP header.

Unlike the existing TCP semantics that depend on the socket API with intermediate buffering on the send path, we assume a zero-sized send buffer other than the sliding window. If there is a request to be sent, the sender immediately sends it as long as there are available tokens (one token per request). If not the send fails. Thus, the application can decide whether it should drop the request, buffer it in the application space and wait, or try using another connection, if any. Compared to the existing TCP implementation, sends will fail more often. A failed send, though, now implies that if this request was actually send, it would probably be an SLO violation. The sender can send again either after a token replenishment, or at the end of the idle period as defined by the receiver.

A multi-threaded server would operate on a similar rational. We assume that every thread serves incoming requests independently from the others, and there is a static mapping of connections to threads. Every thread runs a run-to-completion loop. This is a popular design adopted by latency critical applications, such as NGINX [34] and memcached [30], and operating systems, such as IX [8]. In this design, every thread maintains its own moving averages for service time and inter-arrival distribution that correspond to the connections it is responsible for, and implements flow control based on the load it individually receives.

Assumptions and Limitations: The use of closed forms in the above design depends on the two following assumptions: (1) the RPC inter-arrival distribution is Poisson; (2) the system is modelled as $M/G/1$, namely each core operates as a standalone entity and there is no connection sharing across cores. If either of the two assumptions is violated, we can still use heuristics to correlate the incoming load with the queueing delay as part of a training phase, and then use the same mechanism to keep the incoming load below the threshold that emerged after the training phase.

Finally, note that our latency-aware flow control logic is integrated within TCP without using any extra packets, but is in fact independent of the actual transport used. For example, it could trivially be implemented on top of RDMA by using either extra control messages from the server back to the clients, or by having the clients read the load from the server using one-sided remote read operations.

5 IMPLEMENTATION

We implemented the above design in a userspace TCP/IP stack and used it in a sample RPC client and server that are built on top of Intel's DPDK [15]. The TCP/IP stack does not make any assumption about the kernel-bypass paradigm, namely, how to split network and application processing across threads, and can be used both in a symmetric and an asymmetric setup. We built the client and server applications after the symmetric paradigm, where every thread runs both network and application processing.

Since we specifically target latency-critical RPC services, we didn't expose a POSIX-like API for our proof of concept TCP stack. Instead, we implemented an event-based API similar to IX [8]. In this API, the application has direct access to the packets received. Whenever there is an incoming packet, the application gets notified and receives a pointer and the length of the received payload. Similarly, whenever the application wants to send a packet, writes directly to the `mbuf` to avoid extra copies. Before sending, every sender explicitly asks for a new `mbuf` to put the data to be sent at the right offset. Finally, we implemented two extra function calls to collect the per RPC application processing time. The application calls these two functions at the beginning and end of processing each individual RPC, equivalently.

We implemented the proposed flow-control mechanism by leveraging the existing 16-bit receive window without introducing any extra messages or changing the TCP header. Unlike the existing semantics though, where this field carries a number of bytes, we need to overload it with dual semantics, since it can represent either a number of tokens or the duration of idle time. We use the most significant of the 16 bits as a mode switch. If not set, the lower 15 bits encode the number of available tokens. If set, they carry the number of µs for which this particular connection should remain idle. By configuring the token replenishment rate equivalently, 15 bits are enough to carry the available tokens per connection. In our experiments we replenished each connection's tokens every 500 µs. For the idle time, we assume that no connection should remain idle for more than 32 ms, which is approximately the maximum duration in µs encoded in 15 bits. If for any reason, the 15 bits are not enough, we can employ the existing window scaling mechanism.

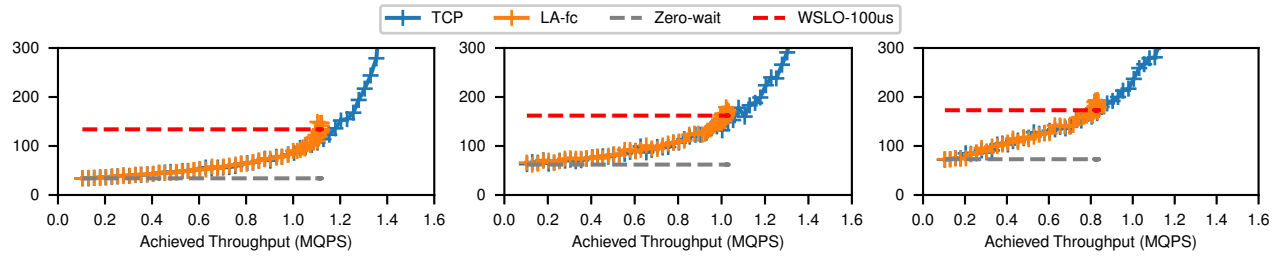
We didn't implement any congestion control in our proof-of-concept TCP stack, since our experiments are application-throttled. Existing window-based congestion control schemes are not effective for this particular type of workload consisting of very small messages [6] that fit in a single packet, across a large number of connections, in an environment with µs RTTs. Congestion control for such workloads is an open research problem [11, 12, 31, 47] with recent research proposals [33] suggesting a generic congestion control agent that can be used in kernel-bypass networking stacks. Such an approach is a good fit for our networking stack, but we leave this for future work.

6 EVALUATION

To evaluate our design we implemented a synthetic RPC service based on our experimental TCP stack. We use fixed-size requests and responses of 8 bytes. Each request encodes the amount of processing time required by the server. Once the server receives the request, spins for the amount of time specified and echoes the received value back. This way we can emulate any service time distribution and evaluate whether our system can adjust accordingly.

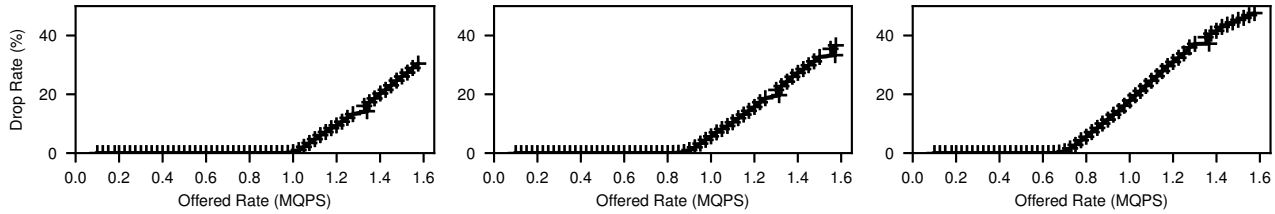
Our experimental setup consists of a cluster of 4 client and 2 server machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and the server machines are Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect the clients and the server to the switch through a single NIC port each. On both clients and servers we use only the one NUMA node.

We run two types of experiments. In the first type we emulate multiple independent clients having a single connection to the RPC



(a) Fixed ($\bar{S} = 10\mu s$) **(b) Exponential ($\bar{S} = 10\mu s$)** **(c) Bimodal ($\bar{S} = 10\mu s$)**

Figure 3: 99th percentile tail latency versus throughput for the three service time distributions with 10µs mean service time and the two flow control mechanisms, TCP and the proposed latency-aware (LA-fc).



(a) Fixed **(b) Exponential** **(c) Bimodal**

Figure 4: Drop rate as a function of the offered throughput in the case of LA-fc for the 3 service time distributions.

server and we show how the system can identify the load that will violate WSLO, and maintain the incoming request rate lower than that level. In the second type of experiment, we show how this mechanism can be used in a replicated RPC service, so that clients avoid overloaded replicas, and thus achieving better tail-latency.

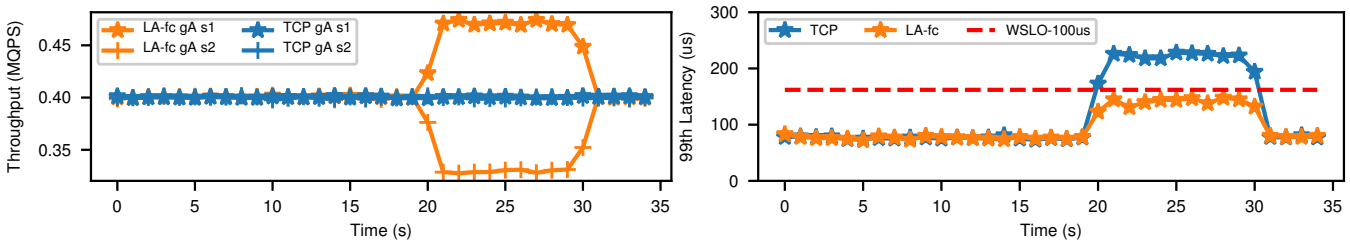
Single-node RPC service: In this experiment we use 4 client machines and 1 server machine. Each client machine opens 1024 connections spread across 8 threads. Every client thread generates requests with a Poisson inter-arrival distribution, randomly chooses one of the available connections, and sends the request to the server. This workload is equivalent to 4096 independent clients generating requests with a Poisson inter-arrival against the RPC server. If flow control allows it, the client sends the request to the server. Otherwise, the request is dropped. Note, that this is an extreme case that will help us evaluate the maximum throughput loss because of the new flow control mechanism. In a more realistic scenario, the client could decide to wait for a certain amount of time for the connection to become available before dropping the request depending on the application SLOs.

We use three different service time distributions with the same average of 10µs, but different dispersion. We consider a fixed, an exponential and a bimodal distribution where 90% of the requests require 5 µs of processing time and the rest 10% requires 55 µs. We run the client and the server both with the standard TCP flow control mechanism and with the proposed latency-aware flow control mechanism, and we set a WSLO of 100µs at the 99-*th* percentile.

Figure 3 shows the 99-*th* percentile end-to-end latency in µs measured at the client as a function of the achieved throughput in million requests per seconds for the two flow control mechanisms, TCP and latency-aware. The server runs on 16 cores, so with an average service time of 10µs, the theoretically maximum possible throughput is 1.6 million requests per second. We limit the y-axis at 300 µs. The red dashed line is the WSLO defined as 100 µs above the unloaded latency. Firstly, we observe that in the case of TCP flow control, latency can increase beyond the SLO. TCP is agnostic to the application SLO and and there are multiple clients with outstanding requests. In the case of latency-aware flow control, the system manages to throttle the incoming rate according to the WSLO. Moreover, we can see that the maximum allowed load changes according to the service time distribution. This load increases as the service time dispersion decreases.

In this experiment we traded the drop rate for better tail-latency, since clients instantly drop their request if the connection is not available. Figure 4 shows the drop rate as a function of the offered request rate for the same three distributions of service time. In this plot, it becomes obvious that our mechanism can decide when to start throttling based on the different service times. Clients start dropping requests at different request rates across the three service time distributions. As expected, as the service time dispersion increases, client start dropping requests earlier.

Multi-node RPC service: This experiment addresses the motivating example of Figure 1 by showing how the mechanism can be used by clients of a replicated RPC service to avoid the overloaded replicas. We assume a replicated RPC service across two servers where the two servers, *S1* and *S1*, can be used interchangeably. We split the client machines into two groups. GroupA emulates 4096



(a) Throughput over time

(b) Latency over time

Figure 5: GroupA (gA) throughput against server1 (s1) and server2 (s2), and 99th percentile tail latency for the two flow control mchanisms, TCP and the proposed latency-aware (LA-fc)

independent clients that have one connection to each server and generate requests with a Poisson inter-arrival (λ_A). According to the client logic, each client randomly chooses between the two connections. If the chosen connection can be immediately used for sending, they send their request. Otherwise they try their other connection. If this one is available, they use this to send the request, else they drop it. GroupA generates a constant load of 800 kQPS with a 10-µs exponential service time against the two servers. The second group of clients, groupB, again consists of 4096 independent clients but they can only use *S2*. GroupB clients generate a constant load of 800 kQPS with a 10-µs exponential service time starting at $t_0 = 20$ for 10 seconds and then they stop. We measure and plot the throughput and latency achieved by groupA for the two flow control mechanisms, TCP and latency-aware.

Figure 5a shows the achieved throughput for groupA, and how it is distributed across the two servers. In the case of TCP, groupA selects both *S1* and *S2* with the same likelihood, independently of groupB's behaviour. However, in the case of the latency-aware flow control mechanism, groupA starts showing preference to *S1* at $t_0 = 20$. After groupB starts loading, *S2* operates beyond its WLSO, so it starts throttling. As a result groupA's connection to *S2* can not support the same load as before. Thus, *S1* is chosen more often. GroupA's request rate to *S1* increases, and the rate to *S2* decreases, while the aggregate throughput remains the same. The drop rate for groupA is close to zero. Regarding groupB throughput, although not shown in the graph, it is 800 kQPS in the case of TCP, while it drops to around 660 kQPS in the case of latency-aware flow control. This is in accordance with the results in Figure 3b for *S2*, since 1 MQPS (330 kQPS from groupA and 660 kQPS from groupB)is approximately the maximum allowed load for the exponential distribution with a WSLO of 100 µs. We could avoid dropping groupB's throughput by choosing a different load allocation policy in *S2* that favours groupB connections.

Figure 5b shows the end-to-end tail-latency as it is measured by the groupA clients, by choosing either *S1* or *S2* according to the client logic. The red dashed line is the WSLO for the exponential distribution as shown in Figure 3b. As expected, groupA tail-latency increases when groupB starts loading server2. In the case of TCP, though, the WSLO is violated since *S2* is still selected with the same likelihood. As a result, *S2* serves approximately 1.2 MQPS (800 kQPS from groupB and 400 kQPS from groupA) resulting in a 99-th percentile latency around 230 µs. This result is in accordance with the TCP result in Figure 3b. The latency-aware flow control mechanism manages to adjust the *S2* throughput so that the WSLO is not violated, given that groupA's 99-th percentile latency is kept below 160 µs.

7 RELATED WORK

Several kernel-bypass systems implement TCP/IP stacks [8, 16, 17, 22, 28, 29, 35–37, 39], and remain fully compatible with the exact TCP semantics. Similarly to TCP, protocols such as QUIC [26] and HTTP2 [9], also, implement flow control mechanisms based on buffer occupancy, and remain agnostic to latency SLOs.

Token bucket algorithms are the canonical way of implementing flow control in any kind of communication in fields such as wide-area networks [43], networks-on-chip [23], storage [25, 42, 44, 46], and network congestion control [12].

There are several research proposals on changing the semantics or implementation of TCP, according to application needs, such as sharing the congestion window [20] for faster convergence, advertising the send buffer occupancy for better load balancing and network utilization [1], or adaptively changing the send buffer size for streaming [18].

Finally, there a several research proposals dealing with selecting servers in a replicated service and reducing tail-latency because of strugglers [5, 14, 19, 32, 41, 45]

8 ACKNOWLEDGEMENTS

The authors want to thank Katerina Argyraki, Jonas Fietz, Adrien Ghosn and George Prekas for the early discussions on specializing TCP for low-latency RPCs. This work is funded by a VMware Research Grant.

9 CONCLUSION

We presented a latency-aware flow control mechanism specifically for latency-critical RPCs that use TCP. We introduced the notion of wait time SLO and we showed how our flow control mechanism prevents applications from violating their WSLOs. Finally, we showed how this mechanism can benefit clients of replicated services to avoid overloaded replicas, and thus reduce the end-to-end tail-latency.

REFERENCES

[1] Alexandru Agache and Costin Raiciu. 2015. Oh Flow, Are Thou Happy? TCP Sendbuffer Advertising for Make Benefit of Clouds and Tenants.. In *HOTCLOUD*.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: distributed congestion-aware load balancing for datacenters.. In *SIGCOMM*. 503–514. https://doi.org/10.1145/2619239.2626316

[3] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP).. In *SIGCOMM*. 63–74. https://doi.org/10.1145/1851182.1851192

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center.. In *NSDI*. 253–266.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones.. In *NSDI*. 185–198.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store.. In *SIGMETRICS*. 53–64. https://doi.org/10.1145/2254756.2254766

[7] Luiz André Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54. https://doi.org/10.1145/3015146

[8] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2017. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4 (2017), 11:1–11:39. https://doi.org/10.1145/2997641

[9] M. Belshe, R. Peon, and M. Thomson. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard). , 96 pages. https://doi.org/10.17487/RFC7540

[10] Andrew Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* 2, 1 (1984), 39–59. https://doi.org/10.1145/2080.357392

[11] Bob Briscoe and Koen De Schepper. 2015. *Scaling TCP's Congestion Window for Small Round Trip Times.* Technical Report TR-TUB8-2015-002. BT, Bell Labs.

[12] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters.. In *SIGCOMM*. 239–252. https://doi.org/10.1145/3098822.3098840

[13] David D. Clark. 1988. The design philosophy of the DARPA internet protocols.. In *SIGCOMM*. 106–114. https://doi.org/10.1145/52324.52336

[14] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. https://doi.org/10.1145/2408776.2408794

[15] dpdk [n. d.]. Data Plane Development Kit. http://www.dpdk.org/.

[16] dpdk-ans [n. d.]. ANS (Accelerated Network Stack) on DPDK. https://github.com/ansyun/dpdk-ans.

[17] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science* 2 (2001), 77.

[18] Ashvin Goel, Charles Krasic, and Jonathan Walpole. 2008. Low-latency adaptive streaming over tcp. *TOMCCAP* 4, 3 (2008), 20:1–20:20. https://doi.org/10.1145/1386109.1386113

[19] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface.. In *SOSP*. 168–183. https://doi.org/10.1145/3132747.3132774

[20] Safiqul Islam and Michael Welzl. 2016. Start Me Up: Determining and Sharing TCP's Initial Congestion Window. In *Proceedings of the 2016 Applied Networking Research Workshop (ANRW '16)*. ACM, New York, NY, USA, 52–54. https://doi.org/10.1145/2959424.2959440

[21] V. Jacobson, R. Braden, and D. Borman. 1992. TCP Extensions for High Performance. RFC 1323 (Proposed Standard). , 37 pages. https://doi.org/10.17487/RFC1323 Obsoleted by RFC 7323.

[22] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.. In *NSDI*. 489–502.

[23] Natalie D. Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. *On-Chip Networks, Second Edition.* Morgan & Claypool Publishers.

[24] J. F. C. Kingman. 1961. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57, 4 (1961), 902âĂŞ904. https://doi.org/10.1017/S0305004100036094

[25] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash.. In *ASPLOS-XXII*. 345–359. https://doi.org/10.1145/3037697.3037732

[26] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment.. In *SIGCOMM*. 183–196. https://doi.org/10.1145/3098822.3098842

[27] Jean-Yves Le Boudec. 2010. *Performance Evaluation of Computer and Communication Systems.* EPFL Press, Lausanne, Switzerland.

[28] libuinet [n. d.]. libuinet a library version of FreeBSD's TCP/IP stack. https://github.com/pkelsey/libuinet.

[29] Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2014. Network stack specialization for performance.. In *SIGCOMM*. 175–186. https://doi.org/10.1145/2619239.2626311

[30] memcached [n. d.]. Memcached. https://memcached.org/.

[31] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter.. In *SIGCOMM*. 537–550. https://doi.org/10.1145/2785956.2787510

[32] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (2001), 1094–1104.

[33] Akshay Narayan, Frank Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. 2017. The Case for Moving Congestion Control Out of the Datapath.. In *HOTNETS-XVI*. 101–107. https://doi.org/10.1145/3152434.3152438

[34] nginx [n. d.]. NGINX. https://www.nginx.com/.

[35] ofp [n. d.]. Open FastPath. http://www.openfastpath.org/.

[36] ool [n. d.]. Open OnLoad. http://www.openonload.org/.

[37] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2016. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4 (2016), 11:1–11:30. https://doi.org/10.1145/2812806

[38] Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (1984), 277–288. https://doi.org/10.1145/357401.357402

[39] seastar [n. d.]. Seastar Project. http://www.seastar-project.org/networking/.

[40] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network.. In *SIGCOMM*. 183–197. https://doi.org/10.1145/2785956.2787508

[41] P. Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection.. In *NSDI*. 513–527.

[42] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony I. T. Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: a software-defined storage architecture.. In *SOSP*. 182–196. https://doi.org/10.1145/2517349.2522723

[43] Jonathan Turner. 1986. New directions in communications(or which way to the information age?). *IEEE communications Magazine* 24, 10 (1986), 8–15.

[44] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph A. Becker-Szendy. 2006. Zygaria: Storage Performance as a Managed Resource.. In *RTAS*. 125–134.

[45] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services.. In *NSDI*. 543–557.

[46] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage.. In *SOCC*. 29:1–29:14. https://doi.org/10.1145/2670979.2671008

[47] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments.. In *SIGCOMM*. 523–536. https://doi.org/10.1145/2785956.2787484