Tail-tolerance as a Systems Principle not a Metric

Marios Kogias
EPFL, Switzerland

Edouard Bugnion
EPFL, Switzerland

ABSTRACT

Tail-latency tolerance (or just simply tail-tolerance) is the ability for a system to deliver a response with low-latency nearly all the time. It it typically expressed as a system metric (*e.g.,* the 99th or 99.99th percentile latency) or as a service-level objective (*e.g.,* the maximum throughput so that the tail latency is below a desired threshold).

We advocate instead that modern datacenter systems should incorporate tail-tolerance as a core systems design principle and not a metric to be observed, and that tail-tolerant systems can be built out of large and complex applications whose individual components may suffer from latency deviations. This is analogous to fault-tolerance, where a fault-tolerant system can be built out of unreliable components.

The general solution is for the system to control the applied load and keep it under the threshold that violates the latency SLO. We propose to augment RPC semantics with an architectural layer that measures the observed tail latency and probabilistically rejects RPC requests maintaining throughput under the threshold that violates the SLO. Our design is application-independent, and does not make any assumptions about the request service time distribution.

We implemented a proof of concept for such a tail-tolerant layer using programmable switches, called SVEN. We demonstrate that the approach is suitable even for microsecond-scale RPCs with variable service times. Moreover, our approach does not induce measurable overheads, and can maintain the maximum achieved throughput very close to the load level that would violate the SLO without SVEN.

ACM Reference Format:
Marios Kogias and Edouard Bugnion. 2020. Tail-tolerance as a Systems Principle not a Metric. In *4th Asia-Pacific Workshop on Networking (APNet '20), August 3–4, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3411029.3411032

APNet '20, August 3–4, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-8876-4/20/08…$15.00
https://doi.org/10.1145/3411029.3411032

1 INTRODUCTION

Datacenter applications, such as web-search, e-commerce, social-networking, *etc.* need to operate under strict Service-Level-Objectives (SLO) for their tail-latency. Complying or violating those SLOs instantly reflects on user satisfaction and engagement. Thus, there is a tremendous on-going effort both in academia and industry in building low-latency systems for microsecond-scale computing [2].

Single-digit round-trip times inside a datacenter are considered commonplace with recent advancements in hardware and software. Dataplane operating systems, such as IX [3], and Arrakis [28] and kernel-bypassing techniques have reduced the system overheads present in commodity operating systems. µs-scale schedulers such as ZygOS [29], and Shinjuku [17] managed to reduce tail-latency even further compared to dataplanes, with smarter scheduling. Systems such as Shenango [25] brought efficiency while maintaining the performance benefits. On the networking side, datacenter-specific congestion control algorithms, such as Homa [23], deal with tail behaviours coming from in-network delays by almost eliminating in-network queueing. However, all of the previous efforts approach tail-tolerance as a metric, and report the achieved throughput at a specific latency SLO in steady state when the offered load is below the system capacity, ignoring the system behaviour under more realistic unpredictable conditions with load bursts.

A tail-tolerant system, on the other hand, is designed to operate according to a specific latency SLO and minimizes the SLO violations under any conditions, trading throughput for predictable behaviour. In this paper we advocate that tail-tolerance should be approached as a system design principle instead of a best-effort metric.

We draw inspiration from research on fault-tolerant systems. The latency of a complex fan-out request is defined by the slowest sub-request [6]. Similarly the mean time to failure of a complex system was defined by the mean time to failure of its most failure-prone sub-component before the advancement in fault-tolerance research. However, reasoning about fault-tolerance is a ubiquitous part of the system design process today. System designers, leveraging the appropriate form of redundancy, provision for their systems according to the expected risk of failure and can provide explicit availability guarantees. For example, a primary backup [4] system can tolerate f failures out of $f + 1$ replicas, state machine

replication [22, 24] f out of $2f + 1$, while and tandem processes [12] one out of two, *etc.* There is no similar systematic approach to reason about tail behaviours, though.

As a first step towards systematically tail-tolerant systems, we design SVEN (**S**LO **V**iolations **E**limi**N**ator), a system to control the tail behaviour of latency critical services. SVEN operates at the remote procedure call (RPC) level and is part of the RPC transport or library, thus remaining application agnostic. It is orthogonal to any existing scheduling or flow control mechanisms, and can be used at any level inside a fan-out/fan-in application, namely at the aggregator nodes or the leaves. SVEN performs dynamic RPC admission control based on the currently observed latency distribution aiming to keep the incoming load below the threshold that violates the target latency SLO.

We implemented an initial SVEN proof of concept on top of R2P2 [21] that runs on a Barefoot Tofino ASIC [1]. Our evaluation shows that SVEN can identify the load level that violates the latency SLO across a variety of service time distributions without any application-specific configuration, and maintain throughput below that level, even in cases where the in-coming load was above the system capacity. SVEN splits its functionality between the programmable switch's control and dataplane and does not consume any server resources.

2 MOTIVATION AND BACKGROUND

In this section we describe existing system mechanisms and why they fell short in implementing tail-tolerance. Then, we analyze the need and the fit for a tail-tolerant design based on admission control for datacenter applications. Finally, we briefly introduce R2P2 [21] a transport protocol for datacenter RPCs that offers the right abstraction for our design, on which we built SVEN.

2.1 Scheduling and Flow Control

Existing systems at any scale, single-node or distributed, implement two basic functionalities, scheduling and flow control. Scheduling determines the order at which requests are executed in the system and the dedicated resources to be used. Flow control is used to match the producer and the consumer pace, it guarantees that the system has enough capacity to serve new incoming requests, and creates backpressure if that is not the case.

Scheduling has a significant impact on the end-to-end request latency. Thus, different scheduling policies and systems have been proposed for different usecases and timescales. Examples of scheduling policies are FIFO in ZygOS [29] and Processor Sharing in Shinjuku [17] for CPU resources, or TCP's fair share and Homa's *Shortest-Remaining-Processing-Time* (SRPT) for network resources. Scheduling is a fine-grained mechanism that operates at the request level, making decisions for each individual request, considering execution time, and trying to minimize latency, while it is evaluated based on specific metrics, *e.g.,* throughput at the latency SLO, flow completion time, *etc.*.

Flow control, on the other hand, is a coarse-grained mechanism to prevent system or performance collapse when the system is under heavy load. It is SLO-agnostic and in most cases it depends on the available memory on the system to make decisions. While there are no explicit metrics to evaluate a flow control mechanism, it can provide hard guarantees about the system operation, *e.g.,* the memory consumption will never exceed a certain threshold.

Despite intertwined, those two mechanisms are independent in most systems. Schedulers, such as CFS (Linux's default scheduler) and BVT [8], control the CPU resources of the system and are agnostic to the number of processes. Operating systems, though, limit the number of processes, which is the unit of scheduling, that can be created, as a form of flow control for the available system memory. Systems such as ZygOS [29], perform connection-level scheduling in a µs-scale improving throughput at the latency SLO. The same systems need to limit the number of currently opened connections to avoid memory scarcity, but ignore the number of connections the system can serve under a certain latency SLO. TCP's congestion control decides when a sender can transmit bytes, which is a form of scheduling, while TCP's flow control makes sure that the receiver has available space in the socket buffer. Exceptions in this mechanism separation are deadline-aware schedulers [35] and certain schedulers for storage systems [14, 18].

Thus, we understand that the latency impact of those two mechanisms is significant. Independently of scheduling and flow control, the latency behavior as a function of the incoming load for any system is well-known and well-studied in queuing theory. As the offered load approaches the system saturation, latency goes to infinity since queuing time increases. A scheduling mechanism can push this curve to the right to achieve more throughput under the latency SLO, without being able to control maximum latency, though. Flow control controls queue depths at the receiver, thus can cap the maximum latency at saturation.

Although flow control is the right mechanism to implement tail-tolerance, as it caps the maximum latency and provides explicit guarantees, it is very coarse-grained in its current form. Our goal is to design and build an SLO-aware flow control mechanism that operates at the right abstraction, which is not memory availability, while remaining agnostic to existing scheduling and flow control mechanisms.

2.2 Datacenter RPCs

Remote Procedure Calls (RPCs) are the basis of communication both within and across datacenters. Any interaction that is described through a request-response pattern can be considered an RPC and different protocols have been proposed to deliver and serve RPCs, such as HTTP,gRPC, Thrift, *etc.*. Latency SLOs are, also, defined at the RPC boundaries as the percentage of requests that complete under a certain latency threshold. Thus, RPCs provide the right abstraction for any application-agnostic mechanism for tail-tolerance.

Datacenter applications usually communicate over RPCs in complex fan-out/fan-in patterns in which an aggregator node needs to contact several leaf nodes to compile a reply for a specific request. The end-to-end latency for the top request is determined by the slowest sub-request, as described by Dean et. al. [6]. This communication pattern, though, reveals a promising trade-off for system design.

Certain datacenter applications can trade-off harvest for yield [9], according to the specific latency SLOs they need to provide. Namely, latency-critical applications might be willing to ignore some of the slow sub-RPCs, and thus reduce the harvest and the quality of their response, in order to reply sooner back to the client. Consider for example web-search: a fast result is more useful than a complete result.

Mechanisms that trade-off harvest for yield can be implemented either proactively or reactively. Previously proposed systems, such as Zeta [15], terminate requests that exceed their given service time, and return partial answers to the client. Such reactive designs are steps towards tail-tolerance, but suffer from two main pitfalls. First, they are application specific — cancelling a request and removing transient state requires application knowledge. Second, they waste resources serving requests that are not used in the final answer instead of serving new requests. Instead, a proactive design, that avoids performing sub-RPCs rather than cancelling them, would reduce the need for application awareness and use resources more efficiently.

2.3 R2P2

Given that latency SLOs are expressed at the RPC boundaries, the RPC layer is the natural fit for a tail-tolerant mechanism. We chose to implement SVEN on top of R2P2 [21] due to its design choices that match our needs. We further describe why R2P2 could be easily extended with an application agnostic-mechanism for tail-tolerance to provide SLO compliance guarantees at the transport layer.

R2P2 [21] (**R**equest-**R**esponse-**P**air-**P**rotocol) is a transport protocol designed for datacenter RPCs. It exposes the request and response abstraction to the end-points and the network, thus making RPCs application-agnostic and the network RPC-aware. So, R2P2 already provides the right abstraction for the proposed mechanism.

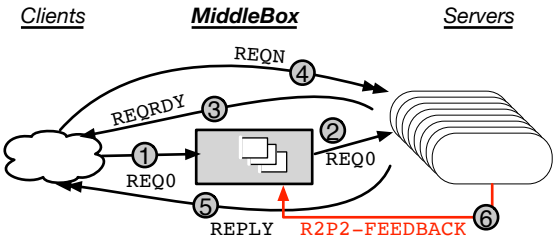


Figure 1: A request-reply exchange over R2P2. Only REQ0 goes through the policy enforcing middlebox.

R2P2 was designed with in-network RPC policy enforcement in mind. It separates the request and the reply streaming from the RPC policy enforcement that is offloaded to an in-network middlebox. Only the first packet of an multi-packet R2P2 request goes through the policy enforcer, thus reducing the IO bottleneck at the policy enforcing middlebox. Single packet requests first go through the middlebox and then to the destination server without an extra RTT. Given that R2P2 exposes the RPC semantics to the network, policy enforcement can be implemented in a programmable switch, thus eliminating the latency overhead and enabling line-rate processing speeds.

R2P2 optionally implements a tightly-coupled communication scheme between the RPC servers and the policy enforcing middlebox through the use of `FEEDBACK` messages. Servers send those messages to the middlebox after executing each RPC and middlebox interprets and uses those messages based on the policy it implements. For example, R2P2's request router used those messages to implement a novel scheduling policy called *Join-Bounded-Shortest-Queue* [21]. `FEEBACK` messages are completely application-agnostic and the R2P2 stack sends those automatically without the application intervention.

Figure 1 describes the packet exchange for a multi-packet request and a multi-packet reply on top of R2P2 with a generic policy enforcing middlebox. Only the first packet of the request (`REQ0`) goes through the middlebox, while request and reply streams bypass it (`REQN` and `REPLY`). Step 6 depicts the `FEEDBACK` message from the server to the middlebox.

Based on the above description we conclude that R2P2's design for in-network policy enforcement is an ideal basis for out proposed tail-tolerant mechanism. Our goal is to use the R2P2 infrastructure and implement SVEN as an R2P2 in-network enforced policy.

3 DESIGN

SVEN is a system designed to reduce the latency SLO violations for RPC services that run inside the datacenter by

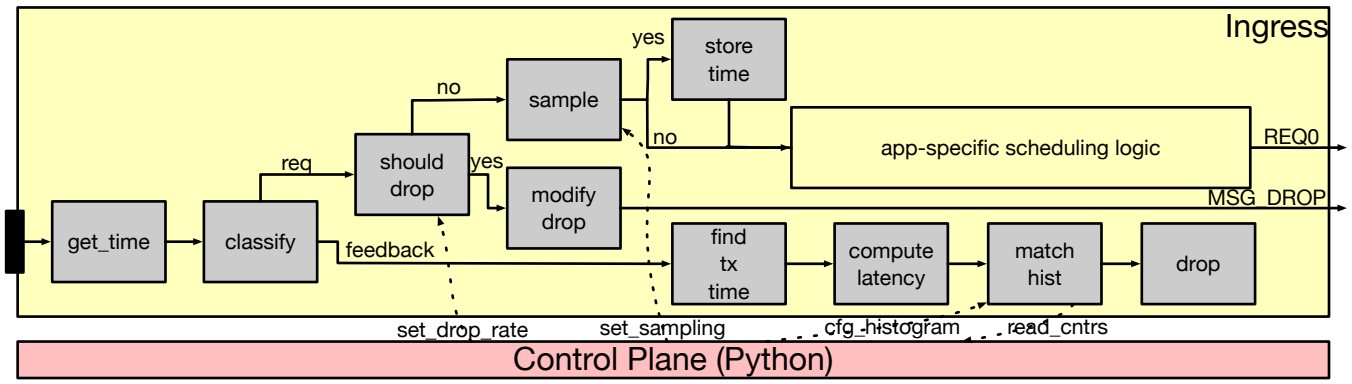


Figure 2: SVEN's dataplane sample implementation running in Tofino's ingress pipeline.

proactively controlling the incoming load based on an estimation of the current end-to-end latency distribution of the RPC service. Reducing the offered load will reduce queueing and as a result the end-to-end request latency.

We set the following requirements while designing SVEN: i) our solution should be application-agnostic and it should work for different service time distributions without need for re-configuration; ii) it should be independent of and complementary to any scheduling mechanism or policy both at the leaf or the aggregator nodes in a fan-out/fan-in application; iii) it should be implementable on programmable network devices, *e.g.,* P4 switches, to avoid adding any CPU overhead to the RPC servers or latency to the end-to-end result.

We design SVEN as an R2P2 middlebox (Figure 1) that can do SLO-aware RPC admission control for several servers. R2P2 [21] revealed the benefits of implementing an in-network RPC scheduling policy that come from the global view of the infrastructure. We place SVEN before the scheduling logic on the same middlebox so that the scheduler deals only with the requests to be executed and not with the ones that got rejected due to admission control. We split SVEN's functionality between a control and a dataplane.

3.1 SVEN Dataplane

SVEN's dataplane deals with the `REQ0` packets from the clients and the `FEEDBACK` messages from the servers. It is in charge of dropping incoming requests based on a drop rate defined by the control plane, and it also needs to measure RPC latencies so that the control plane can estimate the current RPC latency distribution.

The dataplane probabilistically decides whether to forward `REQ0` to one of the servers, or drop it, based on the drop rate set by the control plane. In case of a drop, the dataplane sends a `DROP_MSG` back to the client to avoid request timeouts. This way the client early on knows that the request will not be executed and it should not wait for it.

Incoming requests can be dropped uniformly or there can be different drop rates for different request types. If there are different request types, the control plane can set different drop rates per type. R2P2's header defines a message type field, as well as a policy field, that can be used to encode different request types in an application-agnostic manner, so that the control plane can apply the equivalent drop rate. Based on this mechanism each application can set different priorities for its requests corresponding to different drop rates. Also, this way an application can define requests that should not be dropped, *e.g.,* non read-only requests, acknowledging the risk of violating the latency SLO, though.

For the latency estimation, SVEN's dataplane takes timestamps when receiving `REQ0`s from clients, and `FEEDBACK`s from servers that signal a request completion. The difference between the `FEEDBACK` and the `REQ0` timestamps for the same request can be used as a proxy for the latency perceived by the RPC client. SVEN keeps the up-to-date latency distribution in the programmable switch dataplane through a histogram implemented as counters to a match-action table whose cells correspond to the latency histogram buckets. The dataplane can maintain different latency histograms for different request types. In the simplest case, SVEN needs one table, assuming one request type, and two buckets, for requests under and above the SLO.

There are different ways to match the `REQ0` and `FEEDBACK` timestamps to get the latency of a specific request. One approach is to add the `REQ0` timestamp in the `REQ0` packet itself and require the R2P2 stack to echo the timestamp back with the `FEEDBACK` message, similarly to TCP's timestamp option used for RTT estimation. The alternative is to keep the `REQ0` timestamp in the dataplane indexed with R2P2's 3-way tuple of `(src_ip, src_port, req_id)` and include the same tuple in the FEEDBACK message to do the matching. In our implementation we used the latter approach, because it required fewer changes to the R2P2 code base.

The current design assumes that servers send a FEEDBACK message for each request. However, this number can be reduced if it affects the server scalability. The middlebox may sample the timestamped RPCs using a specific request type, *e.g.,* timestamped request, so that the server only sends back a FEEDBACK message for certain requests.

Figure 2 describes the dataplane processing processing pipeline and its interactions with the control plane. Once a packet arrives, the dataplane first take a timestamp to avoid measuring any time spent in the middlebox. Then, there are different paths based on the packet type (REQ0 or FEEDBACK). In the case of REQ0, SVEN is completely independent and runs before the application specific scheduling logic, *e.g.,* JBSQ or random server selection.

3.2 Control plane

SVEN's control plane runs a control loop that computes the RPC drop rate to be applied based on the current latency distribution. The only configuration parameter for this control loop is the target latency SLO at a specific latency percentile, *e.g.,* 300µs at the 99-th percentile. The input to the control loop is the current estimation for the latency at the current latency percentile, coming from the dataplane. The output of the control loop is the drop-rate that the middlebox needs to apply to the incoming requests in order to reduce latency violations.

There are different ways to design this control plane, such as simple heuristics, control theory, or even online learning. The design and implementation of this control plane is orthogonal to the design of SVEN. In our proof-of-concept implementation we used a simple additive-increase-additive-decrease control over an exponentially weighted average estimation of the target-percentile latency. Listing 1 describes the control loop logic and the dataplane-control plane interaction. We acknowledge that this is not an optimal control plane design and it makes many sweeping simplifications, such as ignoring hysterisis, but its sole purpose is to showcase the usability of our design.

3.3 Client and server applications

SVEN is completely transparent to the client and server applications as it is implemented entirely in the transport protocol. SVEN depends on the clients' ability to deal with request rejections and request classification for different drop rates. R2P2's API already requires clients to define an error callaback function that deals with early rejections. Also, R2P2's API allows clients to define different request policies. Those policies can be used to apply different request rates or mark requests as non-droppable if they are critical. The above classification is non-SVEN specific, it can be used by other policy enforcing mechanisms, and it is already supported by the existing R2P2 API.

SVEN's proactive drop mechanism guarantees that requests reaching the server will be executed, thus avoiding the need for an application-specific cancellation mechanism as in Zeta [15]. From the server perspective the transmission of FEEDBACK messages are internal to the R2P2 stack and the application should not take special care.

```
STEP = 0.01
SLEEP_INTERVAL = 20 # in milliseconds
dr = 0              # drop rate
TARGET_P = 99       # percentile
while True:
  cntr = dataplane.readCounters()
  # Estimate percentile @ target SLO
  p = estimate_p(cntr)
  # e.g. below 99% for SLO@99-th
  if p < TARGET_P:
    dr = min(0.99, dr + STEP)
  else
    dr = max(0, dr - STEP)
  dataplane.apply_drop_rate(dr)
  sleep(SLEEP_INTERVAL)
```

Listing 1: SVEN's proof-of-concept control loop logic

4 EVALUATION

To evaluate the effectiveness of SVEN in transforming a generic low latency RPC service to a tail-tolerant system we implemented the above design on a Tofino [1] programmable switch, splitting the functionality between the P4 dataplane and a Python control plane. We run a series of synthetic microbenchmarks in which we control the service time distribution to investigate how SVEN performs across service times without application-specific configurations.

We used the open-source DPDK-based version of R2P2 [5], modified it as described in Section 3, and deployed it on a 16-core Xeon E5-2650 server connected with an Intel x520 10GbE NIC. Each core exposes its own Tx and Rx queues for a total of 16 independent workers. We ran the middlebox that implements the dataplane described in Figure 2 in a Barefoot Tofino v1 Edgecore Wedge100BF-32X.

In our experiments we use three different service time distributions with the same average service time of $\bar{S} = 10\mu s$: a fixed, an exponential, and a bimodal distribution in which 10% of the requests are 10 times slower than the rest, similar to the evaluation performed in R2P2 [21]. We consider an SLO at 300 µs for the 99-th percentile latency, and we use a random FIFO scheduling policy, meaning that incoming requests are randomly assigned to queues by the switch and are executed in order by each core. The scheduling policy is implemented by the *app-specific scheduling logic* of the

middlebox in Figure 2 and it is orthogonal to the SVEN design. We chose this policy for simplicity. On the client side we use the Lancet [20] load generator.

In our first experiment we run the server without SVEN to understand the system behaviour across different loads and identify the load level that will violate the latency SLO for each service time distribution. Figure 3 shows the 99-th percentile latency as a function of the achieved throughput for the three service time distributions. We also plot the 99-th percentile latency SLO at 300 µs in red, and the throughput that violates the SLO in dashed grey. We observe that different service time distributions violate the SLO at different load levels: 1.54 MRPS for fixed, 1.39 MRPS for exponential, and 1.22 MRPS for bimodal. SVEN's purpose is to keep the offered throughput below those levels.

In the second experiment, we evaluate how SVEN performs as the offered request load changes. We change the offered load every 20 seconds starting from 1 MRPS up to 1.8 MRPS which is beyond the system capacity. Note that the system capacity is 1.6 MRPS — 16 cores and average service time of $\bar{S} = 10\mu s$. We measure the achieved throughput and latency every second and plot the results in Figure 4.

In the throughput plots we include the offered load, the achieved throughput, the system capacity, and the throughput that violates the SLO as identified in the previous experiment. The difference between the orange (offered load) and blue (achieved throughput) lines correspond to the drop rate as identified by the control plane. We observe that SVEN manages to approximately identify the load that will violate the latency SLO (light grey line) for each distribution and maintain throughput (blue line) close to that threshold even when the offered load was much higher and despite the simplicity of the control loop. We, also, see that SVEN does not waste throughput to achieve tail-tolerance. When SVEN drops requests the achieved throughput stays close to the throughput that violates the SLO.

The latency plots, in the same figure, show the 99-th percentile latency as a function of time and the latency SLO of 300µs for the same experiment. We observe that the tail latency stays close to the target SLO, even when the offered load is above the system capacity (when the orange line is above the dark grey line in the throughput graphs). In such cases of extremely high loads beyond capacity, latency would be arbitrarily high without SVEN.

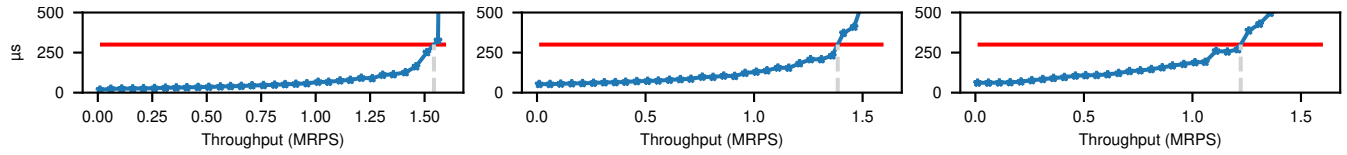
The latency oscillation is explained by the AIAD control policy that requires a latency violation to control the drop rate. Note that different service times lead to different oscillations. This is explained by the slope of the latency versus throughput curve at the point of the SLO violation. A drop rate misconfiguration can lead to a more significant SLO violation in the case of the fixed service time distribution where the slope is steeper compared to the bimodal case. We expect that a more robust and carefully tuned control loop will reduce the observed latency oscillation.

5 DISCUSSION AND FUTURE WORK

Layering and Placement: The current SVEN implementation runs in the network on P4 switches, already used for RPC scheduling in R2P2, is placed before the scheduling logic, and servers multiple servers. However, our design is not limited to programmable switches. An alternative would be to implement SVEN's control and dataplane within servers. Such a deployment assumes the server is able to accurately estimate the time a request spends waiting and being processed. This would require the server to timestamp the first packet of a request once it enters the system before any queueing time and the last last reply packet. To do so, servers should depend either on hardware timestamping at the NIC or on an asymmetric design with a dispatcher, similar to RamCloud [26] or Shinjuku [17]. Note though, that this approach uses CPU resources that could be used for RPC serving. To avoid CPU usage, SVEN could also run on a smartNIC on the serverside. The smartNIC has full visibility to incoming requests and outgoing replies and can easily estimate the time a request spends on the server by tracking the request reception and the reply transmission time. A server-based SVEN should not be combined with an in-network scheduler though, since the scheduler would waste resources to schedule requests that are eventually dropped at the server.

SVEN, being an in-network solution, poses an interesting deployment question regarding its placement inside a complex RPC service. In our evaluation we only looked at a single-level application and placed SVEN before the RPC server. However, in a more complicated fan-out/fan-in application SVEN can be employed both for the leaf and the aggregator nodes. Although it depends on the application type and whether it can leverage the latency vs completeness trade-off, we suggest employing tail-tolerant mechanisms at the higher levels of the application, *e.g.,* the aggregator nodes, letting other mechanisms, *e.g.,* scheduling to deal with the leaf nodes. This way SVEN can better capture the user perceived latency, as opposed to the latency of a leaf service.

Control plane: SVEN currently depends on a very simplistic control plane and it treats equally all requests. As future work we would like to look at different alternatives for the control loop design and investigate their impact on latency oscillation. Also, future control loops should consider different request types and compute different drop rates for each request type, as those are defined by the R2P2 request type or policy. The control loop can depend on either more complicated heuristics [30], control theory [27], or online learning [11]. Given the separation between the control and



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

Figure 3: 99-th Latency for different service time distributions with $\bar{S} = 10\mu s$ without SVEN. The red line shows the latency SLO@300µs. The vertical grey line shows the throughput that violates that SLO and it is shown with the same colour in Figure 4.

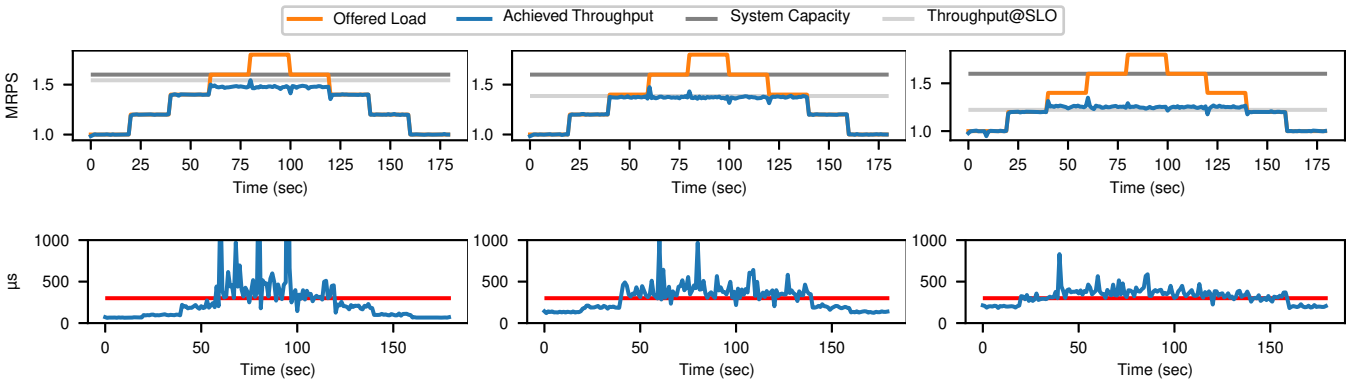


Figure 4: Throughput (MRPS) and 99-th Latency (µs) over time for a step load pattern and the same 3 service time distributions. The difference between the orange and the blue throughput lines corresponds to SVEN's drop rate. The red line shows the latency SLO@300µs.

the dataplane and the fact that the control loop only identifies the drop rate, thus not being on the critical path, the duration of the control loop could be in the order of 100s of µs. This can open up the design and implementation space for more accurate but slower implementations.

Dropped vs degraded requests: In our experiments we assumed that clients can leverage the latency vs completeness trade-off and did not re-issue the dropped requests. However, this will not always be the case. Depending on the deployment clients can have different options. First, we described that clients could prioritize requests or mark them as non-droppable based on existing R2P2 mechanisms. Also, in case of a replicated service clients can re-issue a dropped request to another server. The early rejection notification enables a more informed and less wasteful version of tied requests [6].

SVEN depends on explicit request drops to control latency in order to eliminate the need for server modification. Another approach that requires server support and cooperation, though, is one that marks requests as degraded instead of dropping them completely. So, as tail latency approaches the SLO SVEN changes the request type to `degraded-request` before forwarding them to the server based on the same probabilistic logic. A server would provide a reply of less quality back to the client, *e.g.,* an image with lower resolution, for a degraded request. Similar ideas have already been explored in previous works [10], and in different domains, *e.g.,* in the NDP congestion control scheme [13] that forwards only packet headers and drops the packet payload instead of dropping the entire packet.

Stricter Guarantees: SVEN is only the first step towards tail-tolerant systems and does not provide any strong guarantees at the moment, while it depends on the assumption that requests can be dropped. We believe that providing stronger tail-tolerance guarantees is viable but will require more complicated application analysis, such as such performance contracts [16]. Also, avoiding dropped or degraded requests can be achieved through redundancy similar to fault-tolerance. Combining performance verification, redundancy, and in-network control loops on top of the right abstractions can lead to by construction tail-tolerant systems.

6 RELATED WORK

The idea of dynamic admission control has been explored in different contexts [7, 32–34]. These approaches were either application specific or targeting different deployment environments. SVEN is a transport layer mechanism, thus application agnostic, and can provide guarantees in µs-scale for datacenter applications based on in-network programmability. In the storage domain, Reflex [18] controls application latency through achieved throughput, while Mittos [14] leverages early rejections for fast request re-issue. NeBula [31] implements dynamic RPC admission control in hardware. Prior work that is closest to SVEN is our previously proposed SLO-aware flow control for TCP [19]. SVEN does not use complex queueing formulas to identify the right rate and depends on a better-fitted request abstraction.

7 CONCLUSION

We advocate for tail-tolerance as a system design principle instead of a best-effort system metric. As a first step in this direction, we propose SVEN, an application-agnostic system for in-network SLO-aware RPC admission control implemented as part of the R2P2 transport protocol.

REFERENCES

[1] Barefoot Networks. 2018. Tofino Product Brief. https://barefootnetworks.com/products/brief-tofino/. (2018).

[2] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[3] 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.

[4] Navin Budhiraja and Keith Marzullo. 1992. Highly-Available Services Using the Primary-Backup Approach.. In *Workshop on the Management of Replicated Data*. 47–50.

[5] DCSL. 2020. R2P2 Github repository. https://github.com/epfl-dcsl/r2p2. (2020).

[6] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[7] Peter Druschel and Gaurav Banga. 1996. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems.. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI)*. 261–275.

[8] Kenneth J. Duda and David R. Cheriton. 1999. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedular.. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*. 261–276.

[9] Armando Fox and Eric A. Brewer. 1999. Harvest, Yield and Scalable Tolerant Systems.. In *Proceedings of The 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*. 174–178.

[10] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. 1997. Cluster-Based Scalable Network Services.. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. 78–91.

[11] Robert Glaubius, Terry Tidwell, Christopher D. Gill, and William D. Smart. 2010. Real-Time Scheduling via Reinforcement Learning.. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI)*. 201–209.

[12] Jim Gray. 1985. Fault Tolerance in Tandem Systems.. In *Proceedings of the 1985 International Workshop on High-Performance Transaction Systems (HTPS)*.

[13] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance.. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 29–42.

[14] 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 *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 168–183.

[15] Yuxiong He, Sameh Elnikety, James R. Larus, and Chenyu Yan. 2012. Zeta: scheduling interactive services with partial execution.. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*. 12.

[16] Rishabh R. Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina J. Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions.. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*. 517–530.

[17] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency.. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*. 345–360.

[18] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash.. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*. 345–359.

[19] Marios Kogias and Edouard Bugnion. 2018. Flow control for Latency-Critical RPCs.. In *Proceedings of the 2018 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*. 15–21.

[20] Marios Kogias, Stephen Mallon, and Edouard Bugnion. 2019. Lancet: A self-correcting Latency Measuring Tool.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 881–896.

[21] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 863–880.

[22] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[23] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities.. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 221–235.

[24] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*. 305–319.

[25] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*. 361–378.

[26] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*

33, 3 (2015), 7:1–7:55.

[27] Sujay S. Parekh, Neha Gandhi, Joseph L. Hellerstein, Dawn M. Tilbury, T. S. Jayram, and Joseph P. Bigus. 2001. Using Control Theory to Achieve Service Level Objectives In Performance Management.. In *Integrated Network Management*. 841–854.

[28] 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.

[29] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks.. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 325–341.

[30] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy proportionality and workload consolidation for latency-critical applications.. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*. 342–355.

[31] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NeBuLa RPC-Optimized Architecture. *[Proceedings of ISCA 2020]* (2020), 14. http://infoscience.epfl.ch/record/277391

[32] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. 2001. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers.. In *USENIX Annual Technical Conference*. 189–202.

[33] Matt Welsh and David E. Culler. 2002. Overload management as a fundamental service design primitive.. In *ACM SIGOPS European Workshop*. 63–69.

[34] Matt Welsh and David E. Culler. 2003. Adaptive Overload Control for Busy Internet Servers.. In *USENIX Symposium on Internet Technologies and Systems*.

[35] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Antony I. T. Rowstron. 2011. Better never than late: meeting deadlines in datacenter networks.. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 50–61.